

How LangChain Can Help You Create LLM-Powered Chatbots, Summarizers and More 🦜🔗

Estimated time needed: 120 minutes

Major Update:

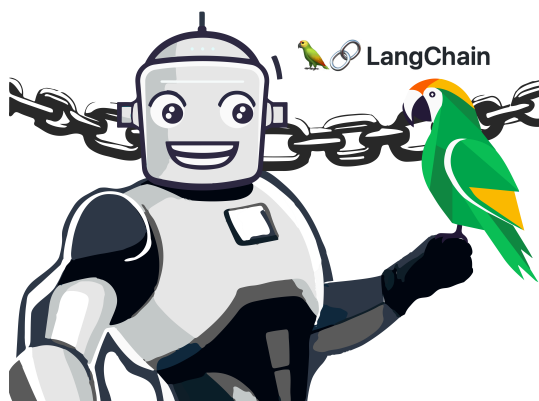
No need to insert your OpenAI key anymore! The Skills Network team has streamlined the process by taking care of the API. Now, you'll automatically receive results from the GPT 3.5 Turbo model! 🚀🤖

LangChain is a Python library that helps you with natural language processing. It gives you access to many powerful language models from different providers like OpenAI, Cohere, Huggingface Hub, IBM WatsonX, and other LLM models.

But that's not all! LangChain is not just about integration, it's about versatility. It comes equipped with document loaders that can handle a wide array of formats - from text and HTML to PDF and PowerPoint, even emails. This means you can extract and process information from virtually any document, making it a Swiss Army knife of data extraction.

And there's more! LangChain is also your key to creating engaging conversational interfaces. With its innovative tools for crafting conversation chains and agents, you can design interactive dialogues with language models. Combine these conversation chains with memory and prompt templates, and voila! You have conversations that are so human-like, they blur the line between man and machine.

So, whether you're a developer looking to create a conversational AI, or a data scientist needing to extract information from a myriad of document formats, LangChain is the Python library that brings your language processing tasks to life.



Learning Objectives

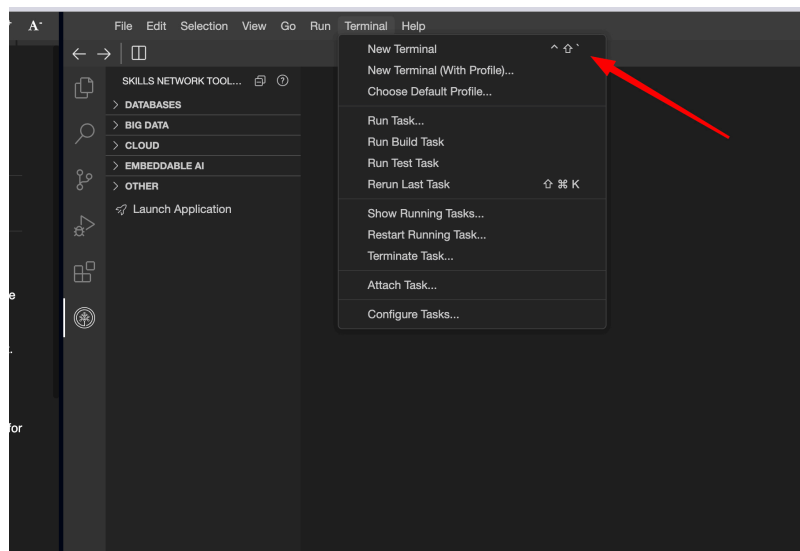
LangChain is a versatile Python library that simplifies building applications with Large Language Models (LLMs). It offers:

1. **Universal Interface:** Seamless integration with various LLM models.
2. **Prompt Management:** Tools for handling, optimizing, and serializing prompts.
3. **Conversation Chains:** Enables creation of complex dialogues with LLMs.
4. **Memory Interface:** Facilitates storing and retrieving model information.
5. **Indexes:** Utility functions for loading custom text data.
6. **Agents and Tools:** Allows setting up agents that can use tools like Google Search, Wikipedia, or a calculator.

Setting up the environment and installing libraries

In this project, to build an AI app, we will utilize [Gradio](#) interface provided by hugging face.

Let's set up the environment and dependencies for this project. Open up a new terminal and make sure you are in the home/project directory. Open a terminal:



Create a python virtual environment and install Gradio using the following commands in the terminal:

1. 1

```

2. 2
3. 3

1. pip3 install virtualenv
2. virtualenv my_env # create a virtual environment my_env
3. source my_env/bin/activate # activate my_env

```

Copied! Executed!

Then, install the required libraries in the environment:

```

1. 1
2. 2

1. # installing required libraries in my_env
2. pip install langchain==0.1.0 openai==0.28 gradio==4.21.0 chromadb tiktoken

```

Copied! Executed!

Now, our environment is ready to create python files.

Quickstart Gradio

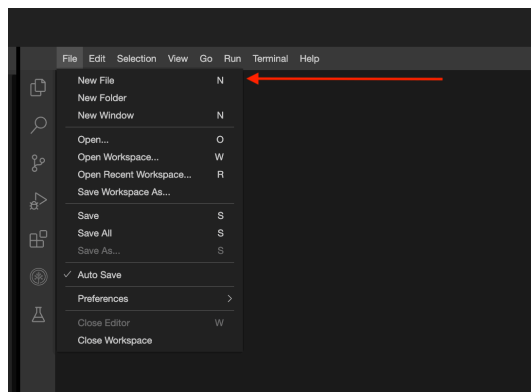
introducing gradio

to demonstrate langchain application we need to have web interface and gradio provides it...

Creating a simple demo

Through this project, we will create different LLM applications with Gradio interface. Let's get familiar with Gradio by creating a simple app:

Still in the project directory, create a Python file and name it `hello.py`.



Open `hello.py`, paste the following Python code and save the file.

```

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8

1. import gradio as gr
2.
3. def greet(name):
4.     return "Hello " + name + "!"
5.
6. demo = gr.Interface(fn=greet, inputs="text", outputs="text")
7.
8. demo.launch(server_name="0.0.0.0", server_port= 7860)

```

Copied!

The above code creates a **gradio.Interface** called `demo`. It wraps the `greet` function with a simple text-to-text user interface that you could interact with.

The **gradio.Interface** class is initialized with 3 required parameters:

- `fn`: the function to wrap a UI around
- `inputs`: which component(s) to use for the input (e.g. "text", "image" or "audio")
- `outputs`: which component(s) to use for the output (e.g. "text", "image" or "label")

The last line `demo.launch()` launches a server to serve our `demo`.

Launching the demo app

Now go back to the terminal and make sure that the `my_env` virtual environment name is displayed at the beginning of the line. Now run the following command to execute the Python script.

```

1. 1

1. python3 hello.py

```

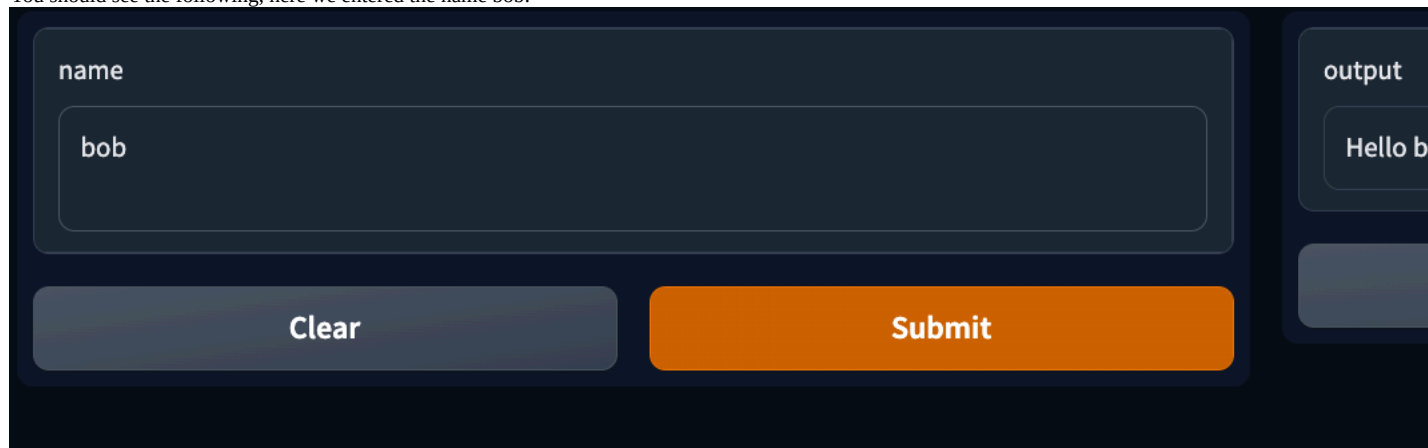
Copied! Executed!

