# LangChan LLM Ecosystem

Ram N Sangwan

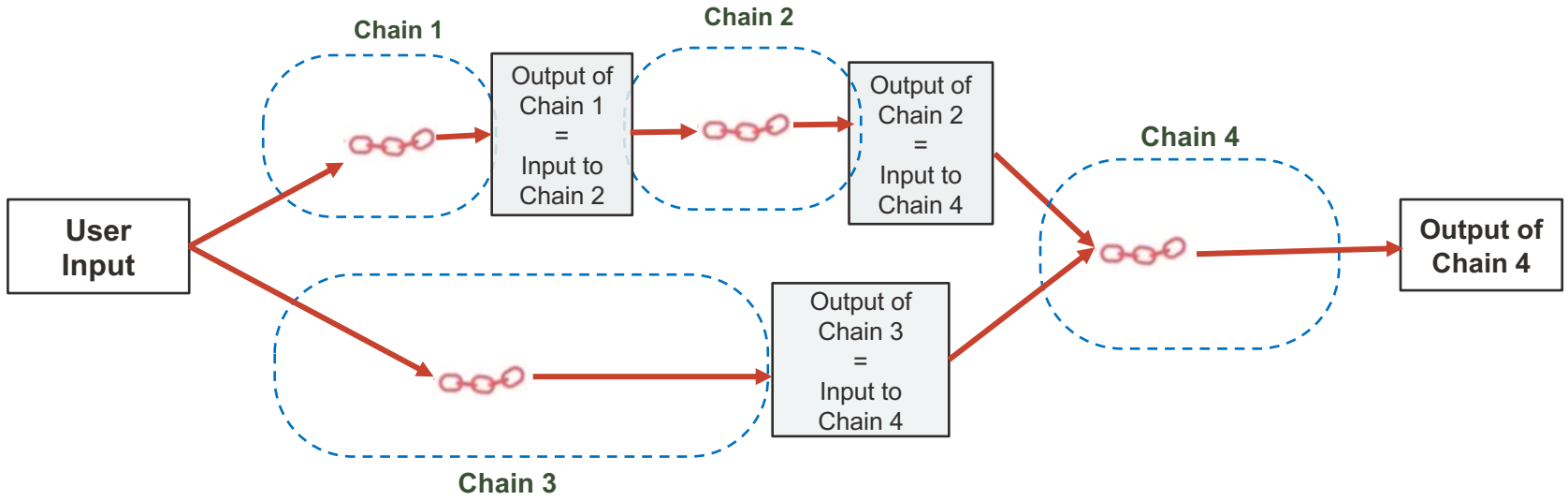
- LangChain Ecosystem
- Langchain Concepts
- Using Multiple LLMs (Chains)
- Working with Chains

# What is LangChain?

- **A robust library designed to streamline interaction with several large language models (LLMs) providers like Cohere, 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

os.environ["COHERE_API_KEY"] = "wiytrjNSrookxQDxxkpYc1wixxxxxxx”

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"})
```

# What makes LangChain Chains so Important?

Effortlessly link various LLMs, amalgamating the strengths of different models.

They augment the abilities of traditional single-model arrangements, paving the way for inventive solutions to intricate problems.

By bridging the gaps between models in an accessible way, they could become an invaluable tool for developers worldwide.

# Using Multiple Chains

```python
from langchain.llms import Cohere
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
from langchain.chains import SimpleSequentialChain

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

## Chain 1.

```python
template = """Your job is to come up with a classic dish from the area that the users suggests.
% USER LOCATION
{user_location}

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

location_chain = LLMChain(llm=llm, prompt=prompt_template)
```

## 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_template = PromptTemplate(input_variables=["user_meal"], template=template)

# Holds my 'meal' chain
meal_chain = LLMChain(llm=llm, prompt=prompt_template)
```

# Using Multiple Chains

- You'll notice that we haven't explicitly mentioned input_variables for SimpleSequentialChain.
- This is because the output from chain 1 is passed as input to chain 2.
- Now, let's run this chain:

```
overall_chain = SimpleSequentialChain(chains=[location_chain, meal_chain], verbose=True)
```

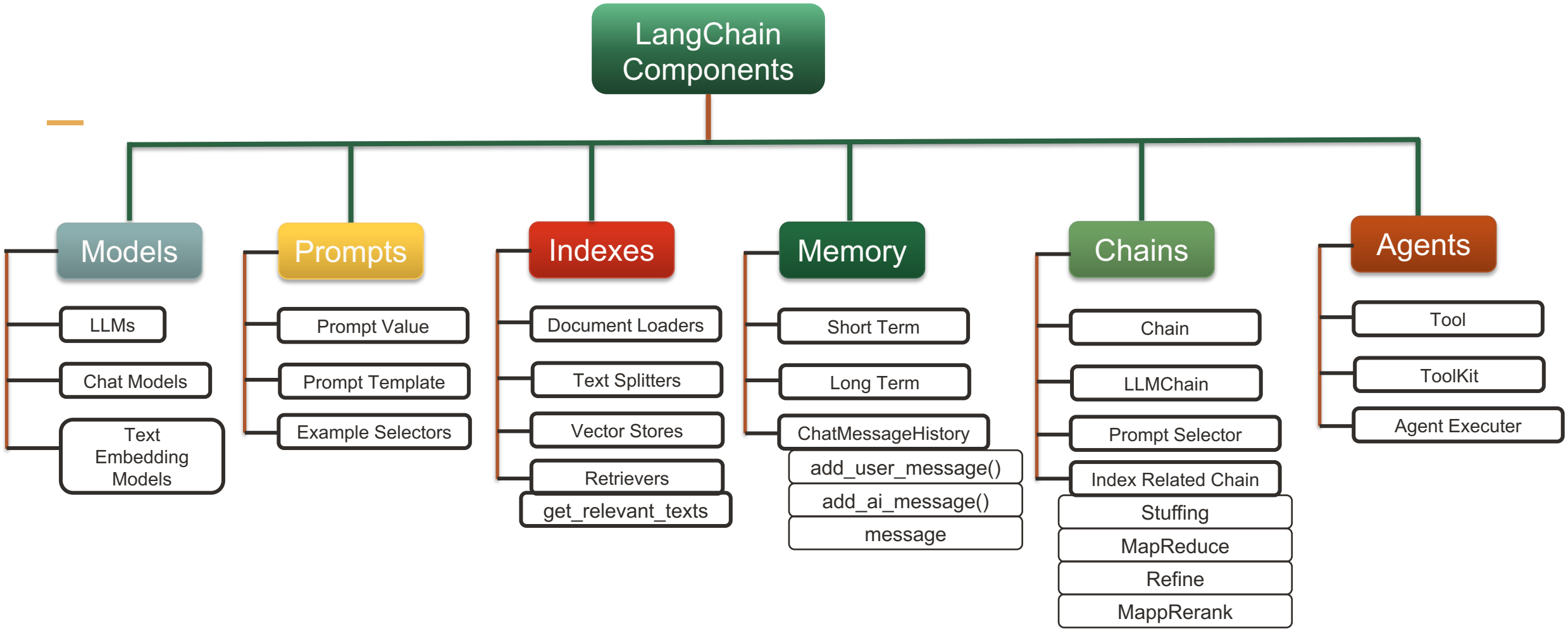
```
review = overall_chain.invoke("Rome")
```

```
> 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
LLMs
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

# Text

The natural language way to interact with LLMs

You'll be working with simple strings.

[87]:
```
# You'll be working with simple strings (that'll soon grow in complexity!)
my_text = "What day comes after Friday?"
my_text
```

[87]:
```
'What day comes after Friday?'
```

# Chat Messages

Like text, but 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 storing details and metadata about the model's training data.*
- *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.*

