A Comprehensive LangChain Guide

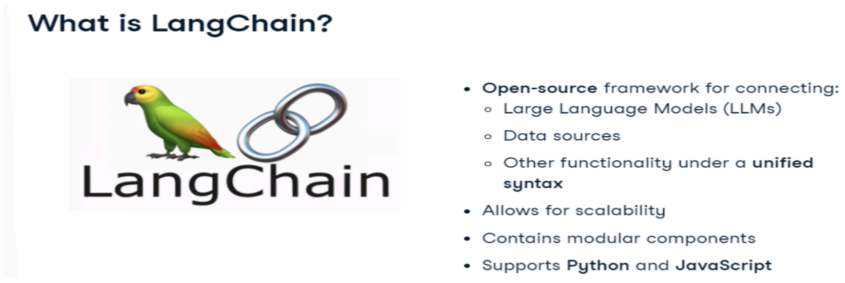
Table of Contents

---

1. Introduction to LangChain

What is LangChain?

LangChain is a Python library designed to simplify the development of applications that utilize large language models (LLMs), such as those from OpenAI, Hugging Face, and other providers. As artificial intelligence evolves, LLMs have proven to be powerful tools across industries, enabling applications that generate text, answer questions, summarize documents, and even assist with decision-making processes. However, building sophisticated applications using LLMs can be challenging due to the complexities of chaining workflows, managing contextual interactions, and ensuring optimal integration with the models themselves. LangChain addresses these challenges by providing a modular, composable framework that allows developers to design, manage, and deploy advanced LLM-based applications more easily.



What is LangChain?

- **Open-source** framework for connecting:
  - Large Language Models (LLMs)
  - Data sources
  - Other functionality under a **unified syntax**
- Allows for scalability
- Contains modular components
- Supports **Python** and **JavaScript**

Why is LangChain Necessary?

LangChain abstracts complex interactions with language models, allowing developers to focus on application logic instead of integration details. It supports chaining multiple LLMs, managing prompts, and deploying agents to automate workflows. LangChain is ideal for complex applications that require modularity, prompt management, or multi-step workflows, like chatbots, retrieval-augmented generation, and decision-making systems.

LangChain's design philosophy centers on the need for flexibility and modularity in working with LLMs. Without LangChain, developers face several challenges:

1. **Complex Workflow Management**: LLMs often require a series of tasks to deliver useful results. For example, a conversational AI might need a setup where context is maintained over multiple messages, memory is retained, and previous responses are taken into account. Implementing such workflows without a standardized library can lead to fragmented code and maintenance headaches.
2. **Prompt Management**: Writing prompts for LLMs requires a balance of precision and flexibility. Prompts must be structured to ensure that the model understands the intent and context while allowing room for varied responses. Without LangChain's templates, managing prompts and dynamically adjusting them becomes tedious, especially in applications with multiple prompts or varied tasks.
3. **Multi-Step Processing**: Many applications need LLMs to interact with various other tools, perform calculations, or retrieve data from external sources before providing a final response. LangChain allows developers to chain these operations seamlessly, creating a pipeline where each component can depend on the previous one's output.
4. **Efficient Interaction with LLMs and APIs**: LangChain integrates with popular APIs, providing out-of-the-box support for various LLM providers. This integration simplifies the process of switching between providers or models without extensive code rewrites.
5. **Enhanced Scalability and Modularity**: By breaking down complex interactions into reusable components, LangChain encourages clean, modular code that's easier to debug, extend, and maintain. This approach also makes it straightforward to scale applications or integrate new features without disrupting existing functionality.

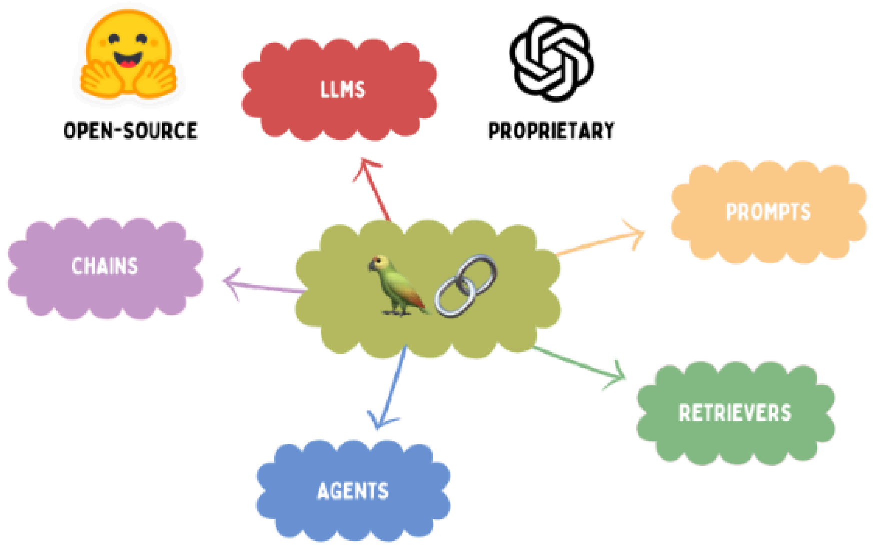
Core Capabilities of LangChain

LangChain provides several key modules that together create a robust environment for developing LLM-based applications:

1. **LLM Wrappers**: These wrappers enable developers to interact with LLMs through a common interface, abstracting away the complexities of different APIs. For example, with the same `OpenAI` wrapper, you can easily switch models by changing configuration parameters.
2. **Prompt Templates**: LangChain offers tools for crafting structured prompts that can adapt to different inputs, enhancing consistency in responses across varied queries. Prompt templates are ideal for building applications where prompts are dynamically generated, such as chatbots, Q&A systems, or content generation tools.
3. **Chains**: LangChain's chaining capabilities allow developers to link multiple components together, creating workflows where each step relies on the previous one's output. For instance, a summarization workflow might involve a pre-processing step, followed by a text generation step,

and finally a post-processing step. LangChain makes it easy to manage and execute these chains.
4. **Agents**: Agents in LangChain are powerful components that can act autonomously, interpreting user inputs and performing actions accordingly. Agents can integrate with external APIs, databases, or custom tools, allowing for applications where LLMs can make decisions, execute tasks, or retrieve information as needed.
5. **Memory Management**: For applications like chatbots, maintaining memory across interactions is essential for a natural and coherent user experience. LangChain provides memory modules that can store conversation history, remember context, and summarize previous interactions, enabling long-term memory for dialogue-based applications.
6. **Integrations and Plugins**: LangChain has built-in support for popular APIs (e.g., Hugging Face, OpenAI), document loaders (e.g., for PDFs, CSVs), and tools for working with structured data. This ecosystem of plugins ensures that LangChain is compatible with a wide range of applications, from document-based analysis to data retrieval and embedding.
7. **Advanced Components**: Features like Retrieval Augmented Generation (RAG) workflows, vector databases, and custom agents make LangChain suitable for complex use cases. RAG workflows, for example, enable applications to retrieve specific information from large datasets, enhancing response relevance in data-intensive applications.
8. **Deployment and Monitoring Tools**: LangChain's ecosystem includes tools like LangServe for easy deployment and Langsmith for debugging and performance evaluation, making it production-ready and manageable at scale. Features like LangGraph allow multi-agent coordination, and SCIPE (Systematic Chain Improvement and Problem Evaluation) assists with optimizing workflows.