As the Python code is served by local host, click on the button below and you will be able to see the simple application we just created. Feel free to play around with the input and output of the web app!

Click here to see the application:

Web Application

You should see the following, here we entered the name bob:



If you finish playing with the app and want to exit, **press ctrl+c in the terminal and close the application tab.**

You just had a first taste of the Gradio interface, it's easy right? If you wish to learn a little bit more about customization in Gradio, you are invited to take the guided project called **Bring your Machine Learning model to life with Gradio**. You can find it under **Courses & Projects** on cognitiveclass.ai!

For the rest of this project, we will use Gradio as an interface for LLM apps.

1. Models: Generic Interface for Many LLMs



LangChain is a package that provides a generic interface to many foundation models, including Large Language Models (LLMs). It simplifies prompt management and optimization, and acts as a central interface to other components. This generic interface is designed to work with various LLM providers such as OpenAI, Cohere, Hugging Face, Anthropic, and AI21. LangChain is not a provider of LLMs, but rather provides a standard interface through which you can interact with various LLMs.

Let's create simple chatbot using langchain.

Creating a simple chatbot

The following script is set up to send a request to the OpenAI API using the LangChain library. The request is to generate a response to the prompt "tell me interesting fact about potato". The response from the API is then printed to the console. You can specify which OpenAI LLM model you want to select (e.g. `model_name="gpt-3.5-turbo"`).

To get an OpenAI API key for ChatGPT, you can follow these steps:

1. Visit OpenAI's official website on your preferred browser [click here visiting the website](https://openai.com).
2. Click on "Log-In" and add your email to create an account.
3. Once logged in, click on the "View API Keys" icon located in the top-right corner of the screen.

4. Click on “Create an API Key” to generate your ChatGPT API Key.

As of June 2023, new free trial users receive \$5 (USD) worth of credit which expires after three months.

Still in the project directory, create a Python file and name it `simple_llm.py`.

Open `simple_llm.py`, paste the following Python code and save the file.

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11

1. import os
2. from langchain.llms import OpenAI
3.
4. # Set the environment variable "OPENAI_API_KEY" to your OpenAI API key. This is required to authenticate with the OpenAI API.
5. os.environ["OPENAI_API_KEY"] = "YOUR API KEY"
6.
7. # Specifying the LLM model
8. gpt3 = OpenAI(model_name="gpt-3.5-turbo" )
9.
10. text = "tell me interesting fact about potato"
11. print(gpt3.invoke(text))
```

Copied!

Now, go back to the terminal and make sure that the `my_env` virtual environment name is displayed at the beginning of the line, and run the following command to execute the Python script.

```
1. 1
1. python3 simple_llm.py
```

Copied!

Executed!

The output of the model will be printed out in terminal.

To create an App for this simple chatbot, let's use Gradio.

Replace the following code in `simple_llm.py`.

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16

1. import os
2. from langchain.llms import OpenAI
3. #from langchain.chat_models import ChatOpenAI
4. import gradio as gr
5.
6. # assign API key (no need to specify your api key as it is taken care of)
7. # os.environ["OPENAI_API_KEY"] = "YOUR API KEY" # if you running in cloudIDE environment, you dont need to insert API
8.
9. gpt3 = OpenAI(model_name="gpt-3.5-turbo" )
10.
11. def chatbot(inpt):
12.     return gpt3.invoke(inpt)
13.
14. demo = gr.Interface(fn=chatbot, inputs="text", outputs="text")
15.
16. demo.launch(server_name="0.0.0.0", server_port= 7860)
```

Copied!

Now run the code:

```
1. 1
1. python3 simple_llm.py
```

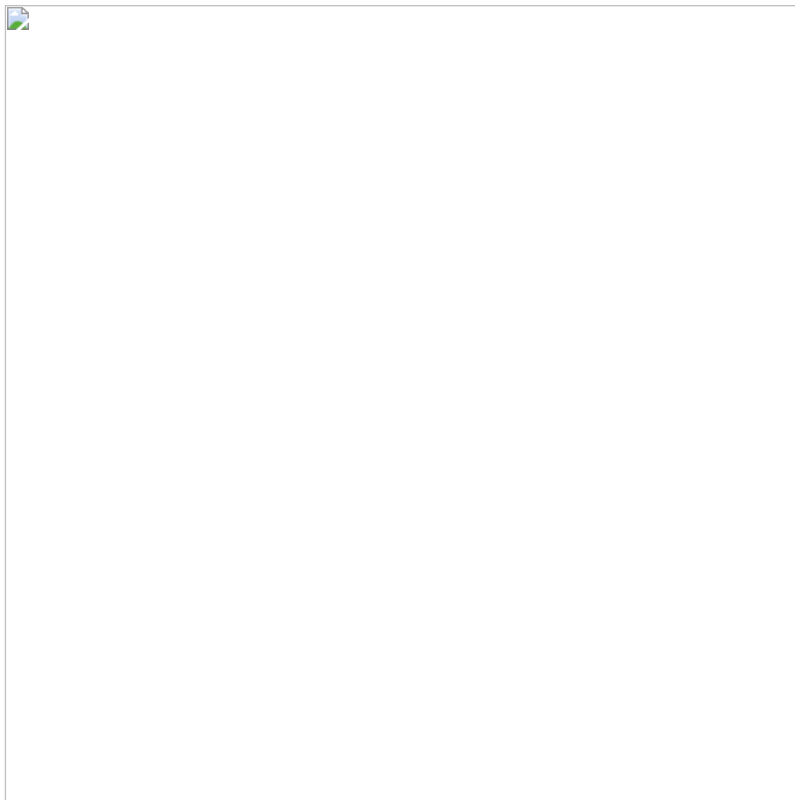
Copied!

Executed!

Click here to run the app:

Web Application

You should see the following, here we entered the name ‘tell a joke using potato’:



If you finish playing with the app and want to exit, **press `ctrl+c` in the terminal and close the application tab.**

You just had a first taste of the Gradio interface, it's easy right? If you wish to learn a little bit more about customization in Gradio, you are invited to take the guided project called **Bring your Machine Learning model to life with Gradio**. You can find it under **Courses & Projects** on cognitiveclass.ai!

you can also use HuggingFaceHub and Cohere to building LLM model

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8

1. # using Hugging FaceHub
2. from langchain import Hugging FaceHub
3.
4. os.environ["HUGGINGFACEHUB_API_TOKEN"] = "YOUR API KEY"
5. llm = HuggingFaceHub(repo_id="google/flan-t5-xl")
6. text = "tell me interesting fact about potato"
7. result = llm(text)
8. print(result)
```

Copied!

```
1. 1
2. 2
3. 3

1. # using Cohere
2. from langchain.llms import Cohere
3. cohere = Cohere(model='command-xlarge')
```

Copied!

2. Working With Prompts

In programming, we now use “prompts” as inputs for models. These prompts are not fixed; they’re made from different parts. LangChain helps us build these prompts easily. It also helps manage and optimize them.

One of the key features of LangChain is the ability to create a prompt template. This template takes the user’s input and transforms it into a final prompt for the model. This approach allows for a more dynamic and flexible interaction with the model, enhancing its usability and effectiveness.

Creating a Cover Letter Bot

Using LangChain’s prompt template, you have the opportunity to develop an application that accepts specific inputs such as position, company, and skills, ultimately generating a customised cover letter based on these parameters.

To begin, we need to create a new Python file. Let’s name it: `prompt_with_template.py`.

Next, let’s focus on developing our cover letter creator. We will design a web-based interface with Gradio, which will generate a tailored cover letter based on the user’s input.