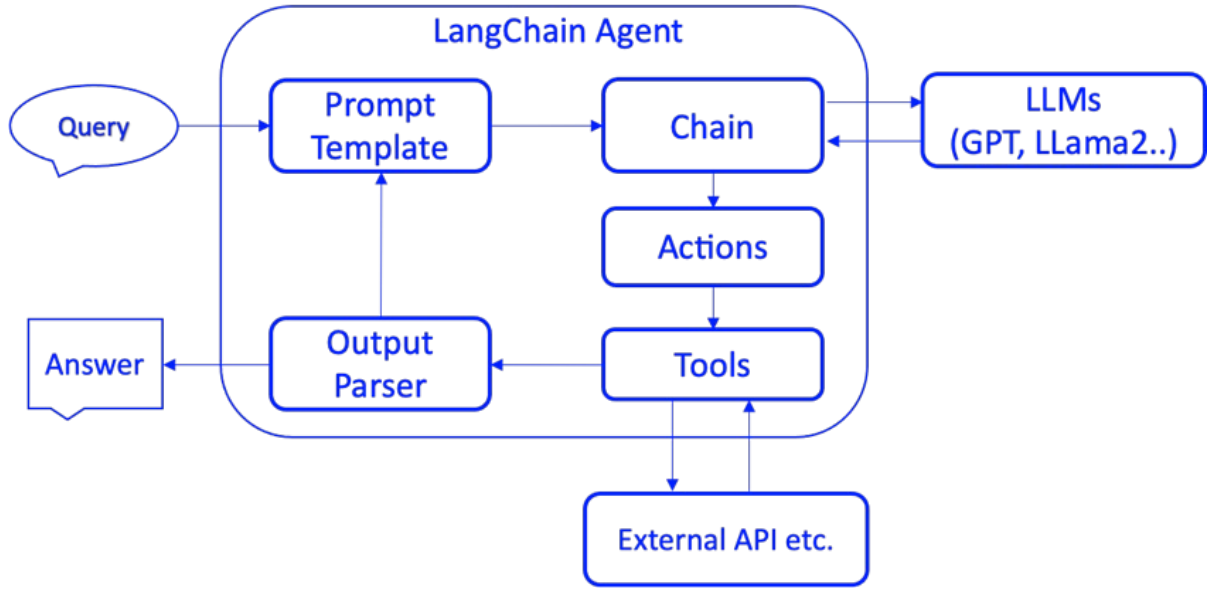
## Example Selectors

- *Identify appropriate instances from the model's training data.*
- *Can be adjusted to favour certain examples or filter out unrelated ones, providing a tailored AI response based on user input.*

# Agents in LangChain

## Agents

- Agents in LangChain present an innovative way to dynamically call LLMs based on user input.
- They not only have access to an LLM but also a suite of tools (like Google Search, Python REPL, math calculator, weather APIs, etc.) that can interact with the outside world.



# Summarization

Summarization chains compress large documents into shorter forms without losing the essence.

## Content Understanding

Analysing and understanding the main points of the content.

## Salience Detection

Identifying the most important pieces of information.

## Rewriting and Paraphrasing

Condensing and paraphrasing the salient points.

## Coherence Checking

Ensuring the summary is logical and fluent.

# Summarization Chain

Easily run through long numerous documents and get a summary.

[21]:
```
from langchain.chains.summarize import load_summarize_chain
from langchain.document_loaders import TextLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter

loader = TextLoader('data/disc.txt')
documents = loader.load()

# Get your splitter ready
text_splitter = RecursiveCharacterTextSplitter(chunk_size=700, chunk_overlap=50)

# Split your docs into texts
texts = text_splitter.split_documents(documents)

# There is a lot of complexity hidden in this one line. I encourage you to check out the video above for more detail
chain = load_summarize_chain(llm, chain_type="map_reduce", verbose=True)
chain.run(texts)
```

# Conversational Retrieval

Are designed to handle the dynamic nature of human conversation. Each component plays a specific role. For instance:

**Input Understanding**

The system interprets the user's input, which involves understanding the context, intent, and entities mentioned.

**Retrieval**

The chain then retrieves information relevant to the query.

As the conversation progresses, the system maintains the context through memory or context stacks.

**Context Management**

The system generates a response that is coherent and relevant to the ongoing conversation.

