LangChain Framework and Components

Generative AI and Prompt Engg.

Ram N Sangwan

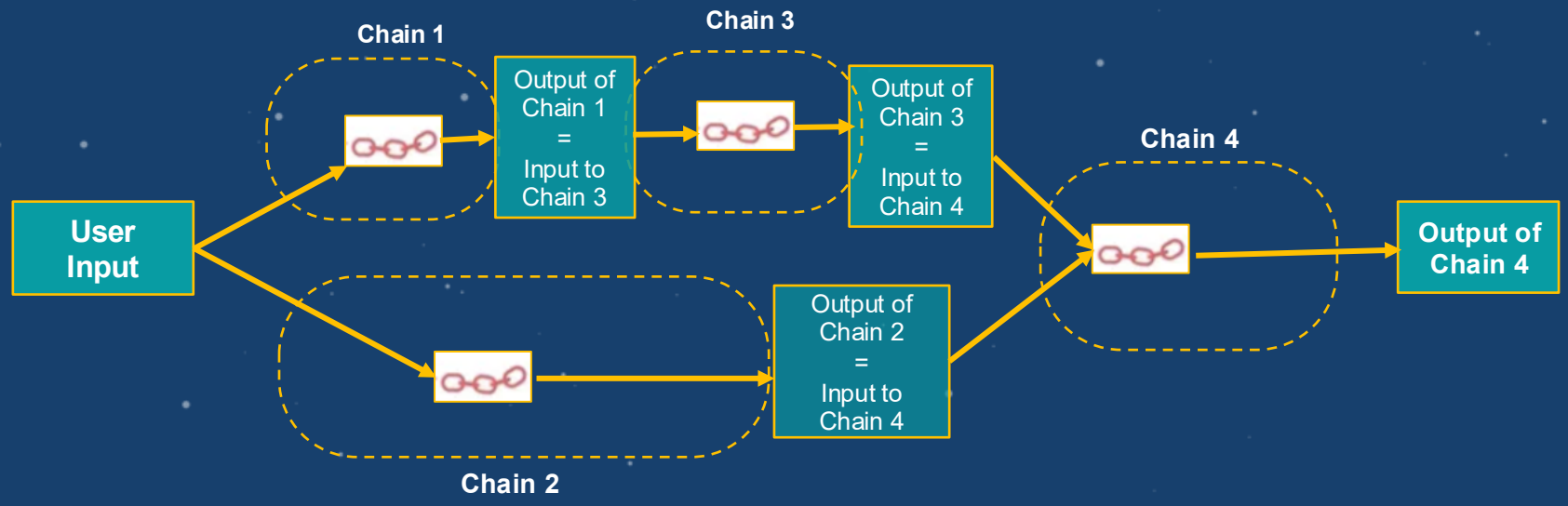
- Introducing LangChain
- Components of LangChain
- LangChain Pipeline Architecture
- LangChain SQL Connector

What is LangChain?

- A robust library designed to streamline interaction with several LLM providers like Cohere, OpenAI, Ollama, Huggingface etc.
- Chains are logical links between one or more LLMs.

What are LangChain Chains?

- Chains are the backbone of LangChain's functionality.
- Chains can range from simple to complex, contingent on the necessities and the LLMs involved.



Basic Chains

- A basic chain is the simplest form of a chain.
- It involves a single LLM receiving an input prompt and using that prompt for text generation.

```
import os
from langchain.llms import Cohere
from langchain_core.prompts import PromptTemplate

prompt = PromptTemplate.from_template("Tell me a joke about {topic}")

chain = prompt | model

model = Cohere(model="command", max_tokens=256, temperature=0.75)

chain.invoke({"topic": "bears"})
```

Using Chains

```python
from langchain_community.chat_models.oci_generative_ai import ChatOCIGenAI
from langchain.prompts import PromptTemplate
from langchain.chains import SimpleSequentialChain

llm = ChatOCIGenAI(
    model_id="cohere.command-r-plus",
    service_endpoint=service_endpoint,
    compartment_id=compartment_ocid,
    model_kwargs={"temperature": 0, "max_tokens": 500},
)
```