```
1. 1
```

```

2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17
18. 18
19. 19
20. 20
21. 21
22. 22
23. 23
24. 24
25. 25
26. 26
27. 27
28. 28
29. 29
30. 30
31. 31
32. 32
33. 33
34. 34
35. 35
36. 36
37. 37
38. 38

1. import gradio as gr
2. from langchain.prompts import PromptTemplate
3. import os
4. from langchain.llms import OpenAI
5.
6. # you dont need to insert API
7. # openai_api_key = "YOUR API KEY"
8. # os.environ["OPENAI_API_KEY"] = openai_api_key
9.
10. # initialize the models
11. llm = OpenAI(
12.     model_name="gpt-3.5-turbo"
13. )
14.
15. # Define a PromptTemplate to format the prompt with user input
16. prompt = PromptTemplate(
17.     input_variables=["position", "company", "skills"],
18.     template="Dear Hiring Manager,\n\nI am writing to apply for the {position} position at {company}. I have experience in {skills}."
19. )
20.
21. # Define a function to generate a cover letter using the llm and user input
22. def generate_cover_letter(position: str, company: str, skills: str) -> str:
23.     formatted_prompt = prompt.format(position=position, company=company, skills=skills)
24.     response = llm(formatted_prompt)
25.     return response
26.
27. # Define the Gradio interface inputs
28. inputs = [
29.     gr.Textbox(label="Position"),
30.     gr.Textbox(label="Company"),
31.     gr.Textbox(label="Skills")
32. ]
33.
34. # Define the Gradio interface output
35. output = gr.Textbox(label="Cover Letter")
36.
37. # Launch the Gradio interface
38. gr.Interface(fn=generate_cover_letter, inputs=inputs, outputs=output).launch(server_name="0.0.0.0", server_port= 7860)

```

Copied!

Here's a brief explanation of what each part does:

1. Set up OpenAI API key, Initialize the OpenAI model: The script initializes the OpenAI model using the API key and the model name.
2. Define a PromptTemplate: The script defines a PromptTemplate that formats a cover letter based on the user's input for position, company, and skills.
3. Define a function to generate a cover letter: The script defines a function that takes the user's input, formats the prompt using the PromptTemplate, sends the prompt to the OpenAI model, and returns the model's response.
4. Define the Gradio interface inputs and output: The script defines the inputs and output for the Gradio interface. The inputs are text boxes for the position, company, and skills, and the output is a text box for the generated cover letter.
5. Launch the Gradio interface: Finally, the script launches the Gradio interface. The interface will call the generate_cover_letter function when the user submits their input, and display the generated cover letter in the output text box. The interface is launched on the local machine (0.0.0.0) at port 7860.

Now run the code:

- ```

1. 1
1. python3 prompt_with_template.py

```

Copied!

Executed!

Web Application

Here is a sample output:

Position

data scientist

Company

google

Skills

python, sql, power BI

Clear

Submit

output

Dear Hiring Manager,

I am writing to apply for the data scientist position at Google. I am confident that my experience in data analysis, machine learning, and programming languages like Python, SQL, and Power BI will be an asset to your team.

I am a highly organized and detail-oriented individual who is passionate about data and analytics. I have an excellent understanding of how to use data to draw insights and make informed decisions.

I am excited to join a team of like-minded professionals and use my skills to make an impact. I am confident that I can be an asset to your team and contribute to the success of your company.

Thank you for your time and consideration. I look forward to hearing from you.

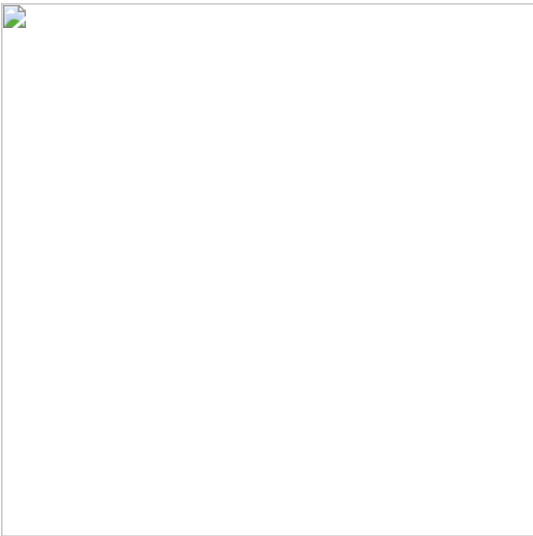
Sincerely,  
[Your Name]

To stop the app press `ctrl + c`.

Therefore, a prompt template refers to a reproducible way to generate a prompt. It contains a text starting (`template`), that can take in a set of parameters from the end user and generate a prompt.

Exercise: Create a Template that Provides “how to step by step”.

Your code should provdie step by step of the input prompt, e.g. here we entered the ‘How to backflip’:



► Click here for the answer

Run your python code in termical and start application:

Web Application

If you finish playing with the app and want to exit, **press ctrl+c in the terminal and close the application tab.**

### 3. Chains: The Sequences of LLM Calls

In LangChain, “Chains” refer to sequences of Large Language Model (LLM) calls. They allow you to go beyond a single LLM call and involve sequences of calls, which could be to an LLM or a different utility. This enables more complex workflows and applications.

Chains are particularly useful when you want to sequence together multiple LLM calls or other utilities to create more complex applications. They allow you to combine primitives and LLMs to process user input, generate prompts, and more.

#### Customer Support Chatbot



Let's create the customer support chatbot, using chain of commands. Create a new python file and call it `chain_customerSupport.py`.

The app is a simple chain using LangChain that takes the user input, formats the prompt with it, and then sends it to the LLM. It has a web interface where you can enter a customer complaint and see the identified issue, and provides a solution.

```

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17
18. 18
19. 19
20. 20
21. 21
22. 22
23. 23
24. 24
25. 25
26. 26
27. 27