Core components of LangChain



Example Use Cases of LangChain

LangChain’s modularity and flexibility make it ideal for a wide range of applications:

- **Conversational AI**: Building chatbots that retain memory over long conversations, ensuring contextually aware interactions, is straightforward with LangChain’s memory and prompt management features.
- **Q&A Systems**: LangChain allows integration with databases and external APIs, enabling it to fetch information and answer complex queries. With RAG workflows and data retrievers, LangChain can build high-quality, fact-based Q&A systems.
- **Document Analysis**: Document loaders allow users to parse and analyze documents like PDFs and CSVs. Combined with summarization chains and LLMs, LangChain makes it easy to create tools that summarize, search, or analyze documents in bulk.
- **Automation and Task Execution**: Agents can perform predefined tasks or retrieve real-time data by interacting with APIs. For example, an agent could act as a personal assistant, checking emails, fetching weather information, or setting reminders autonomously.
- **Content Generation and Summarization**: LangChain’s prompt templates and chains make it ideal for creating structured content, such as articles, reports, or summaries, that maintain coherence across multiple paragraphs or sections.

Installing LangChain

To get started, install LangChain using pip:

```
pip install langchain
```

---

2. Core Components of LangChain

LangChain’s core components are modular and include the following:

- **Chains:** Sequences of tasks executed in a specified order.
- **Agents:** Autonomous units that perform actions based on user input.
- **Retrievers:** Tools for fetching and processing data efficiently.
- **Prompts:** Structured instructions or queries for the LLM.
- **LLMs:** Language models used to process prompts.

Each component is designed to interact seamlessly with the others, making it easy to build and deploy sophisticated workflows.

2.1. Building Applications with LangChain LLM Wrapper

LangChain’s LLM wrapper makes it easy to interact with various language models. Here’s how to use it with OpenAI’s API:

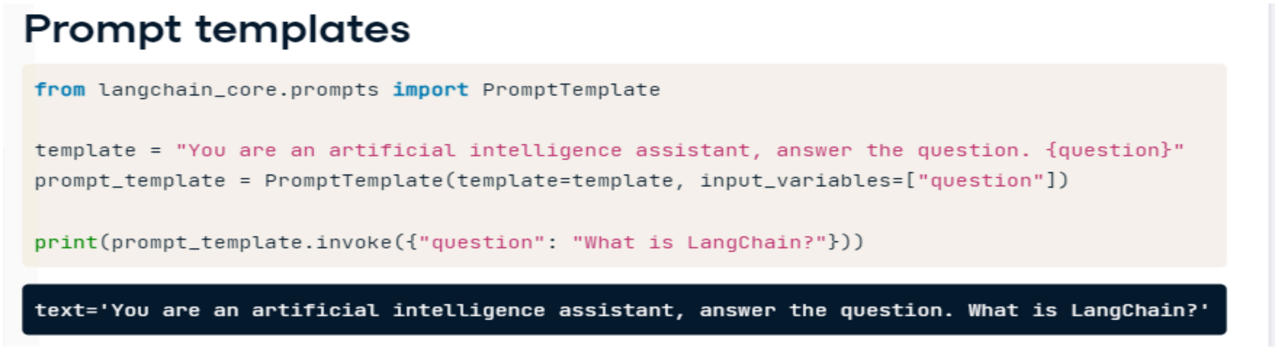
```
from langchain_openai import OpenAI

# Initialize the LLM with an API key
llm = OpenAI(
            model= “model_name”,
            api_key="your_openai_api_key")

# Generate a response
response = llm.invoke("What is the capital of France?")
print(response)
```

You can use similar wrappers for other models, like Hugging Face transformers, making it easy to switch between models as needed.

2.2 Prompt Templates in LangChain

Prompt templates standardize and structure prompts, making them reusable and dynamic for different tasks.

```
from langchain.prompts import PromptTemplate
# Define a template with placeholders
template = PromptTemplate(
    input_variables=["city"],
    template="Tell me an interesting fact about {city}."
)

# Use the template
formatted_prompt = template.format(city="Paris")
print(formatted_prompt)
```



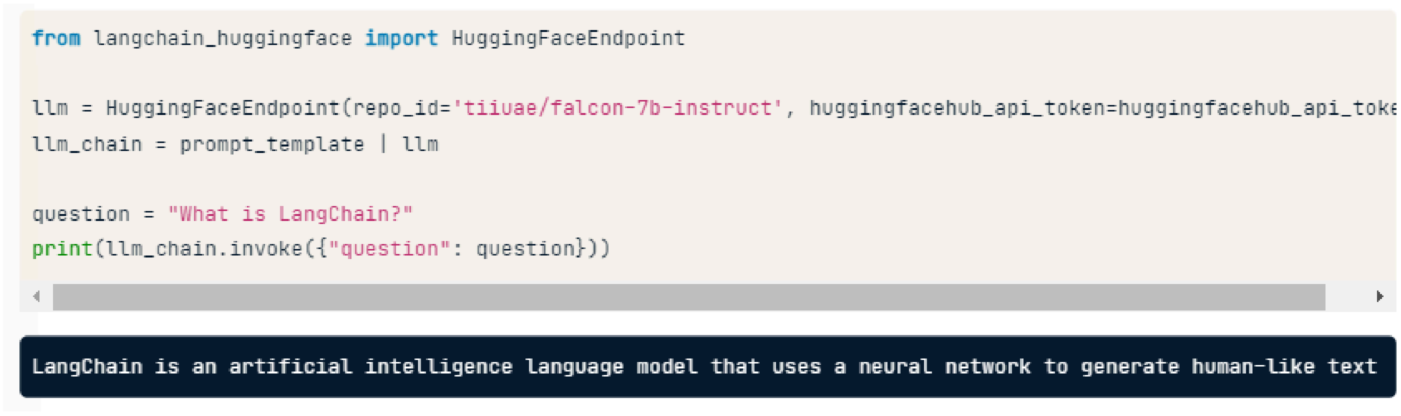
Prompt templates

```
from langchain_core.prompts import PromptTemplate

template = "You are an artificial intelligence assistant, answer the question. {question}"
prompt_template = PromptTemplate(template=template, input_variables=["question"])

print(prompt_template.invoke({"question": "What is LangChain?"}))
```

```
text='You are an artificial intelligence assistant, answer the question. What is LangChain?'
```

In this example, `PromptTemplate` formats the prompt dynamically. Using structured prompts helps maintain consistency across different tasks and ensures effective communication with LLMs.