**Response Generation**

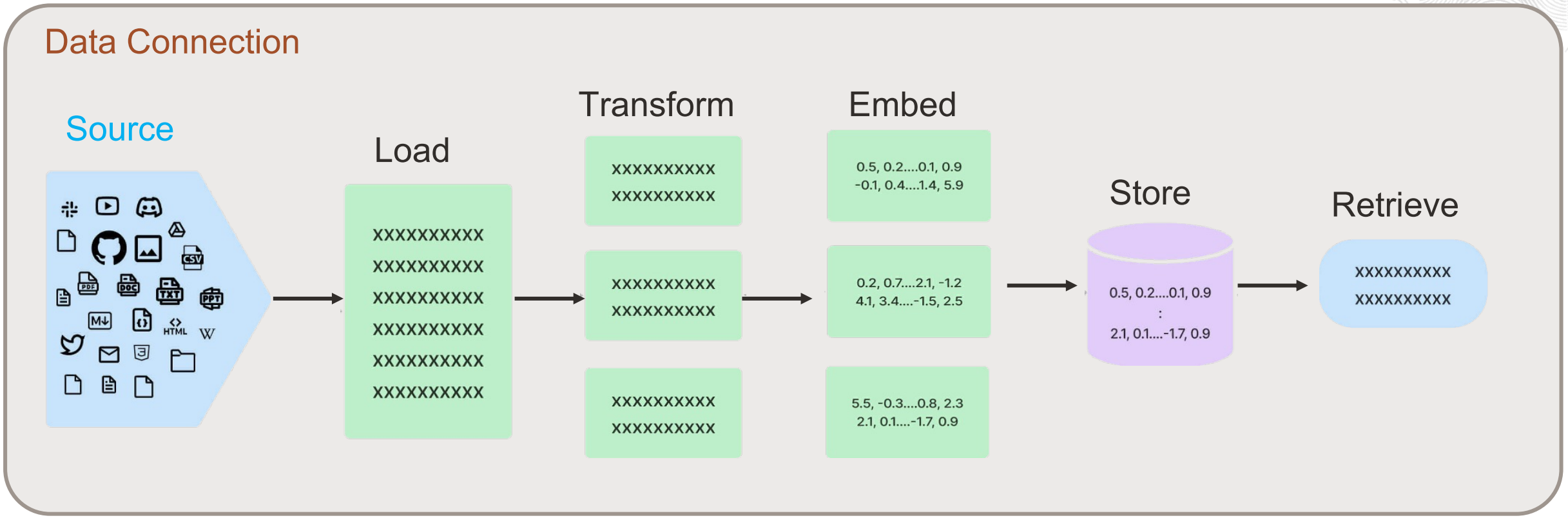
# Retrieval QA

Retrieval QA systems typically handle single-turn interactions.

The chain here includes:

1. **Question Processing**: Interpretation of the user's query to understand the question's intent.
2. **Document Retrieval**: Finding relevant documents or data sources that contain potential answers.
3. **Answer Extraction**: Identifying exact or paraphrased answers from the retrieved documents.
4. **Answer Ranking**: Scoring the possible answers based on relevance and accuracy.

# Retrieval


Data Connection
Source
Load
Transform
Embed
Store
Retrieve
xxxxxxxxxx
xxxxxxxxxx
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 types of locations (private S3 buckets, public websites).

# Working with text splitters

## 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.

# 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 different embedding providers and methods, from open-source to proprietary API, allowing you to choose the one best suited for your needs.
- 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 (as 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

LangChain offers many different types of text splitters.

Check the table on next slide with all of them, along with a few characteristics:

**Name**: Name of the text splitter

- **Splits On**: How this text splitter splits text
- **Adds Metadata**: Whether or not this text splitter adds metadata about where each chunk came from.
- **Description**: Description of the splitter, including recommendation on when to use it.

# 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,
```

# API Chains

In the context of APIs, chains refer to a sequence of API calls where the output of one call is used as input for the next.

For example, in an NLP application:

1. **Data Retrieval API**: Initially, an API call retrieves raw data from a database or external source.
2. **Processing API**: The data is then passed to another API for processing, such as language detection or entity recognition.
3. **Analysis API**: Further, the processed data might be sent through an analytical model for sentiment analysis or classification.
4. **Response API**: Finally, an API returns the processed and analyzed data to the end-user or another system for use.

# 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 for efficiency and accuracy.

# Thank You