1. import gradio as gr
2. from langchain.chains import LLMChain
3. from langchain.llms import OpenAI
4. from langchain.prompts import PromptTemplate
5. import os
6.
7. # setting up the LLM
8.
9. os.environ["OPENAI_API_KEY"] = "YOUR API KEY"
10. llm = OpenAI(
11. model_name="gpt-3.5-turbo"
12.)
13.
14. def handle_complaint(complaint: str) -> str:
15. #Create an instance of the LLM with a temperature value of 0.9 (higher values make the output more random).
16.
17.
18. # Define the prompt template for the complaint handling
19. prompt = PromptTemplate(input_variables=["complaint"], template="I am a customer service representative. I received the followir
20.
21. # Create a language model chain with the defined prompt template
22. chain = LLMChain(llm=llm, prompt=prompt)
23. return chain.run(complaint)
24.
25. # Create a Gradio interface for the handle_complaint function
26. iface = gr.Interface(fn=handle_complaint, inputs="text", outputs="text")
27. iface.launch()

```

Copied!

Now run the code:



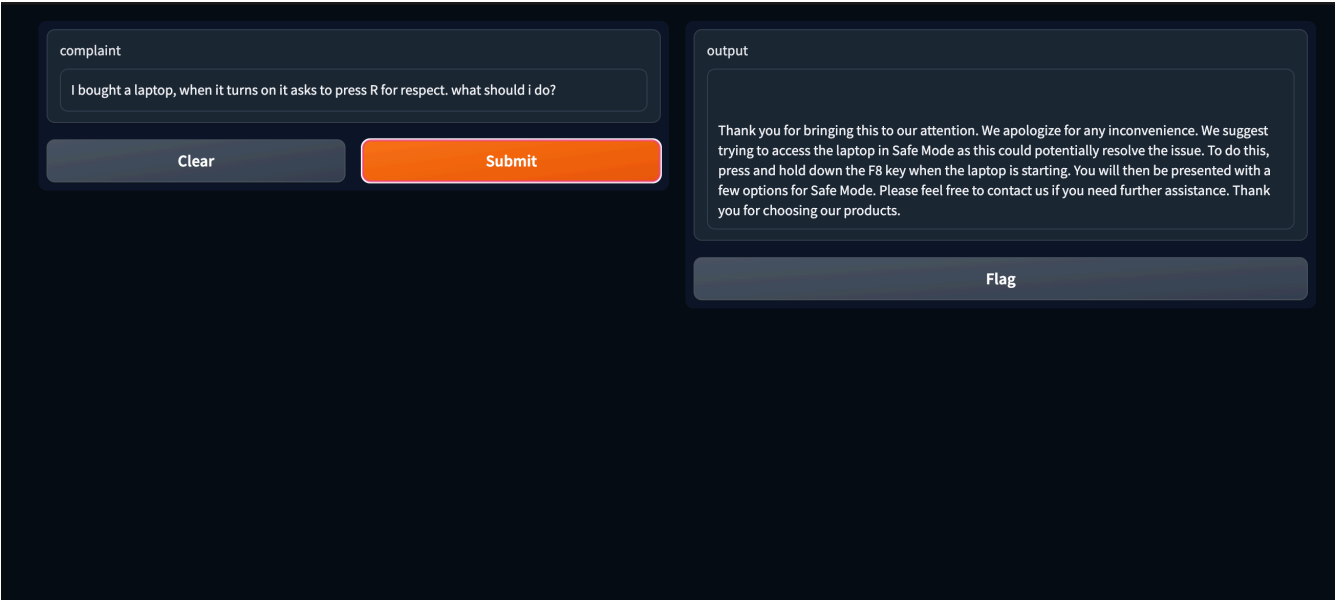
```
1. 1
1. python3 chain_customerSupport.py
```

Copied! Executed!

Click here to run the app:

Web Application

You should see the following:



Press Cntrl + c to stop the application.

# 4. Indexes: Utility Functionality to Load and Store Your Files

LangChain provides a dedicated module known as the ‘index’ module, designed specifically to enhance the interaction between Language Learning Models (LLMs) and various documents. It equips the LLMs with the ability to effectively navigate through a range of indexes, thereby enhancing their retrieval capabilities. The primary function of these indexes lies in their power to fetch the most relevant documents in response to a user’s query, a process facilitated through a unique interface called the “Retriever”.

While indexes do serve additional purposes within the LangChain framework, their main application is centered around document retrieval, particularly for unstructured data such as text documents. This module is primarily designed to focus on indexing and retrieving such unstructured data. For example, it can load data and provides vector store interface to make it store and searchable.

## Ask-Your-Documents Bot



Let’s set up a text summarization bot using the langchain library. The Gradio interface allows users to input a text query, which is then used to summarize the loaded text data using the created index. The summarized result is displayed as the output in the interface.

Create a new python file and call it summarizer.py and copy the following:

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
```

```

14. 14
15. 15
16. 16
17. 17
18. 18
19. 19
20. 20
21. 21
22. 22
23. 23
24. 24
25. 25
26. 26
27. 27
28. 28
29. 29

1. from langchain.document_loaders import TextLoader
2. from langchain.indexes import VectorstoreIndexCreator
3. import wget
4. import os
5. import gradio as gr
6.
7. # link to a text document
8. url = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMSkillsNetwork-GPXX0W2REN/images/state_of_the_union.txt"
9. output_path = "state_of_the_union.txt" # Local the file
10.
11. # Check if the file already exists
12. if not os.path.exists(output_path):
13. # Download the image using wget
14. wget.download(url, out=output_path)
15.
16. loader = TextLoader('state_of_the_union.txt')
17.
18. # Load the data loader
19. data = loader.load()
20.
21. # Create the index instance that makes it searchable
22. index = VectorstoreIndexCreator().from_loaders([loader])
23.
24. # Running into Gradio
25. def summarize(query):
26. return index.query(query)
27.
28. iface = gr.Interface(fn=summarize, inputs="text", outputs="text")
29. iface.launch(server_name="0.0.0.0", server_port= 7860)

```

Copied!

Let's go through the code step by step:

1. The code checks if the file specified by `output_path` already exists using the `os.path.exists()` function. If the file doesn't exist, it proceeds to download the file using the `wget.download()` function. This step avoids duplication of file from each run.
2. The `loader` variable is assigned an instance of the `TextLoader` class, initialized with the path to the downloaded text file.
3. The `data` variable is assigned the loaded text data using the `load()` method of the loader instance.
4. An index is created using the `VectorstoreIndexCreator` class, and the `from_loaders()` method is called on it, passing the loader instance as an argument. This step creates a vector representation of the text data for efficient querying.
5. The Gradio interface is created using the `gr.Interface` class, initialized with the `summarize()` function as the callable function, and specifying the input and output types as "text".

Now run the code:

1. 1
1. `python3 summarizer.py`

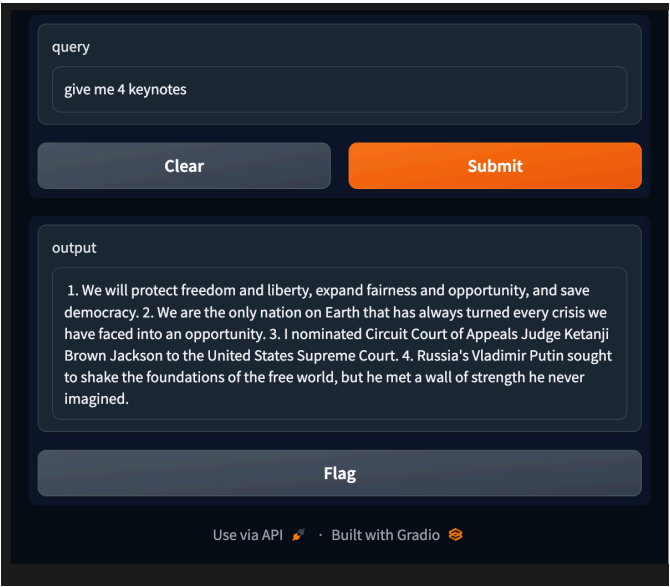
Copied!

Executed!

Click here to run the app:

Web Application

You should see the following:

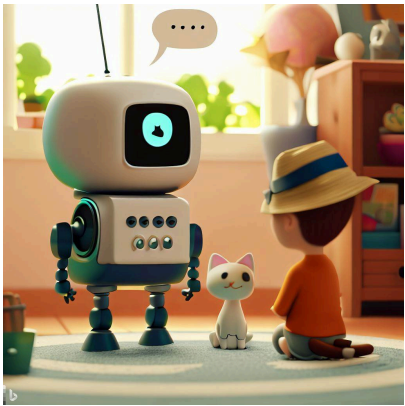


5. Memory: An Interface for Memory Implementation

LangChain provides a **standard interface for memory**, which helps maintain state between chain or agent calls. It also offers a **range of memory implementations** and examples of chains or agents that use memory. Memory is a crucial concept in LangChain, as it involves persisting state between calls of a chain/agent.

for example you can store chat story, and ask follow up question.

Chatbot Buddy



Let’s create interactive chatbot with memory. let’s create the file `memory.py`.

To do so we need following steps:

- Configures memory for the conversation: This allows the chatbot to maintain context from prior exchanges in the conversation. Two types of memory are used: one to store the recent conversation turns and another to generate a summarized version of the conversation.
- Sets up the language model: The code sets up an instance of the GPT-3 model provided by OpenAI. The temperature parameter, set to 0 here, determines the randomness of the model’s outputs. A temperature of 0 makes the output deterministic, meaning it will always produce the same output given the same input.
- Creates the conversation handler: This is done using the ConversationChain object from the LangChain library. This object manages the conversation flow and uses the memory and language model configurations set up in the previous steps.
- Builds the web interface using Gradio: The interface consists of a chatbox for the conversation, a textbox for user input, and a button to clear the conversation.