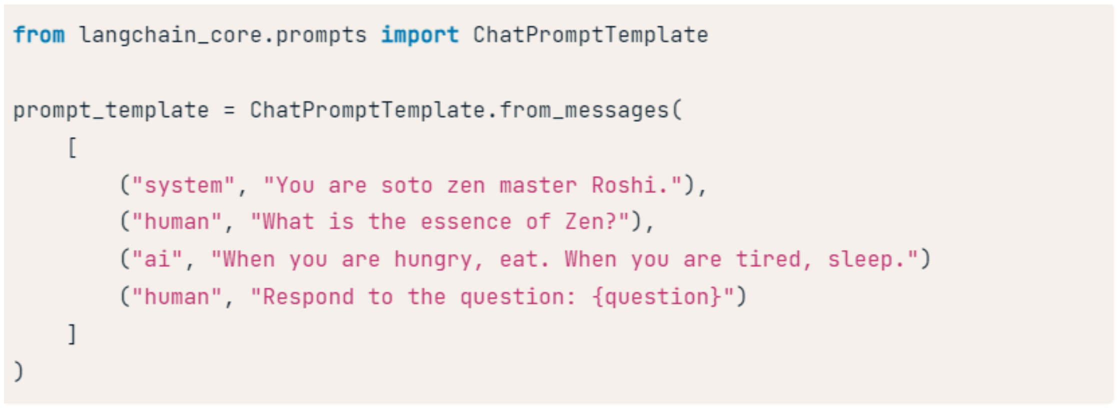
Integrating PromptTemplate with LLMs using Lanchain Expression Language (LCEL) :



```python
from langchain_huggingface import HuggingFaceEndpoint

llm = HuggingFaceEndpoint(repo_id='tiiuae/falcon-7b-instruct', huggingfacehub_api_token=huggingfacehub_api_toke
llm_chain = prompt_template | llm

question = "What is LangChain?"
print(llm_chain.invoke({"question": question}))
```

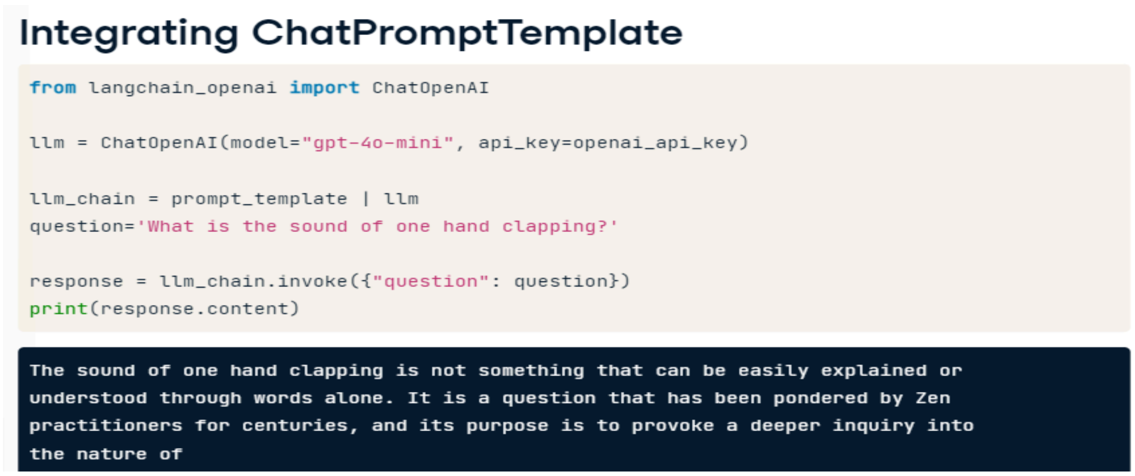
```
LangChain is an artificial intelligence language model that uses a neural network to generate human-like text
```

Chat models have different prompt template classes, and a series of messages are to be passed for the chat model. Three different roles were added.



```python
from langchain_core.prompts import ChatPromptTemplate

prompt_template = ChatPromptTemplate.from_messages(
    [
        ("system", "You are soto zen master Roshi."),
        ("human", "What is the essence of Zen?"),
        ("ai", "When you are hungry, eat. When you are tired, sleep.")
        ("human", "Respond to the question: {question}")
    ]
)
```

ChatOpenAI class is used to instantiate OpenAI models



Integrating ChatPromptTemplate

```python
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o-mini", api_key=openai_api_key)

llm_chain = prompt_template | llm
question='What is the sound of one hand clapping?'

response = llm_chain.invoke({"question": question})
print(response.content)
```

```
The sound of one hand clapping is not something that can be easily explained or
understood through words alone. It is a question that has been pondered by Zen
practitioners for centuries, and its purpose is to provoke a deeper inquiry into
the nature of
```

2.3 Chaining Components in LangChain

LangChain’s chaining capability allows you to combine prompts, models, and other components into a sequential workflow. This is useful for multi-step tasks like text summarization, question answering, and chatbots.

```
from langchain.chains import LLMChain

# Define a chain using an LLM and a prompt template
chain = LLMChain(llm=llm, prompt=template)

# Run the chain with parameters
result = chain.invoke({"city": "Tokyo"})
print(result)
```

This example shows a single-step chain, but chains can also be complex, involving multiple models and intermediate processing steps.

2.4 Agents in LangChain

Agents are designed to perform tasks autonomously by interpreting the inputs and taking actions. LangChain provides pre-built agents and allows you to define custom agents for specific workflows.

```
from langchain.agents import create_openai_agent

# Define a custom agent with OpenAI's LLM
agent = create_openai_agent(api_key="your_openai_api_key")

# Run the agent with an instruction
output = agent.invoke("Find the current weather in New York.")
print(output)
```

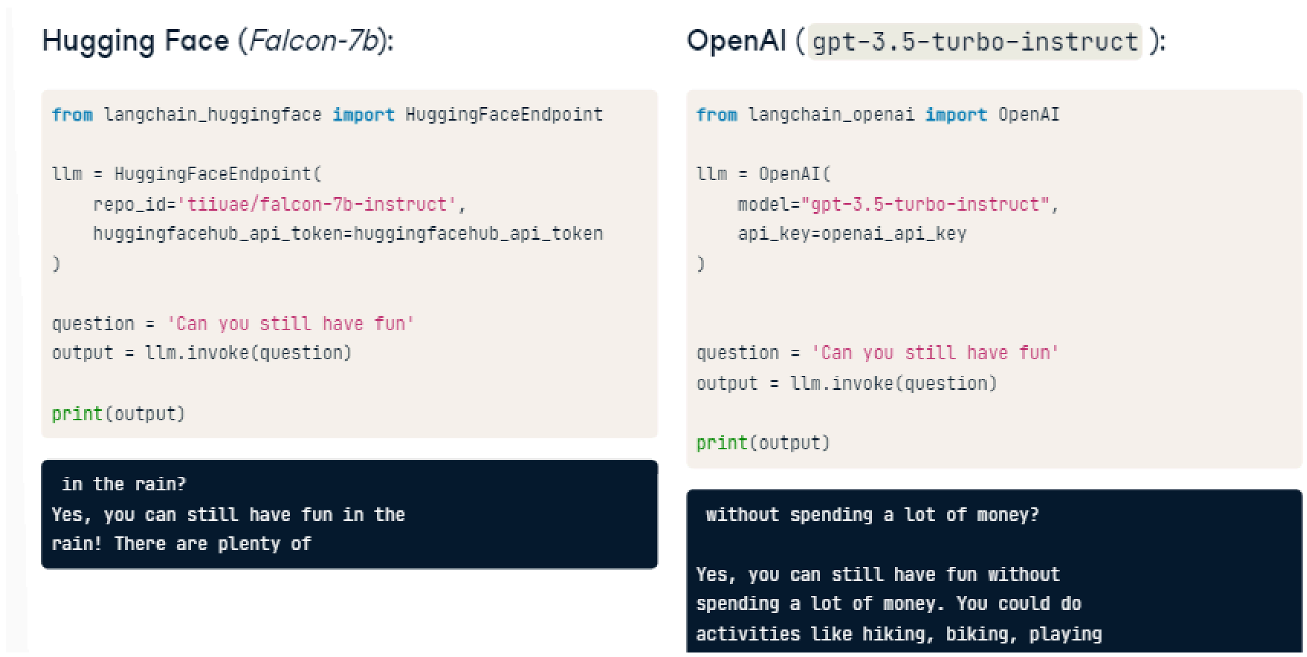
Agents can also interact with external systems, enabling applications to perform complex actions like fetching data from an API or interacting with a database.