- **Chain 1.**

```python
template = """Your job is to come up with a classic dish from the area that the users suggests. Respond with Dish name and One line description of it.
% USER LOCATION
{user_location}

YOUR RESPONSE:
"""
prompt = PromptTemplate(input_variables=["user_location"], template=template)

location_chain = prompt | llm
```

- **Chain 2**

```python
template = """Given a meal, give a short and simple recipe on how to make that dish at home.
% MEAL
{user_meal}

YOUR RESPONSE:
"""
prompt = PromptTemplate(input_variables=["user_meal"], template=template)

# Holds my 'meal' chain
meal_chain = prompt | llm
```

Using Chains

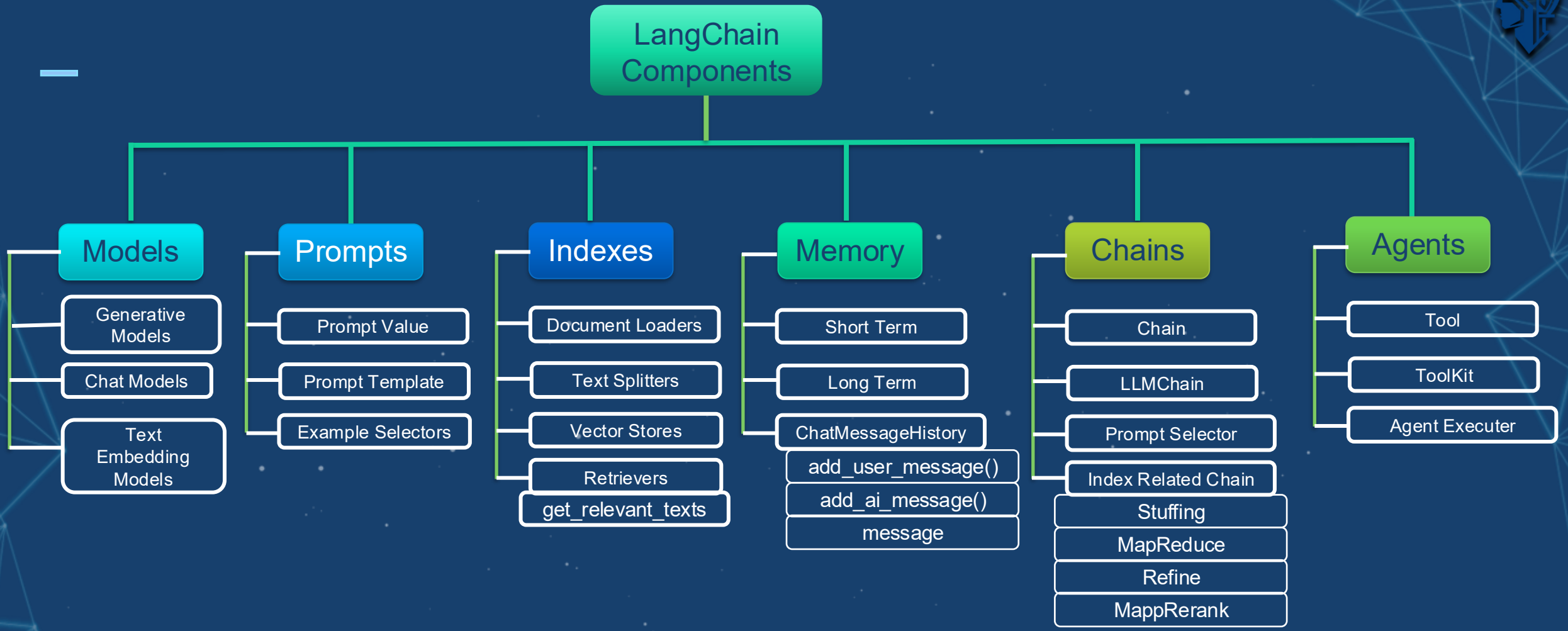
```
overall_chain = location_chain | meal_chain
```