- 1. 1
- 2. 2
- 3. 3
- 4. 4
- 5. 5
- 6. 6
- 7. 7
- 8. 8
- 9. 9
- 10. 10
- 11. 11
- 12. 12
- 13. 13
- 14. 14
- 15. 15
- 16. 16
- 17. 17
- 18. 18
- 19. 19
- 20. 20
- 21. 21
- 22. 22

```
23. 23
24. 24
25. 25
26. 26
27. 27
28. 28
29. 29
30. 30
31. 31
32. 32
33. 33
34. 34
35. 35
36. 36
37. 37
38. 38
39. 39
40. 40
41. 41
42. 42
43. 43
44. 44
45. 45
46. 46
47. 47
48. 48
49. 49
50. 50
51. 51
52. 52
53. 53
54. 54
55. 55
56. 56
57. 57
58. 58
59. 59
60. 60
61. 61
62. 62
63. 63
64. 64
65. 65
66. 66
67. 67
68. 68
69. 69
70. 70
71. 71
```

```
1. # Import necessary libraries
2. from langchain.llms import OpenAI
3. from langchain.prompts import PromptTemplate
4. from langchain.chains import ConversationChain
5. from langchain.memory import ConversationBufferWindowMemory, CombinedMemory, ConversationSummaryMemory
6. import os
7. import gradio as gr
8. import time
9.
10.
11. # Set up conversation memory
12. # This memory will store the last k turns of conversation
13. conv_memory = ConversationBufferWindowMemory(
14. memory_key="chat_history_lines",
15. input_key="input",
16. k=1
17.)
18.
19. # This memory will store a summary of the conversation
20. summary_memory = ConversationSummaryMemory(llm=OpenAI(), input_key="input")
21.
22. # Combine the two memories
23. memory = CombinedMemory(memories=[conv_memory, summary_memory])
24.
25. # Define the template for the conversation prompt
26. _DEFAULT_TEMPLATE = """The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of
27.
28. Summary of conversation:
29. {history}
30. Current conversation:
31. {chat_history_lines}
32. Human: {input}
33. AI: ""
34.
35. # Create the prompt from the template
36. PROMPT = PromptTemplate(
37. input_variables=["history", "input", "chat_history_lines"], template=_DEFAULT_TEMPLATE
38.)
39.
40. # Set up the language model
41. llm = OpenAI(model_name="gpt-3.5-turbo")
42.
43. # Set up the conversation chain
44. # This object will handle the conversation flow
45. conversation = ConversationChain(
46. llm=llm,
47. verbose=True,
48. memory=memory,
49. prompt=PROMPT
50.)
51.
52. # Set up the Gradio interface
53. with gr.Blocks() as demo:
54. chatbot = gr.Chatbot() # The chatbot object
```

```

55. msg = gr.Textbox() # The textbox for user input
56. clear = gr.Button("Clear") # The button to clear the chatbox
57.
58. # Define the function that will handle user input and generate the bot's response
59. def respond(message, chat_history):
60. bot_message = conversation.run(message) # Run the user's message through the conversation chain
61. chat_history.append((message, bot_message)) # Append the user's message and the bot's response to the chat history
62. time.sleep(1) # Pause for a moment
63. return "", chat_history # Return the updated chat history
64.
65. # Connect the respond function to the textbox and chatbot
66. msg.submit(respond, [msg, chatbot], [msg, chatbot])
67.
68. # Connect the "Clear" button to the chatbot
69. clear.click(lambda: None, None, chatbot, queue=False)
70.
71. demo.launch(server_name="0.0.0.0", server_port= 7860)

```

Copied!

Now run the code:

1. 1

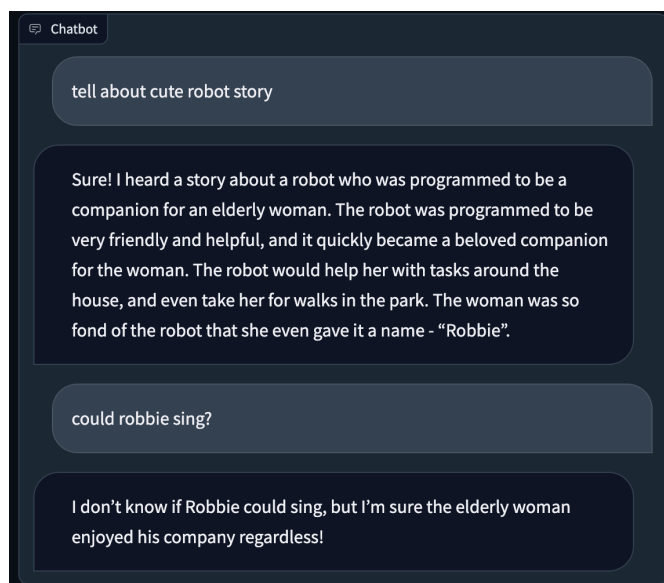
1. python3 memory.py

Copied!

Executed!

Click here to run the app:

Web Application



Let's break down the code:

Setting up memory: LangChain provides different types of memory that can be used to track the conversation. Here, you're setting up two types of memory: ConversationBufferWindowMemory and ConversationSummaryMemory. The first one stores a certain number (k) of the latest conversation turns, while the second one generates a summarized version of the conversation. Both are then combined into a CombinedMemory.

Defining the prompt: You define a template for the prompt that is given to the language model. This template structures the conversation in a way that makes it easier for the model to generate appropriate responses.

Setting up the language model: You use the OpenAI model with a temperature parameter of 0, which means the output will be deterministic.

Setting up the conversation: You set up a ConversationChain object, which is the main object that handles the conversation flow.

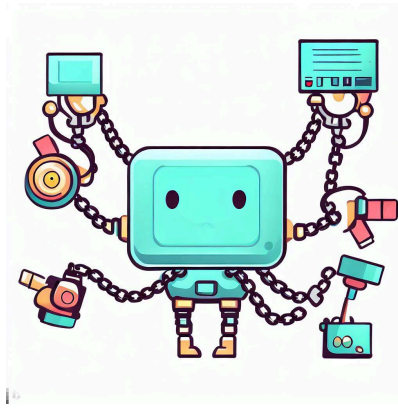
Setting up the Gradio interface: You use the Gradio library to create an interactive web interface for the chatbot. You define a Chatbot object, a Textbox for user input, and a "Clear" button. The respond function is defined to handle the conversation flow, which takes a message as input, runs it through the conversation chain, and appends the generated response to the chat history.

## 6. Agents and Tools: Set Up Agents That Can Use Tools Like Google Search or Wikipedia or Calculator

"Agents" in LangChain seem to be entities that can interact with different tools for specific use cases. For example, an agent might interact with a SQL database using a specific set of tools designed for that purpose.

In LangChain, "tools" are ways that an agent can use to interact with the outside world. They provide functionality for various tasks. For example, with tools, LangChain's agents can search the web, do math, run code, and more. The LangChain library provides these tools for use. There are also "toolkits", which are groups of tools designed for a specific use case.