---

3. LangChain Integrations

3.1 API Classes

LangChain integrates with APIs from providers like OpenAI and Hugging Face. This enables seamless interaction with models, regardless of the provider.



Hugging Face (*Falcon-7b*):

```python
from langchain_huggingface import HuggingFaceEndpoint

llm = HuggingFaceEndpoint(
    repo_id='tiiuae/falcon-7b-instruct',
    huggingfacehub_api_token=huggingfacehub_api_token
)

question = 'Can you still have fun'
output = llm.invoke(question)

print(output)
```

```
 in the rain?
Yes, you can still have fun in the
rain! There are plenty of
```

OpenAI (`gpt-3.5-turbo-instruct`):

```python
from langchain_openai import OpenAI

llm = OpenAI(
    model="gpt-3.5-turbo-instruct",
    api_key=openai_api_key
)


question = 'Can you still have fun'
output = llm.invoke(question)

print(output)
```

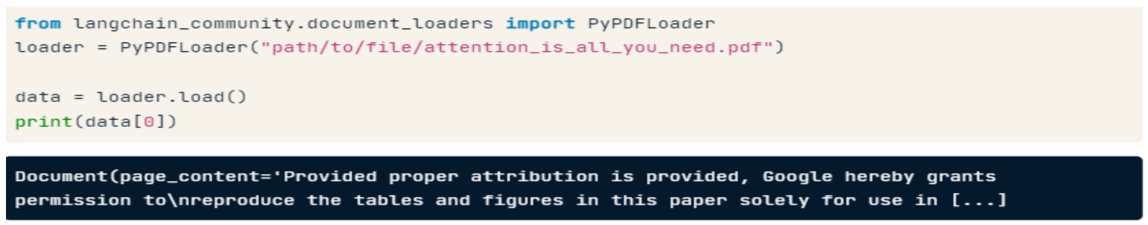
```
 without spending a lot of money?

Yes, you can still have fun without
spending a lot of money. You could do
activities like hiking, biking, playing
```

3.2 Document Loaders

Document loaders are available for managing and loading documents into workflows. Popular loaders include `pyPDFLoader` for PDFs and `csvDocumentLoader` for CSV files, simplifying document processing tasks.

```python
from langchain.document_loaders import pyPDFLoader

# Load a PDF document
loader = pyPDFLoader("example.pdf")
document = loader.load()
print(document[:500])  # Print the first 500 characters
```

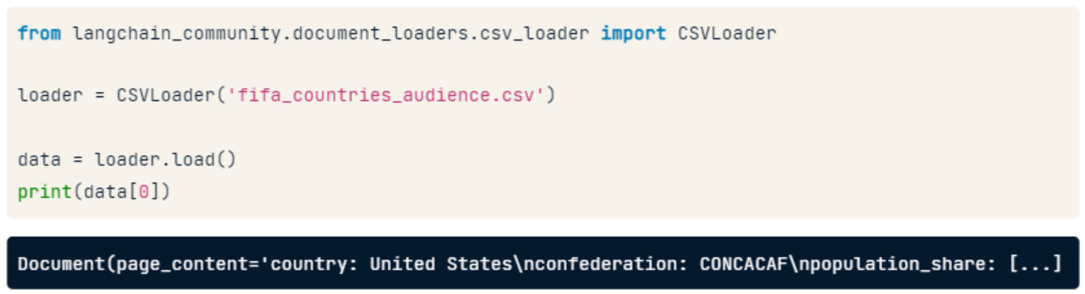
PDF Document Loader:



```python
from langchain_community.document_loaders import PyPDFLoader
loader = PyPDFLoader("path/to/file/attention_is_all_you_need.pdf")

data = loader.load()
print(data[0])
```

```
Document(page_content='Provided proper attribution is provided, Google hereby grants
permission to\nreproduce the tables and figures in this paper solely for use in [...]
```

CSV Loader:

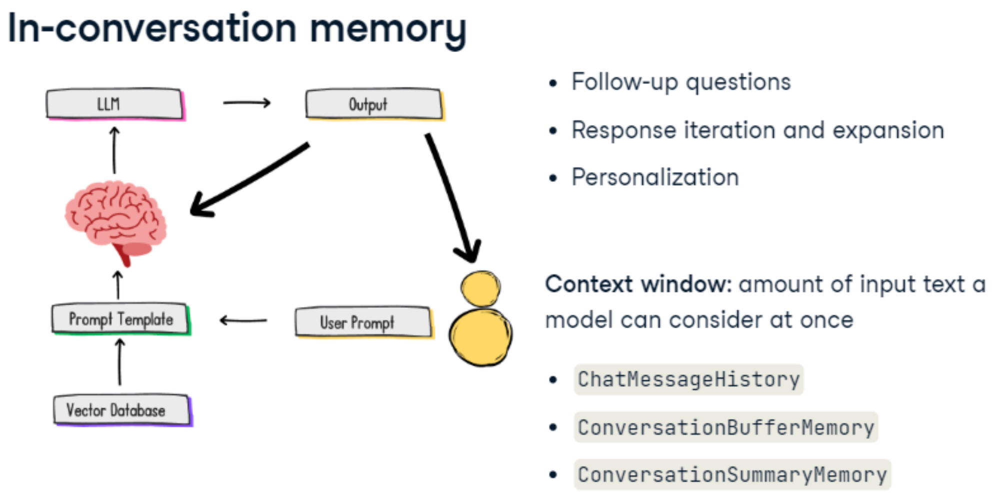


```
from langchain_community.document_loaders.csv_loader import CSVLoader

loader = CSVLoader('fifa_countries_audience.csv')

data = loader.load()
print(data[0])
```

```
Document(page_content='country: United States\nconfederation: CONCACAF\npopulation_share: [...]
```

---

4. LLM Memory & Management



In-conversation memory

- Follow-up questions
- Response iteration and expansion
- Personalization

Context window: amount of input text a model can consider at once

- `ChatMessageHistory`
- `ConversationBufferMemory`
- `ConversationSummaryMemory`

Memory models in LangChain retain context over multiple interactions, making them useful for chatbots and dialogue-based applications. Some key memory models include:

- **ChatMessageHistory- Stores Full Message History**
- **ConversationBufferMemory:** Stores a buffer of recent conversation history.
- **ConversationSummaryMemory:** Provides a summarized context over time, retaining important points while reducing verbosity.

```
from langchain.memory import ConversationBufferMemory

# Initialize memory and store conversation
memory = ConversationBufferMemory()
memory.add_message("User", "What is the weather like?")
memory.add_message("Assistant", "It’s sunny today.")
print(memory.buffer)
```

These memory types ensure that long interactions remain coherent, enhancing user experience.