- The output from chain 1 is passed as input to chain 2.
- Now, let's run this chain:

```
review = overall_chain.invoke("Hyderabd")
print(review.content)
```

```
> Entering new SimpleSequentialChain chain...
One classic dish from Rome is spaghetti alla carbonara. It is a simple pasta dish made with spaghetti, eggs, cheese, and bacon or pancet
ta. The dish is believed to have originated in Rome during World War II, when American soldiers introduced the locals to their rations o
f bacon and eggs. The combination of rich, creamy egg and cheese sauce with the salty, crispy bacon has made spaghetti alla carbonara a
beloved dish in Rome and around the world.
Spaghetti alla Carbonara Recipe:
Ingredients:
- 12 ounces spaghetti
- 4 eggs
- 1 cup grated pecorino romano cheese
- 1 cup grated parmesan cheese
- 4-6 slices of bacon or pancetta, diced
- 2 cloves of garlic, minced
- Salt and pepper
- Fresh parsley, chopped (optional)

Instructions:
1. Bring a large pot of salted water to a boil and cook spaghetti according to package instructions until al dente.
2. In a separate pan, cook diced bacon or pancetta over medium heat until crispy. Remove from pan and set aside.
3. In the same pan, add minced garlic and cook until fragrant.
4. In a bowl, whisk together eggs, pecorino romano cheese, and parmesan cheese.
5. Drain cooked spaghetti and add it to the pan with the garlic. Toss well to coat the spaghetti with the garlic.
6. Remove the spaghetti from heat and let it cool for a minute.
7. Slowly pour in the egg and cheese mixture, stirring constantly to prevent the eggs from scrambling.
8. Add in the cooked bacon or pancetta and mix well.
9. Season with salt and pepper to taste.


> Finished chain.
```

Components of LangChain


LangChain Components
Models
Generative Models
Chat Models
Text Embedding Models
Prompts
Prompt Value
Prompt Template
Example Selectors
Indexes
Document Loaders
Text Splitters
Vector Stores
Retrievers
get_relevant_texts
Memory
Short Term
Long Term
ChatMessageHistory
add_user_message()
add_ai_message()
message
Chains
Chain
LLMChain
Prompt Selector
Index Related Chain
Stuffing
MapReduce
Refine
MappRerank
Agents
Tool
ToolKit
Agent Executer

Chat Messages

Specified with a message type (System, Human, AI)

System

Helpful background context that tell the AI what to do

Human

Messages that are intended to represent the user

AI

Messages that show what the AI responded with

```
[92]: chat(
          [
              SystemMessage(content="You are a nice AI bot that helps a user figure out what to eat in one short sentence"),
              HumanMessage(content="I like tomatoes, what should I eat?")
          ]
      )
```

Chat Messages

You can also pass more chat history w/ responses from the AI

```
[94]: chat(
          [
              SystemMessage(content="You are a nice AI bot that helps a user figure out where to travel in one short sentence"),
              HumanMessage(content="I like the beaches where should I go?"),
              AIMessage(content="You should go to Nice, France"),
              HumanMessage(content="What else should I do when I'm there?")
          ]
      )
```

You can also exclude the system message if you want

```
[95]: chat(
          [
              HumanMessage(content="What day comes after Thursday?")
          ]
      )
```

Documents

An object that holds a piece of text and metadata (more information about that text)

```
[96]: from langchain.schema import Document

[97]: Document(page_content="This is my document. It is full of text that I've gathered from other places",
              metadata={
                  'my_document_id' : 234234,
                  'my_document_source' : "The LangChain Papers",
                  'my_document_create_time' : 1680013019
              })

[97]: Document(page_content="This is my document. It is full of text that I've gathered from other places", metadata={'my_document_id': 23423
      4, 'my_document_source': 'The LangChain Papers', 'my_document_create_time': 1680013019})
```

But you don't have to include metadata if you don't want to

```
[98]: Document(page_content="This is my document. It is full of text that I've gathered from other places")

[98]: Document(page_content="This is my document. It is full of text that I've gathered from other places")
```