### Python Runner Bot



Let's create

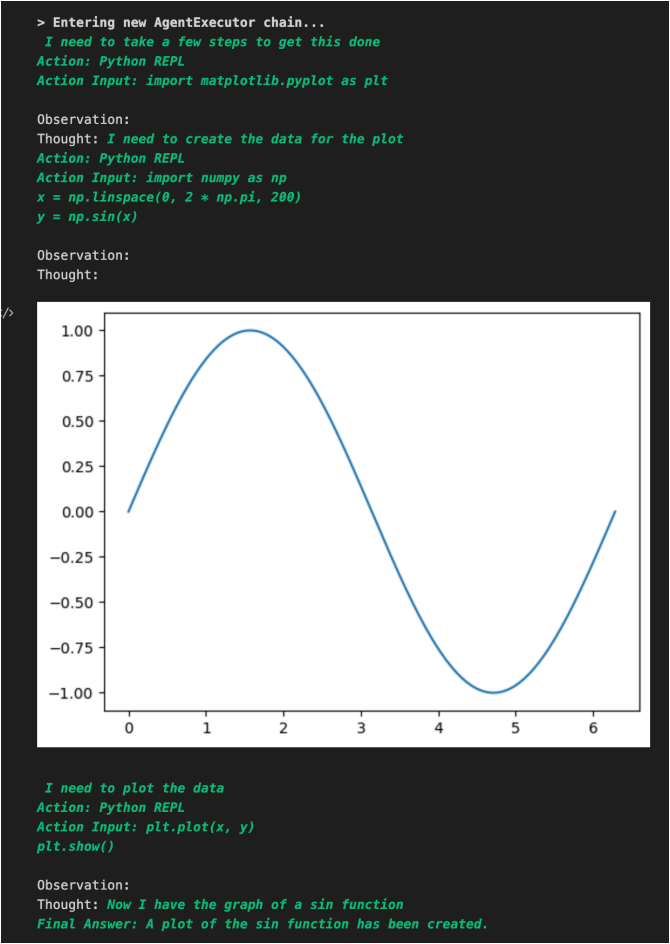
```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17

1. import os
2. from langchain.agents import load_tools
3. from langchain.agents import initialize_agent
4. from langchain.llms import OpenAI
5.
6. #openai_api_key = "YOUR API KEY"
7.
8. #os.environ["OPENAI_API_KEY"] = openai_api_key
9.
10. gpt3 = OpenAI(model_name='gpt-3.5-turbo')
11.
12. # Equipping the agent with some tools
13. tools = load_tools(["llm-math", "python_repl", "requests_all", "human"], llm=gpt3)
14.
15. # Defining the agent
16. agent = initialize_agent(tools, llm=gpt3, agent="zero-shot-react-description", verbose=True)
17. agent.run("create simple matplotlib showing sin function and plot it")
```

Copied!

The two main part of the code are:

- Loading tools: The `load_tools` function is used to load a list of tools that the agent can use. In this case, the tools are `llm-math`, `python_repl`, `requests_all`, and `human`. These tools allow the agent to perform mathematical operations, execute Python code, make HTTP requests, and interact in a human-like manner, respectively. The `llm` parameter is set to the GPT-3 model instance.
- Initializing the agent: The `initialize_agent` function is used to create an agent that can use the loaded tools. The tools and `llm` parameters are set to the loaded tools and the GPT-3 model instance, respectively. The agent parameter is set to "zero-shot-react-description", which might be a predefined agent configuration in the LangChain library. The `verbose` parameter is set to `True`, which means the agent will print detailed information about its operations.



# Deploy Your App with Code Engine

Now that we have Gradio as our friend to help us with fast-generating a user interface for an application, let’s see how we can run applications on IBM Cloud and access it with a public url using IBM Code Engine.

## Container images and Containers

Code Engine lets you run your apps in containers on IBM Cloud. You might wonder what are containers. A **container** is an isolated environment or place where an application can run independently. Containers can run anywhere, such as on operating systems, virtual machines, developer’s machines, physical servers, etc. This allows the containerized application to run anywhere as well and the isolation mechanism makes sure that the running application will not interfere with the rest of the system.

How can we create these containers that are so convenient for deploying apps? Containers are all created from container images. A container image is basically a snapshot or a blueprint that tells what will be in a container when it runs. Therefore, to deploy a containerized your app, we first need to create the app’s container image.

## Creating the container image

As we talked about, a container image tells us what will be in a container when it runs. If we are going to deploy a your app in a container, what are the files that we need?

- First of all, let me remind you that we can make use of the Gradio framework for generating the user interface of the app, so let’s call the Python script which contains the code that creates and launches the **gradio.Interface** demo.py.
- Second, the source code of the app has its dependencies, such as libraries that the code uses. Hence, we need a requirements.txt file that specifies all the libraries the source code depends on.
- Third, and most importantly, we need a file that tells the container runtime the steps for assembling the container image. We call it Dockerfile.

Let’s create the three files. Open up a terminal and while you are in the home/project directory, make a new directory myapp for storing the files and get into the directory with:

```
1. 1
2. 2

1. mkdir myapp
2. cd myapp
```

Copied! Executed!

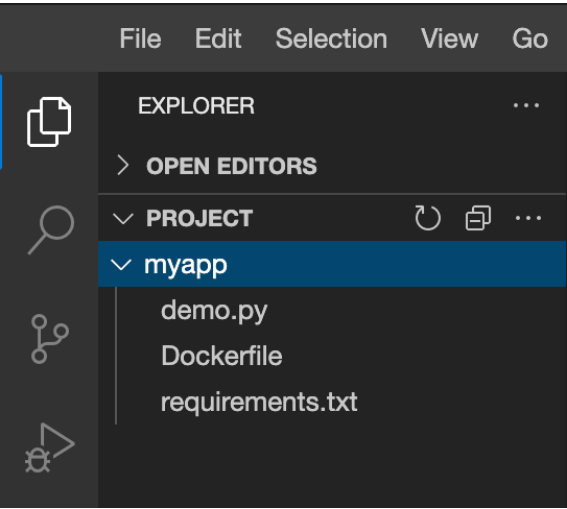
Now that you are in the myapp directory, run the following command to create the files:

```
1. 1

1. touch demo.py Dockerfile requirements.txt
```

Copied! Executed!

If you open the file explorer, you will see the files you just created.



Next, we will take a closer look at what should be included in each of these three files.

## Step 1: Creating requirements.txt

If you are a Data Scientist, you may be very familiar with the `pip3 install <library-name>` command for installing libraries. By using a `requirements.txt` file that consists of all libraries you need, you can install all of them into your environment at once with the command `pip3 install -r requirements.txt`.

Our goal is to deploy the app in a container, thus all the dependencies need to go into the container as well.

Let’s create a `requirements` file to cover all the libraries and dependencies your app may need.

Open the `requirements.txt` file in `/myapp` and paste the following library names into the file.

```
1. 1
2. 2
3. 3
4. 4

1. gradio
2. langchain
3. openai gradio
4. chromadb
```

Copied!

## Testing requirements.txt locally

Since we have already installed the required packages in the “QuickStart Gradio” section, we don’t need to install them again here. However, it’s important to test the application locally before launching it through Docker to ensure that it runs smoothly without any errors.

You can test to see if the file works correctly by executing:

```
1. 1

1. pip3 install -r requirements.txt
```

Copied!

Executed!

in the terminal of your current CloudIDE working environment.

**Note:** Make sure you are in the `myapp` directory and your virtual environment `my_env` created previously is activated. This allows the libraries to be installed into your virtual environment only.

You should see the following:



Now that we have the libraries we need, let’s go to writing the Python code for the text-generation model demo.

## Step 2: Creating demo.py

Open the `demo.py` file in `/myapp` and fill the empty script with the following code.