4.1 ChatMessageHistory

- Stores **full message history**

```python
from langchain.memory import ChatMessageHistory
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model="gpt-4o-mini", api_key=openai_api_key)

history = ChatMessageHistory()
history.add_ai_message("Hi! Ask me anything about LangChain.")
history.add_user_message("Describe a metaphor for learning LangChain in one sentence.")

response = llm.invoke(history.messages)
print(response.content)
```

```
Learning LangChain is like unraveling a complex tapestry of interconnected languages, each thread
revealing a new layer of linguistic understanding.
```

```python
history.add_user_message("Summarize the preceding sentence in fewer words")

response = llm.invoke(history.messages)
print(response.content)
```

```
LangChain is a linguistic bridge that connects learners to a world of new languages.
```

4.2 Conversation BufferMemory

```python
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationChain

llm = ChatOpenAI(model="gpt-4o-mini", api_key=openai_api_key)
memory = ConversationBufferMemory(size=4)

buffer_chain = ConversationChain(llm=llm, memory=memory)
```

```python
buffer_chain.invoke("Describe a language model in one sentence")
buffer_chain.invoke("Describe it again using less words")
buffer_chain.invoke("Describe it again fewer words but at least one word")
buffer_chain.invoke("What did I first ask you? I forgot.")
```

```
{'input': 'What did I first ask you? I forgot.',
 'history': 'Human: Describe a language model in one sentence\nAI:  A language model is a ...',
 'response': ' You asked me to describe a language model in one sentence.'}
```

4.3 Conversation SummaryMemory

- *Summarizes* the history to condense information

```
from langchain.memory import ConversationSummaryMemory

llm = ChatOpenAI(model="gpt-4o-mini", api_key=openai_api_key)

memory = ConversationSummaryMemory(llm=llm)

summary_chain = ConversationChain(llm=llm, memory=memory, verbose=True)
```

```
summary_chain.invoke("Please summarize the future in 2 sentences.")
summary_chain.invoke("Why?")
summary_chain.invoke("What will I need to shape this?")
```

```
> Entering new ConversationChain chain...
...

> Finished chain.
{'input': 'What will I need to shape this?', 'history': 'The human asks the AI to summarize the future in
2 sentences. The AI predicts that rapid technological advancements will lead to increased automation and
interconnectedness, posing challenges related to ethics, privacy, and the impact of artificial intelligence
on jobs and daily life. The AI emphasizes that these advancements will shape the future, highlighting the
need to address ethical concerns and adapt to the changing landscape of technology.', 'response': 'To shape
the future, you will need to stay informed about emerging technologies, engage in discussions about ethical
considerations, and be open to adapting to new ways of living and working in a tech-driven world. It will
also be important to prioritize collaboration, innovation, and continuous learning to navigate the challenges
and opportunities that come with the evolving future.'}
```

---

5. Advanced Chains & Agents

5.0 LangChain Agents

What are agents?

Agents: use LLMs to take *actions*

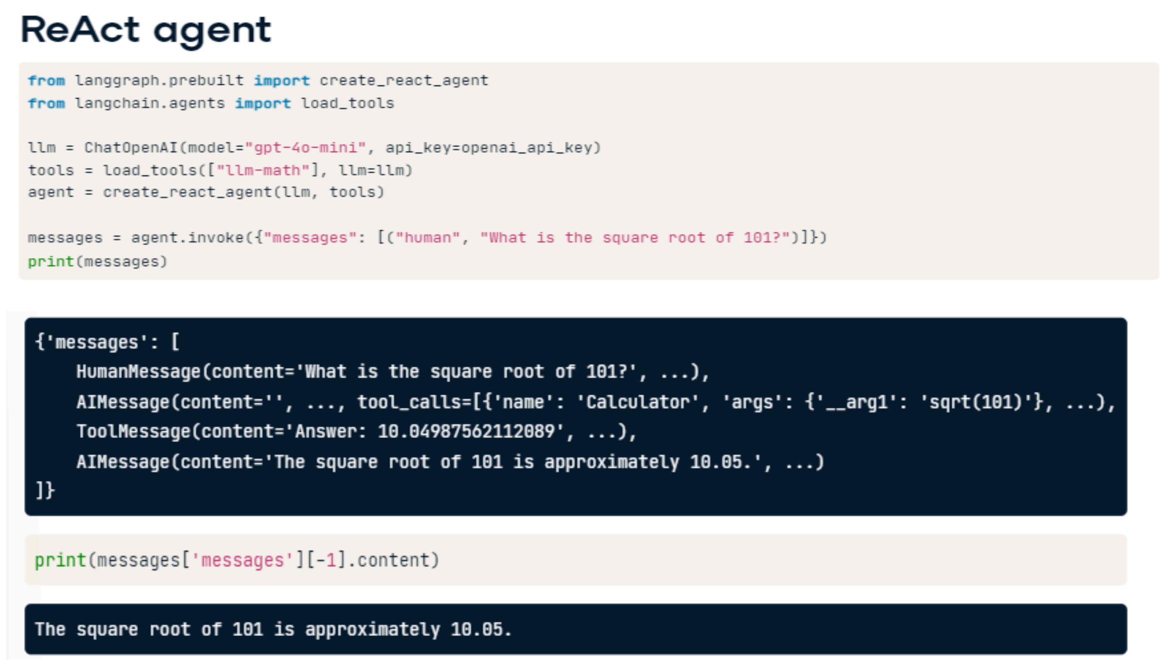
Tools: *functions* called by the agent

- **Now** → *ReAct* Agent

User Input: Why isn't my code working? Here it is
Run Code
Search Internet
Load Document
Agent
Tools

5.1 ReAct Agents

LangChain agents, such as **ReAct (Reason & Act) agents**, allow LLMs to perform tasks based on complex reasoning and make informed decisions. ReAct agents can call APIs, manage workflows, and interpret user intents.



ReAct agent

```python
from langgraph.prebuilt import create_react_agent
from langchain.agents import load_tools

llm = ChatOpenAI(model="gpt-4o-mini", api_key=openai_api_key)
tools = load_tools(["llm-math"], llm=llm)
agent = create_react_agent(llm, tools)

messages = agent.invoke({"messages": [("human", "What is the square root of 101?")]})
print(messages)
```

```
{'messages': [
    HumanMessage(content='What is the square root of 101?', ...),
    AIMessage(content='', ..., tool_calls=[{'name': 'Calculator', 'args': {'__arg1': 'sqrt(101)'}, ...),
    ToolMessage(content='Answer: 10.04987562112089', ...),
    AIMessage(content='The square root of 101 is approximately 10.05.', ...)
]}
```

```python
print(messages['messages'][-1].content)
```

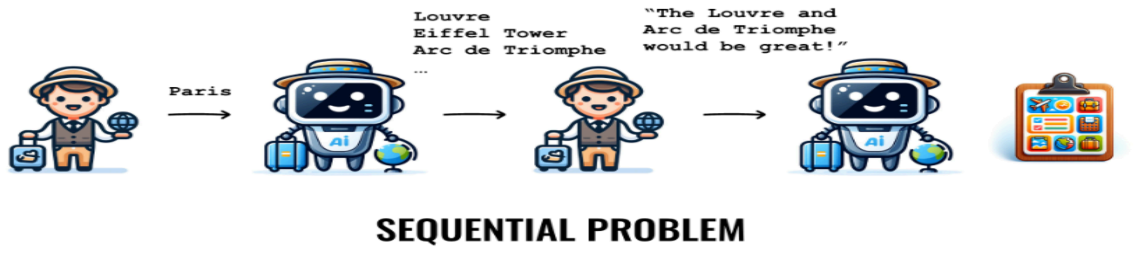
```
The square root of 101 is approximately 10.05.
```

5.2 Custom Tools

You can define custom tools for agents, allowing them to interact with APIs, perform database queries, or trigger workflows. Here's a simplified example:

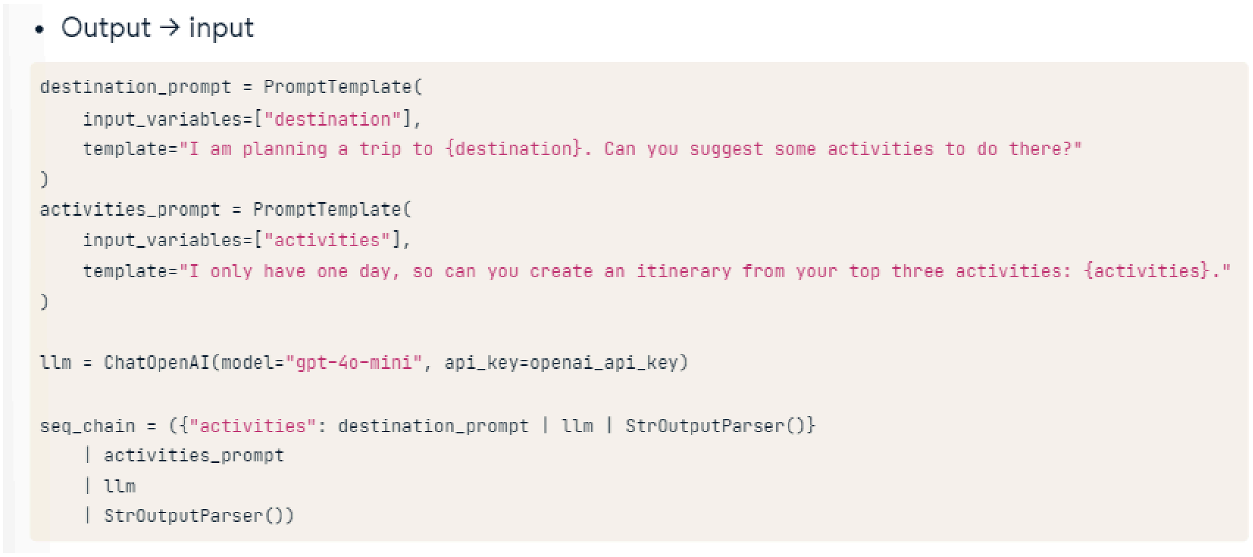
```python
from langchain.agents import Tool

# Define a custom tool
def check_stock_price(symbol):
    # Mock function to simulate stock price retrieval
    return f"The current stock price of {symbol} is $100."

tool = Tool(name="StockPriceTool", func=check_stock_price)
result = tool.invoke("AAPL")
print(result)
```

5.3 Sequential Chains



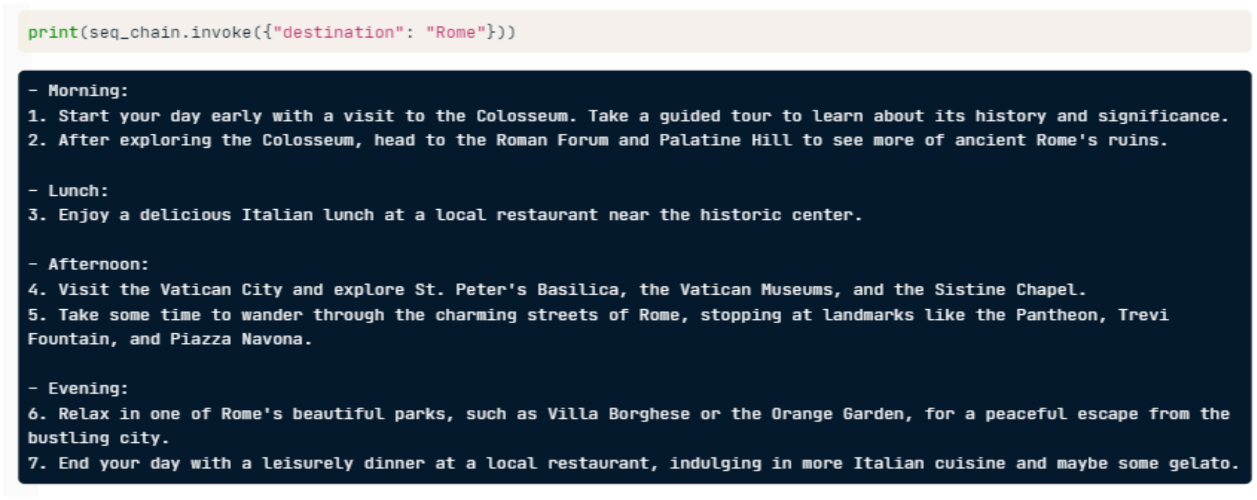
LangChain’s **Sequential Chains** allow you to build workflows where one component’s output becomes the input for the next, creating a multi-step process.

In Sequential Chains output from one chain becomes input to other chain



- Output → input

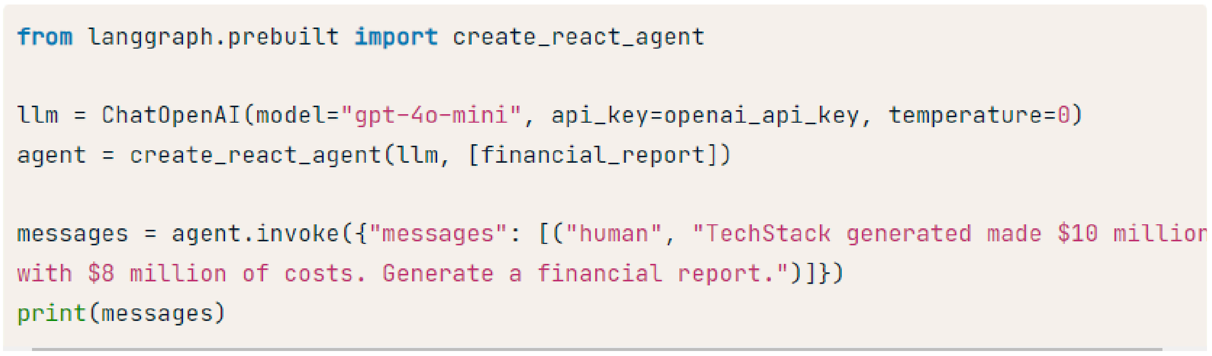
```python
destination_prompt = PromptTemplate(
    input_variables=["destination"],
    template="I am planning a trip to {destination}. Can you suggest some activities to do there?"
)
activities_prompt = PromptTemplate(
    input_variables=["activities"],
    template="I only have one day, so can you create an itinerary from your top three activities: {activities}."
)

llm = ChatOpenAI(model="gpt-4o-mini", api_key=openai_api_key)

seq_chain = ({"activities": destination_prompt | llm | StrOutputParser()}
    | activities_prompt
    | llm
    | StrOutputParser())
```