Prompt Templates and Vector Stores

Prompt Templates

- Reusable predefined prompts across chains.
  - *Can become dynamic and adaptable by inserting specific "values."*
  - *E.g, a prompt asking for a user's name could be personalized by a specific value.*

Vector Stores

- To store and search information via embeddings, essentially analyzing numerical representations of document meanings.
  - *Serves as a storage facility for these embeddings, allowing efficient search based on semantic similarity.*

Prompt Templates and Vector Stores

Indexes and Retrievers

- ***Indexes*** *act as databases for knowledge base.*
- ***Retrievers*** *swiftly search this index for specific information.*

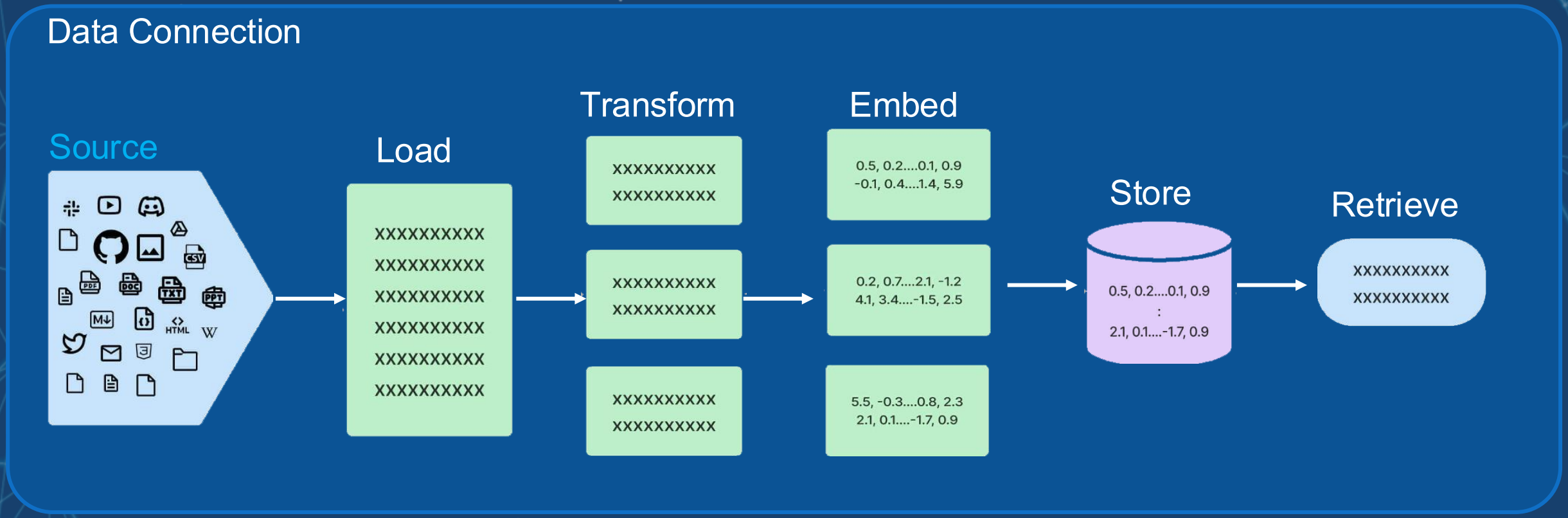
Output Parsers

- *Eliminate undesired content, tailor the output format, or supplement data to the response.*
- *Thus, help extract structured results, like JSON objects, from the LLM's responses.*

Example Selectors

- *Identify appropriate instances from the response.*
- *Can be adjusted to favour certain examples or filter out unrelated ones, providing a tailored AI response.*

Retrieval


Data Connection
Source
Load
Transform
Embed
Store
Retrieve
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
xxxxxxxxxx
0.5, 0.2....0.1, 0.9
-0.1, 0.4....1.4, 5.9
0.2, 0.7....2.1, -1.2
4.1, 3.4....-1.5, 2.5
5.5, -0.3....0.8, 2.3
2.1, 0.1....-1.7, 0.9
0.5, 0.2....0.1, 0.9
:
2.1, 0.1....-1.7, 0.9
xxxxxxxxxx
xxxxxxxxxx

Document Loaders

Document loaders load documents from many different sources.