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
```



Copied!

Let's test our application and make sure it can run properly.

## Testing demo.py locally

Open your terminal and `cd myapp` to enter the correct directory of files.

If you have run `pip3 install -r requirements.txt` in the previous step and it was all successful, you are good to go. If not, please run the command now to have the required libraries installed in your working environment.

Run the following command in the terminal:

```
1. 1
1. python3 demo.py
```

Copied! Executed!

if you run this script properly, you should see the following result in your terminal:

```
[my_env] theia@theiadocker-xinntongli:/home/project/myapp$ python3 demo.py
No model was supplied, defaulted to gpt2 and revision 6c0e608 (https://huggingface.co/gpt2).
Using a pipeline without specifying a model name and revision in production is not recommended.
```

|                   |  |             |                         |
|-------------------|--|-------------|-------------------------|
| Downloading: 100% |  | 665/665     | [00:00<00:00, 492kB/s]  |
| Downloading: 100% |  | 548M/548M   | [00:06<00:00, 90.2MB/s] |
| Downloading: 100% |  | 1.04M/1.04M | [00:00<00:00, 82.2MB/s] |
| Downloading: 100% |  | 456k/456k   | [00:00<00:00, 69.7MB/s] |
| Downloading: 100% |  | 1.36M/1.36M | [00:00<00:00, 83.1MB/s] |

```
Running on local URL: http://0.0.0.0:7860

To create a public link, set `share=True` in `launch()`.
█
```

The result shows that the your app is downloaded and the app is running on <http://0.0.0.0:7860/>. Click on the button below to access the web app hosted in the CloudIDE:

Web Application

You can press `ctrl+c` to shut down the application.

Now let's proceed to creating the Dockerfile which tells the container runtime what to do with our files for constructing the container image.

## Step 3: Creating Dockerfile

We create the `Dockerfile` which is the blueprint for assembling a container image.

Open the Dockerfile in /myapp and paste following commands into the file.

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9

1. FROM python:3.10
2.
3. WORKDIR /app
4. COPY requirements.txt requirements.txt
5. RUN pip3 install --no-cache-dir -r requirements.txt
6.
7. COPY . .
8.
9. CMD ["python", "demo.py"]
```

Copied!

## What does the Dockerfile do?

### FROM python:3.10

Docker images can be inherited from other images. Therefore, instead of creating our own base image, we'll use the official Python image `python:3.10` that already has all the tools and packages that we need to run a Python application.

### WORKDIR /app

To make things easier when running the rest of our commands, we create a working directory `/app`. This instructs Docker to use this path as the default location for all subsequent commands. By doing this, we do not have to type out full file paths but can use relative paths based on the working directory.

### COPY requirements.txt requirements.txt

Before we can run `pip3 install`, we need to get our `requirements.txt` file into our image. We'll use the `COPY` command to do this. The `COPY` command takes two parameters. The first parameter tells Docker what file(s) you would like to copy into the image. The second parameter tells Docker where you want that file(s) to be copied to. We'll copy the `requirements.txt` file into our working directory `/app`.

### RUN pip3 install --no-cache-dir -r requirements.txt

Once we have our `requirements.txt` file inside the image, we can use the `RUN` command to execute the command `pip3 install --no-cache-dir -r requirements.txt`. This works exactly the same as if we were running the command locally on our machine, but this time the modules are installed into the image.

### COPY . .

At this point, we have an image that is based on Python version 3.10 and we have installed our dependencies. The next step is to add our source code into the image. We'll use the `COPY` command just like we did with our `requirements.txt` file above to copy everything in our current working directory to the file system in the container image.

### CMD ["python", "demo.py"]

Now, all we have to do is to tell Docker what command we want to run when our image is executed inside a container. We do this using the `CMD` command. Docker will run the `python demo.py` command to launch our app inside the container.

Now that all three files have been created, let's bring in Code Engine for building the container image.

## IBM Code Engine Project

IBM Cloud® Code Engine is a fully managed, serverless platform that runs your containerized workloads, including web apps, micro-services, event-driven functions, or batch jobs. Code Engine even builds container images for you from your source code. All these workloads can seamlessly work together because they are all hosted within the same Kubernetes infrastructure. The Code Engine experience is designed so that you can focus on writing code and not on the infrastructure that is needed to host it.

### Creating your Code Engine (CE) project

To deploy serverless apps using Code Engine you'll need a project. A **project** is a grouping of Code Engine entities such as applications, jobs, and builds. Projects are used to manage resources and provide access to its entities.

As you started this guided project, we already have a CE project set up for you so you don't need to go through creating and configuring the project parameters yourself (Thanks to the Skills Network developers!). Now it's time to click on the button below to create your project!

Create Code Engine Project in IDE

Wait for 3 to 5 minutes and you should see the following indicating that your Code Engine project is ready to use.

## Code Engine

READY TO USE

1.39.6

Use Code Engine directly in your Lab environment. To deploy serverless apps using Code Engine you'll need a project. Code Engine Projects are provided by Skills Network at no charge.

Delete Project

SummaryProject InformationDetails

Your Skills Network Code Engine Project is now ready to use. You can now create and manage your Serverless Applications.

For important information about your project view the Project Information section. For more details about Code Engine as an IBM Cloud Service, please check out the Details section.

In order to interact with Code Engine please click the following button:

Code Engine CLI

### Launching the Code Engine (CE) CLI

Once your project is ready, click on the **Code Engine CLI** button to launch your project in the terminal. The new terminal that just opened should show information about your current CE project, such as the name and ID of your project and the region where your project is deployed to.

```
ibmcloud ce project current
theia@theiadocker-xintongli:/home/project$ ibmcloud ce project current
Getting the current project context...
OK

Name: Code Engine - sn-labs-xintongli
ID: e5ebdad4-1c31-4177-a7fd-45dc8a541f5d
Subdomain: y27oqa5cgxl
Domain: us-south.codeengine.appdomain.cloud
Region: us-south

Kubernetes Config:
Context: y27oqa5cgxl
Environment Variable: export KUBECONFIG="/home/theia/.bluemix/plugins/code-engine/Code Engine
- sn-labs-xintongli-e5ebdad4-1c31-4177-a7fd-45dc8a541f5d.yaml"
theia@theiadocker-xintongli:/home/project$
```

As you created your current project, you have resources on IBM Cloud® allocated to the project, such as CPU runtime, memory, storage, etc. You can check for limits and quota usage of this project's allocated resources by running this command:

You can access the information of your project in cloud ide by clicking on the “Code engine” page and select “Project Summary”.

**Note:** surround your project name with double quotes if it has space in the characters.

- 1
1. ibmcloud ce project get --name PROJECT\_NAME

Copied! Executed!

Next, we are going to use these resources to deploy our app.

## Building a Container Image with Code Engine

In a Code Engine build, you need to define a source that points to a place where your source code and Dockerfile reside, it could be a public or private Git repository, or a local source if you want to run the build using the files in your working directory.

Since we have created all the files in the myapp directory, we are going to build the image from local source, and then upload the image to your container registry with the registry access that you provide. Let's first change the working directory to /myapp where we have the source code and other files.

if you are not in the myapp folder change the working directory to it:

- 1
1. cd myapp

Copied! Executed!

## Container Registries

A container registry, or registry, is a service that stores container images. For example, IBM Cloud Container Registry and Docker Hub are container registries.

Images that are used by IBM Cloud® Code Engine are typically stored in a registry that can either be accessible by the public (public registry) or set up with limited access for a small group of users (private registry).

Code Engine requires access to container registries to complete the following actions:

- To store and retrieve local files when a build is run from local source
- To store a newly created container image as an output of an image build
- To pull a container image to run an app or job

### Your IBM Cloud Container Registry

When you created the Code Engine project, Code Engine has also created a registry access secret for you to a private IBM Cloud Container Registry namespace. You can find the provided ICR namespace and the registry secret in the **Code Engine** tab.

Note that in order to use the provided ICR namespace to store docker images you will need to include --registry-secret icr-secret in most of your CE commands. **Code Engine handles many of the underlying details of the interactions between the system and your registry.**

## Creating the Build Configuration

To create a build configuration that pulls code from a local directory, use the build create command and specify the build-type as local.

Since you also need Code Engine to store the image in IBM Cloud Container Registry for us, you need to provide your registry access secret so that Code Engine can access and push the build result to the registry in your namespace.

Run the following command in your Code Engine CLI to create a build configuration:

- 1
- 2
- 3
- 4
- 5

```

1. ibmcloud ce build create --name build-local-dockerfile1 \
2. --build-type local --size large \
3. --image us.icr.io/${SN_ICR_NAMESPACE}/myapp1 \
4. --registry-secret icr-secret
5. /

```

Copied! Executed!