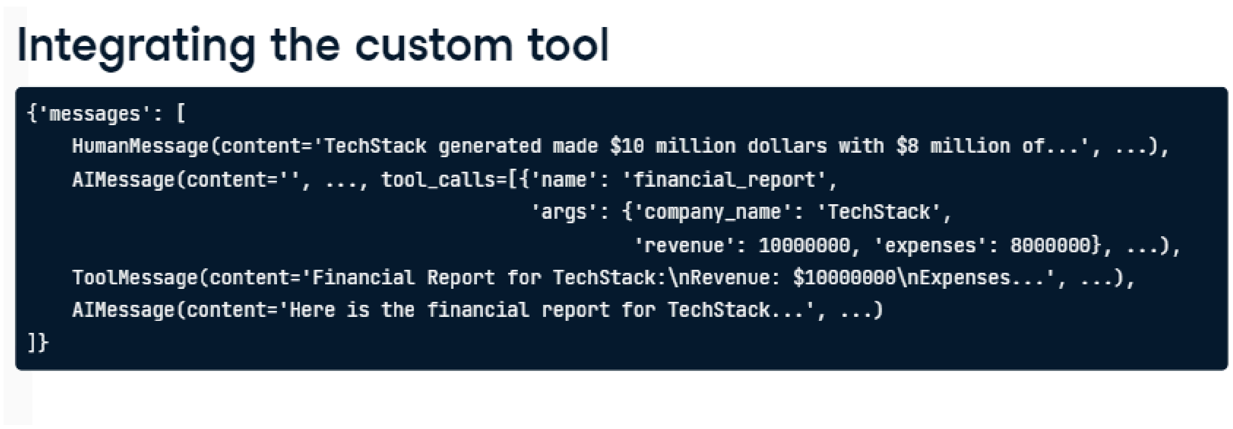
```python
print(seq_chain.invoke({"destination": "Rome"}))
```

```
- Morning:
1. Start your day early with a visit to the Colosseum. Take a guided tour to learn about its history and significance.
2. After exploring the Colosseum, head to the Roman Forum and Palatine Hill to see more of ancient Rome's ruins.

- Lunch:
3. Enjoy a delicious Italian lunch at a local restaurant near the historic center.

- Afternoon:
4. Visit the Vatican City and explore St. Peter's Basilica, the Vatican Museums, and the Sistine Chapel.
5. Take some time to wander through the charming streets of Rome, stopping at landmarks like the Pantheon, Trevi
Fountain, and Piazza Navona.

- Evening:
6. Relax in one of Rome's beautiful parks, such as Villa Borghese or the Orange Garden, for a peaceful escape from the
bustling city.
7. End your day with a leisurely dinner at a local restaurant, indulging in more Italian cuisine and maybe some gelato.
```

Integrating the Custom Tool:



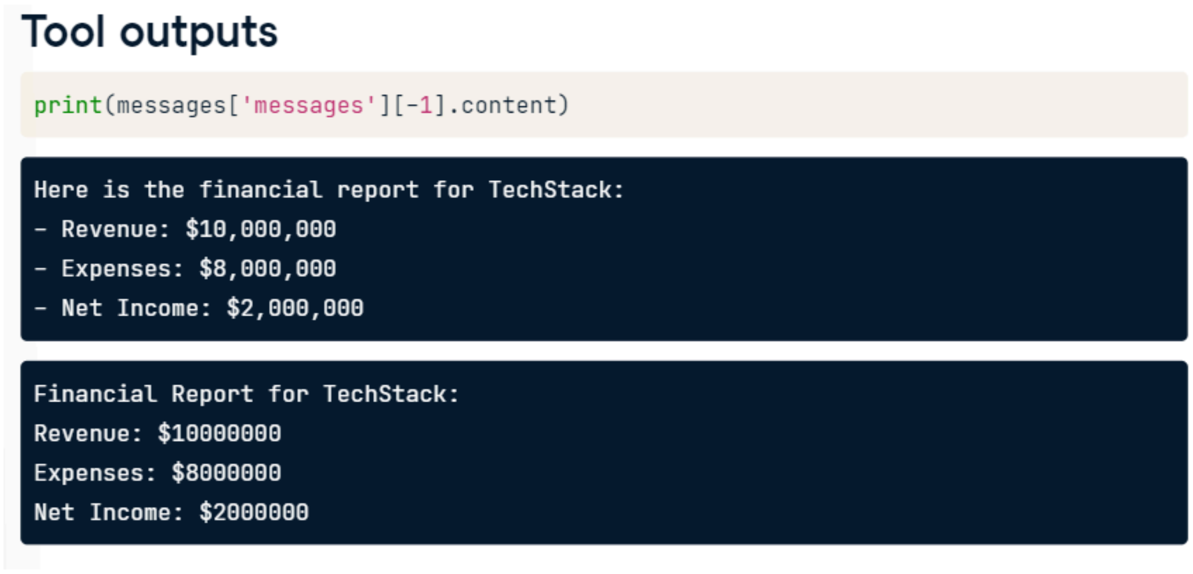
```python
from langgraph.prebuilt import create_react_agent

llm = ChatOpenAI(model="gpt-4o-mini", api_key=openai_api_key, temperature=0)
agent = create_react_agent(llm, [financial_report])

messages = agent.invoke({"messages": [("human", "TechStack generated made $10 millio
with $8 million of costs. Generate a financial report.")]})
print(messages)
```



Integrating the custom tool

```
{'messages': [
    HumanMessage(content='TechStack generated made $10 million dollars with $8 million of...', ...),
    AIMessage(content='', ..., tool_calls=[{'name': 'financial_report',
                                            'args': {'company_name': 'TechStack',
                                                     'revenue': 10000000, 'expenses': 8000000}, ...),
    ToolMessage(content='Financial Report for TechStack:\nRevenue: $10000000\nExpenses...', ...),
    AIMessage(content='Here is the financial report for TechStack...', ...)
]}
```

Output:



Tool outputs

```
print(messages['messages'][-1].content)
```

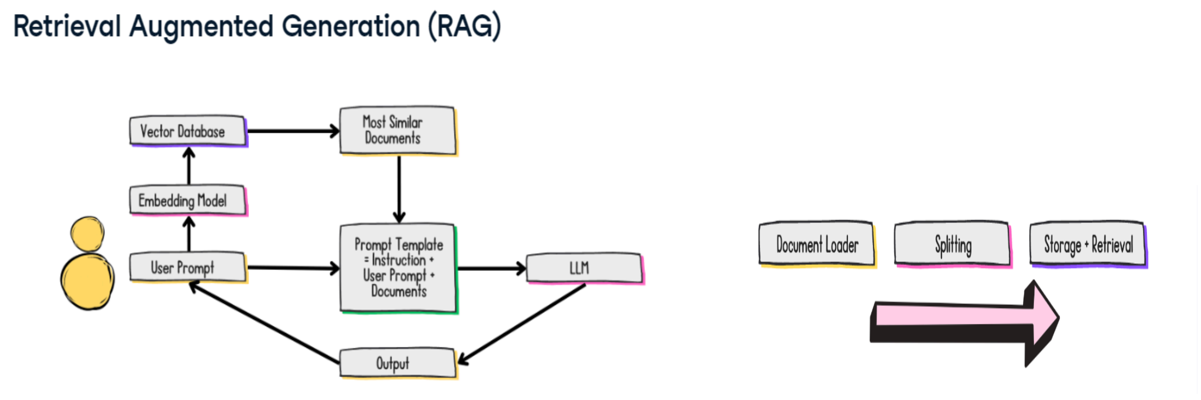
```
Here is the financial report for TechStack:
- Revenue: $10,000,000
- Expenses: $8,000,000
- Net Income: $2,000,000
```

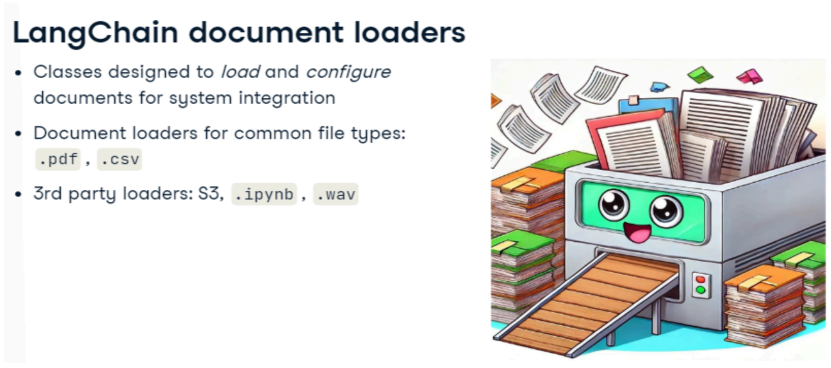
```
Financial Report for TechStack:
Revenue: $10000000
Expenses: $8000000
Net Income: $2000000
```

6. Retrieval Augmented Generation Workflows (RAG)

Retrieval Augmented Generation (RAG) combines retrieval of information with LLMs to enhance applications requiring fact-based responses.