- LangChain provides over 100 document loaders as well as integrations with other major providers.
- LangChain provides integrations to load all types of documents (HTML, PDF, code) from all locations (private S3 buckets, public websites).

Text splitters - Chunking Data

Chunking Data

- Once you've loaded documents, you'll often want to transform them to better suit your application.
- The simplest example is you may want to split a long document into smaller chunks that can fit into your model's context window.
- When you want to deal with long pieces of text, it is necessary to split up that text into chunks.
- Ideally, you want to keep the semantically related pieces of text together.
- What "semantically related" means could depend on the type of text.

Embedding Models

Another key part of retrieval is creating embeddings for documents.

- Embeddings capture the semantic meaning of the text, allowing you to quickly and efficiently find other pieces of a text that are similar.
- LangChain provides integrations with over 25 embedding providers and methods, from open-source to proprietary API.
- LangChain provides a standard interface, allowing you to easily swap between models.

Working with text splitters

[50]:
```
from langchain.document_loaders import HNLoader
```

[51]:
```
loader = HNLoader("https://news.ycombinator.com/item?id=34422627")
```

[52]:
```
data = loader.load()
```

[53]:
```
print (f"Found {len(data)} comments")
print (f"Here's a sample:\n\n{''.join([x.page_content[:150] for x in data[:2]])}")
```

```
Found 76 comments
Here's a sample:

Ozzie_osman on Jan 18, 2023
             | next [–]

LangChain is awesome. For people not sure what it's doing, large language models (LLMs) are veOzzie_osman on Jan 18, 2023
             | parent | next [–]

Also, another library to check out is GPT Index (https://github.com/jerryjliu/gpt_ind
```

Books from Gutenberg Project

[54]:
```
from langchain.document_loaders import GutenbergLoader

loader = GutenbergLoader("https://www.gutenberg.org/cache/epub/2148/pg2148.txt")

data = loader.load()
```

[55]:
```
print(data[0].page_content[1855:1984])
```

```
      At Paris, just after dark one gusty evening in the autumn of 18-,


      I was enjoying the twofold luxury of meditation
```

URLs and webpages

Let's try it out with Paul Graham's website

[56]:
```
from langchain.document_loaders import UnstructuredURLLoader

urls = [
    "http://www.paulgraham.com/",
]

loader = UnstructuredURLLoader(urls=urls)

data = loader.load()

data[0].page_content
```

[56]:
```
'New: Superlinear Returns | How to Do Great Work Want to start a startup? Get funded by Y Combinator . © mmxxiii pg'
```

How it Works

At a high level, text splitters work as following:

1. Split the text up into small, semantically meaningful chunks (often sentences).
2. Start combining these small chunks into a larger chunk until you reach a certain size (measured by some function).
3. Once you reach that size, make that chunk its own piece of text and then start creating a new chunk of text with some overlap (to keep context between chunks).

That means there are two different axes along which you can customize your text splitter:

1. How the text is split
2. How the chunk size is measured

Types of Text Splitters

Name	Splits On	Adds Metadata	Description
Recursive	A list of user defined characters		Recursively splits text. Splitting text recursively serves the purpose of trying to keep related pieces of text next to each other.
HTML	HTML specific characters	✅	Splits text based on HTML-specific characters. This adds in relevant information about where that chunk came from (based on the HTML)
Markdown	Markdown specific characters	✅	Splits text based on Markdown-specific characters. This adds in relevant information about where that chunk came from (based on the Markdown)
Code	Code (Python, JS) specific characters		Splits text based on characters specific to coding languages. 15 different languages are available to choose from.
Token	Tokens		Splits text on tokens.
Character	A user defined character		Splits text based on a user defined character.

Text Splitters

Often times your document is too long (like a book) for your LLM. You need to split it up into chunks. Text splitters help with this.