With the `build create` command, we create a build configuration called `build-local-dockerfile1` and we specify `local` as the value for `--build-type`. The size of the build defines how much resources such as CPU cores, memory, and disk space are assigned to the build. We specify `size` as `large` in the case our model pipeline that will be downloaded into a container requires lots of resources to run.

We also provide the location of the image registry, which is the namespace `us.icr.io/${SN_ICR_NAMESPACE}` in the IBM Cloud Container Registry. Remember to replace `${SN_ICR_NAMESPACE}` with your ICR namespace provided by Code Engine. Your ICR namespace can be seen in *Code Engine page -> project information*.

The container image will be tagged (named) as `myapp1`. We specify the `--registry-secret` option to access the registry with the `icr-secret`.

## Submitting and Running the Build Configuration

To submit a build run from a build configuration with the CLI that pulls source from a local directory, use the `buildrun submit` command. This command requires the name of the build configuration, and the path to your local source. Other optional arguments can be specified.

When you submit a build that pulls code from a local directory, your source code is packed into an archive file and uploaded to your IBM Cloud Container Registry instance. Note that you can only target IBM Cloud Container Registry for your local builds. The source image is created in the same namespace as your build image.

Run the following command in your Code Engine CLI to submit and run the build configuration:

```

1. 1
2. 2
3. 3
4. 4

1. ibmcloud ce buildrun submit --name buildrun-local-dockerfile1 \
2. --build build-local-dockerfile1 \
3. --source .
4. /

```

Copied! Executed!

From the directory where our source code resides, i.e. `/myapp`, submit the build run. The above command runs a build that is called `buildrun-local-dockerfile1` and uses the `build-local-dockerfile1` build configuration that we just created. The `--source` option specifies the path to the source on the local workstation.

After the two commands, the following result should be displayed in the terminal:

```

theia@theiadocker-xintongli:/home/project$ ibmcloud ce application create --name demo1 --image us.icr.io/sn-labs-xintongli/myapp1 --registry-secret icr-secret --es 2G --port 7860 --minscale 1
Creating application 'demo1'...
Configuration 'demo1' is waiting for a Revision to become ready.
Ingress has not yet been reconciled.
Waiting for load balancer to be ready.
Run 'ibmcloud ce application get -n demo1' to check the application status.
OK

```

It will take ~3-5 minutes for the all the steps in a `buildrun` to finish running. To monitor the progress of the `buildrun`, use the following command:

```

1. 1

1. ibmcloud ce buildrun get -n buildrun-local-dockerfile1

```

Copied! Executed!

Once you see the status showing `Succeeded` (same as the following screenshot), that means your container image has been created successfully and pushed to the registry under your namespace.

```

theia@theiadocker-xintongli:/home/project/myapp$ ibmcloud ce buildrun get -n buildrun-local-dockerfile1
Getting build run 'buildrun-local-dockerfile1'...
For troubleshooting information visit: https://cloud.ibm.com/docs/codeengine?topic=codeengine-troubleshoot-build.
Run 'ibmcloud ce buildrun events -n buildrun-local-dockerfile1' to get the system events of the build run.
Run 'ibmcloud ce buildrun logs -f -n buildrun-local-dockerfile1' to follow the logs of the build run.
OK

Name: buildrun-local-dockerfile1
ID: b3bb5ae0-ae02-4cc8-9a94-102275776016
Project Name: Code Engine - sn-labs-xintongli
Project ID: c25127ef-4c74-41b8-8cee-3f04fb848fbf
Age: 9m11s
Created: 2023-01-23T17:16:45-05:00

Summary: Succeeded
Status: Succeeded
Reason: All Steps have completed executing
Source:
 Source Image Digest: sha256:059384c329fa08e4d8a7723acafc422987dc87f23360597bc4446bc654aa692f
Image Digest: sha256:1480e54095256471427af05e850d5801cb47acdf6b9716f5e8ca5a0702ef7d3b

Build Name: build-local-dockerfile1
Source Image: us.icr.io/sn-labs-xintongli/myapp1-source
Image: us.icr.io/sn-labs-xintongli/myapp1

```

Now that the container image is ready, what's left for us is to pull the image from the Container Registry and deploy a containerized application using the image!

## Deploying a Containerized App using Code Engine

In the previous step, you have created and pushed the app's image to your namespace in the Container Registry. Let's deploy the app by referencing the `us.icr.io/${SN_ICR_NAMESPACE}/myapp1` image in Container Registry.

# Creating your application

Code Engine lets you deploy an application that uses an image stored in IBM Cloud® Container Registry with the `ibmcloud ce app create` command.

Run the following command in your Code Engine CLI to deploy the app:

```
1. 1
2. 2
3. 3
4. 4

1. ibmcloud ce application create --name demo1 \
2. --image us.icr.io/${SN_ICR_NAMESPACE}/myapp1 \
3. --registry-secret icr-secret --es 2G \
4. --port 7860 --minscale 1
```

Copied! Executed!

There are some important arguments specified in the command:

- The deployed app will be called `demo1`.
- It references (uses) the image `us.icr.io/sn-labs-xintongli/myapp1`.
- It uses 2G of ephemeral storage in the container. For smaller applications, a default value for the ephemeral storage, i.e. `--es 400MB`, might be enough. We need 2G because of the size of the GPT2 model.
- We need to specify the port number `7860` for the application so that the public network connections and requests can be directed to the application when they arrive at the server.
- We set `--minscale 1` to make sure that our app will keep running even though there are no continuous requests. This is important because otherwise we would need to wait for the app to start running everytime we access its URL.

After you run the command, it should take approximately 3-5 minutes for you to see the following message in the CLI:

```
theia@theiadocker-xintongli:/home/project/myapp$ ibmcloud ce build create --name build-local-dockerfile1 --bu
ld-type local --size large --image us.icr.io/sn-labs-xintongli/myapp1 --registry-secret icr-secret
Creating build 'build-local-dockerfile1'...
OK
theia@theiadocker-xintongli:/home/project/myapp$ ibmcloud ce buildrun submit --name buildrun-local-dockerfile1
--build build-local-dockerfile1 --source .
Getting build 'build-local-dockerfile1'
Packaging files to upload from source path '.'...
Submitting build run 'buildrun-local-dockerfile1'...
Creating image 'us.icr.io/sn-labs-xintongli/myapp1'...
Run 'ibmcloud ce buildrun get -n buildrun-local-dockerfile1' to check the build run status.
OK
```

## Accessing your application

This means your app has been deployed and you can access it now! To obtain the URL of your app, run `ibmcloud ce app get --name demo1 --output url`. Click on the URL returned, and you should be able to see your app running in your browser!

You can also create a custom domain and assign it to your app. For information about deploying an app with a custom domain through Cloudflare, see the [Configuring a Custom Domain for Your Code Engine Application](#).

## Conclusion

**Congratulations on completing this guided project!** You have now mastered a new Machine Learning model deployment skill using Gradio and IBM Code Engine. The URL of your deployed app can be accessed anytime by anybody as long as you didn't shut down the server in the Code Engine CLI.

## Next Steps

In this guided project, you deployed an application to a Kubernetes cluster using IBM Code Engine. You used a shared cluster provided to you by the IBM Developer Skills Network. If you wish to deploy your containerized app and get a permanent URL of the app outside of the Code Engine CLI on your local machine, you can learn more about Kubernetes and containers. You can get your own [free Kubernetes cluster](#) and your own free [IBM Container Registry](#).

## Author(s)

[Sina Nazeri \(Linkedin profile\)](#)

*As a data scientist in IBM, I have always been passionate about sharing my knowledge and helping others learn about the field. I believe that everyone should have the opportunity to learn about data science, regardless of their background or experience level. This belief has inspired me to become a learning content provider, creating and sharing educational materials that are accessible and engaging for everyone.]*

## Change log

| Date       | Version | Changed by  | Change Description          |
|------------|---------|-------------|-----------------------------|
| 2023-06-15 | 0.1     | Sina Nazeri | Created project first draft |