[57]:
```
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

[58]:
```
# This is a long document we can split up.
with open('data/worked.txt') as f:
    pg_work = f.read()

print (f"You have {len([pg_work])} document")
```

```
You have 1 document
```

[59]:
```
text_splitter = RecursiveCharacterTextSplitter(
    # Set a really small chunk size, just to show.
    chunk_size = 150,
    chunk_overlap  = 20,
)

texts = text_splitter.create_documents([pg_work])
```

[60]:
```
print (f"You have {len(texts)} documents")
```

```
You have 610 documents
```

[61]:
```
print ("Preview:")
print (texts[0].page_content, "\n")
print (texts[1].page_content)
```

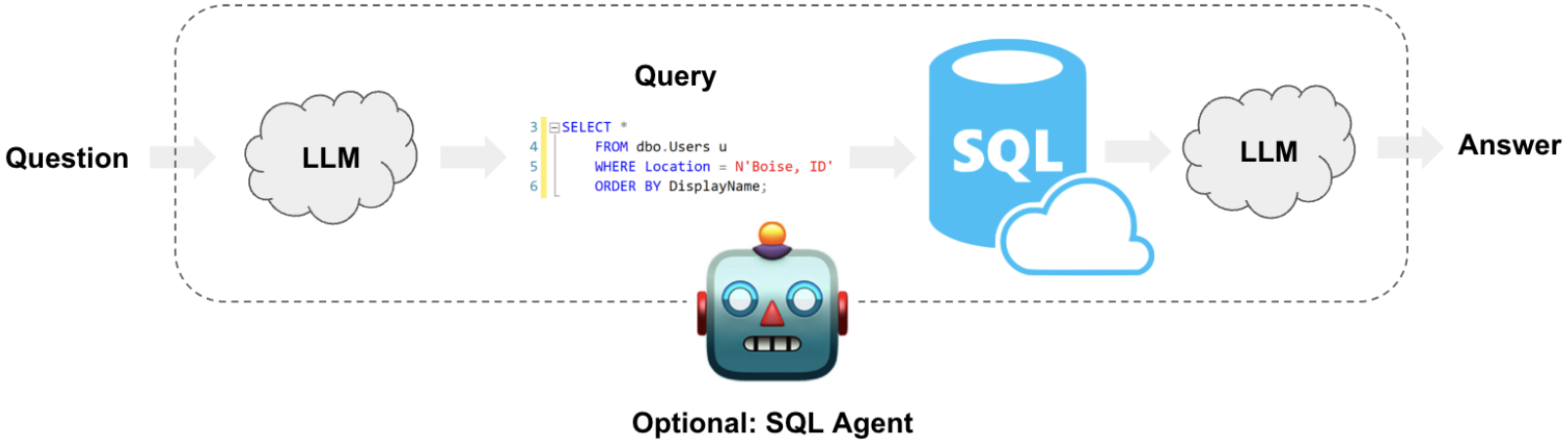
```
Preview:
February 2021Before college the two main things I worked on, outside of school,
were writing and programming. I didn't write essays. I wrote what

beginning writers were supposed to write then, and probably still
are: short stories. My stories were awful. They had hardly any plot,
```

QA System with SQL data

- Enabling a LLM system to query structured data can be qualitatively different from unstructured text data.
- It is common to generate text that can be searched against a vector database, the approach for structured data is often for the LLM to write and execute queries in a DSL, such as SQL.
- The main difference between the two is that our agent can query the database in a loop as many times as it needs to answer the question.

Architecture

- At a high-level, the steps of these systems are:
  - Convert question to DSL query: Model converts user input to a SQL query.
  - Execute SQL query: Execute the query.
  - Answer the question: Model responds to user input using the query results.



Use a SQLite Connection

```
import os
import cohere
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv())
```

- We will use a SQLite connection with Chinook database.:
  - Save this file as *Chinook.sql*
  - Run `sqlite3 Chinook.db`
  - Run `.read Chinook.sql`
  - Test `SELECT * FROM Artist LIMIT 10;`
- Now, Chinook.db is in our directory and we can interface with it using the SQLAlchemy-driven SQLDatabase class:

```
from langchain_cohere import ChatCohere

llm = ChatCohere(model="command")
```

```
from langchain_community.utilities import SQLDatabase

db = SQLDatabase.from_uri("sqlite:///Chinook.db")
print(db.dialect)
print(db.get_usable_table_names())
db.run("SELECT * FROM Artist LIMIT 10;")
```


```
from langchain_community.utilities import SQLDatabase

db = SQLDatabase.from_uri("sqlite:///Chinook.db")
print(db.dialect)
print(db.get_usable_table_names())
db.run("SELECT * FROM Artist LIMIT 10;")
```

Chains

- Chains (compositions of LangChain Runnables) support applications whose steps are predictable.
- We can create a simple chain that takes a question and does the following:
  - convert the question into a SQL query;
  - execute the query;
  - use the result to answer the original question.
- There are scenarios not supported by this arrangement.
- For example, this system will execute a SQL query for any user input-- even "*hello*".
- Importantly, some questions require more than one query to answer.

Convert question to SQL query

- The first step is to take the user input and convert it to a SQL query.
- LangChain comes with a built-in chain for this: create_sql_query_chain.

```
pip install -qU langchain-cohere

from langchain_cohere import ChatCohere

llm = ChatCohere(model="command-r-plus")

from langchain.chains import create_sql_query_chain

chain = create_sql_query_chain(llm, db)
response = chain.invoke({"question": "How many employees are there"})
response

'SELECT COUNT("EmployeeId") AS "TotalEmployees" FROM "Employee"\nLIMIT 1;'
```

- We can also inspect the full prompt like so:

```
chain.get_prompts()[0].pretty_print()
```

Execute SQL query

- This is the most dangerous part of creating a SQL chain.
- Consider carefully if it is OK to run automated queries over your data.
- Minimize the database connection permissions as much as possible.
- Consider adding a human approval step to you chains before query execution.
- We can use the QuerySQLDatabaseTool to easily add query execution to our chain.

```
from langchain_community.tools.sql_database.tool import QuerySQLDataBaseTool

execute_query = QuerySQLDataBaseTool(db=db)
write_query = create_sql_query_chain(llm, db)
chain = write_query | execute_query
chain.invoke({"question": "How many employees are there"})
```

Answer the question

- Now we just need to combine the original question and SQL query result to generate a final answer.
- We can do this by passing question and result to the LLM once more:

```
from operator import itemgetter

from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import PromptTemplate
from langchain_core.runnables import RunnablePassthrough

answer_prompt = PromptTemplate.from_template(
    """Given the following user question, corresponding SQL
query, and SQL result, answer the user question.

Question: {question}
SQL Query: {query}
SQL Result: {result}
Answer: """
)

chain = (
    RunnablePassthrough.assign(query=write_query).assign(
        result=itemgetter("query") | execute_query
    )
    | answer_prompt
    | llm
    | StrOutputParser()
)

chain.invoke({"question": "How many employees are there"})
```

What is happening here?

- After the first RunnablePassthrough.assign, we have a runnable with two elements:
  - `{"question": question, "query": write_query.invoke(question)}`
  - Where write_query will generate a SQL query in service of answering the question.
- After the second `RunnablePassthrough.assign`, we have add a third element `"result"` that contains
  - `execute_query.invoke(query)`, where query was computed in the previous step.
- These three inputs are formatted into the prompt and passed into the LLM.
- The `StrOutputParser()` plucks out the string content of the output message.

Best Practices for Working with Chains

- **Error Handling**: Anticipate and handle potential errors at each link in the chain.
- **State Management**: Especially in conversational systems, manage the state effectively across turns.
- **Logging and Monitoring**: Keep detailed logs for each step to monitor performance and debug issues.
- **Scalability**: Design each component to handle increased loads, possibly by decoupling and allowing parallel processing.
- **Security**: Ensure data security, especially when transferring data between different components or APIs.
- **Optimization**: Continually test and optimize each part of the chain.

– Thank You