RAG storage and retrieval using Vector databases




LangChain document loaders

- Classes designed to *load* and *configure* documents for system integration
- Document loaders for common file types: `.pdf`, `.csv`
- 3rd party loaders: S3, `.ipynb`, `.wav`

Key RAG tools in LangChain include:

- **6.1 Data Splitters:** Tools like `CharacterTextSplitter` and `RecursiveCharacterTextSplitter` divide data into chunks for easier retrieval.
- **6.2 Vector Databases:** Store and retrieve embeddings for efficient document retrieval

6.1.1 Splitting external data for retrieval:

1. Document Splitting into Chunks
2. Break Documents to Fit within the Context Window

```
from langchain.splitters import CharacterTextSplitter
splitter = CharacterTextSplitter(chunk_size=200, chunk_overlap=50)
chunks = splitter.split("This is a long document that needs splitting.")
print(chunks)
```

6.1.2 Splitting characters- CharacterTextSplitter

```
quote = '''One machine can do the work of fifty ordinary humans.\nNo machine can do
the work of one extraordinary human.'''
```

```
len(quote)
```

```
103
```

```
chunk_size = 24
chunk_overlap = 3
```

Using Langchain Character TextSplitter:

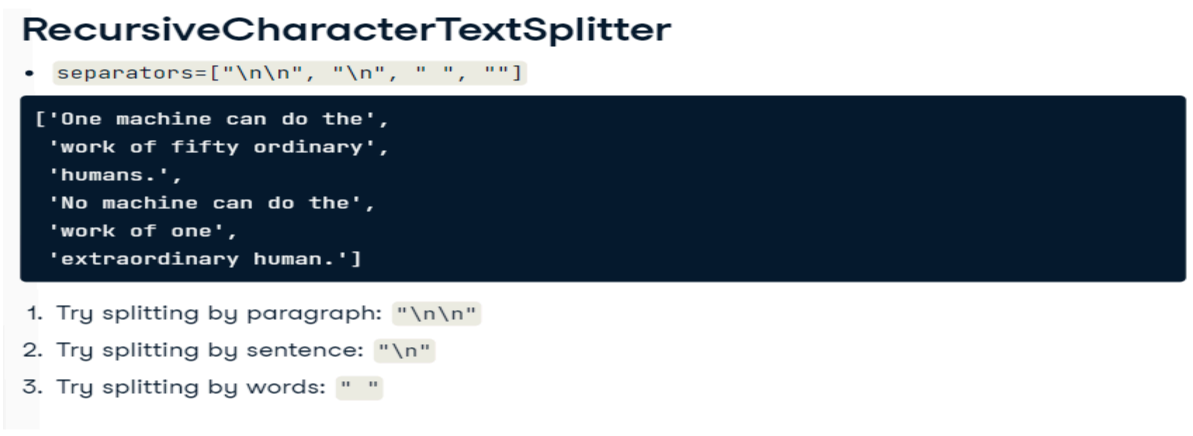
```
from langchain_text_splitters import CharacterTextSplitter

ct_splitter = CharacterTextSplitter(
    separator='.',
    chunk_size=chunk_size,
    chunk_overlap=chunk_overlap)

docs = ct_splitter.split_text(quote)
print(docs)
print([len(doc) for doc in docs])
```

```
['One machine can do the work of fifty ordinary humans',
 'No machine can do the work of one extraordinary human']
[52, 53]
```

- Split on separator so < `chunk_size`, but **may not always succeed!**

6.1.3 Recursively splitting by characters - RecursiveCharacterTextSplitter

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

rc_splitter = RecursiveCharacterTextSplitter(
    separators=["\n\n", "\n", " ", ""]
    chunk_size=chunk_size,
    chunk_overlap=chunk_overlap)

docs = rc_splitter.split_text(quote)
print(docs)
```



RecursiveCharacterTextSplitter

- `separators=["\n\n", "\n", " ", ""]`

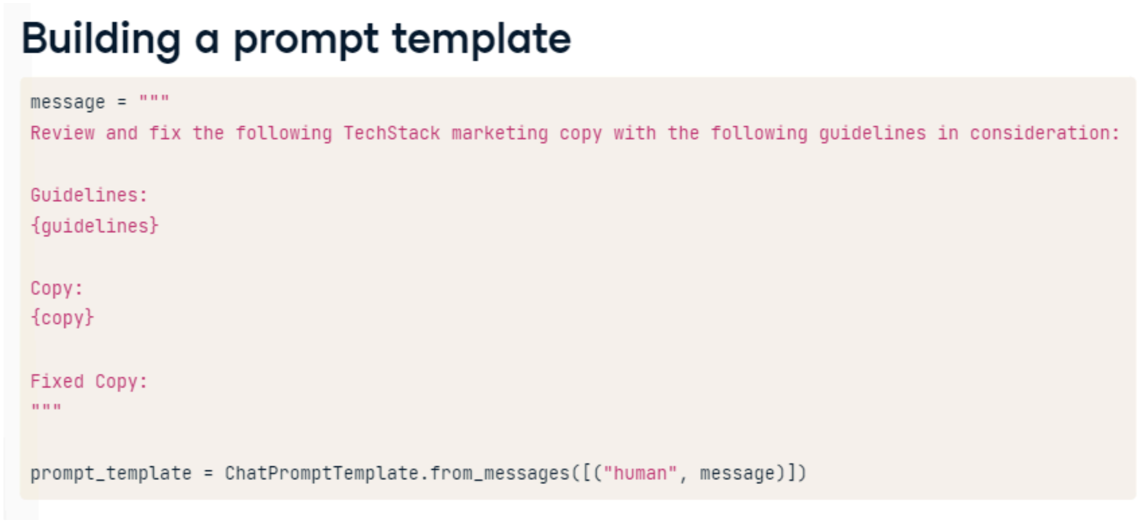
```
['One machine can do the',
 'work of fifty ordinary',
 'humans.',
 'No machine can do the',
 'work of one',
 'extraordinary human.']
```

1. Try splitting by paragraph: `"\n\n"`
2. Try splitting by sentence: `"\n"`
3. Try splitting by words: `" "`

6.2.1 Setting up the Chroma Db for Vector DataBase:

```
from langchain_openai import OpenAIEmbeddings
from langchain_chroma import Chroma

embedding_function = OpenAIEmbeddings(api_key=openai_api_key, model='text-embedding-3-small')

vectorstore = Chroma.from_documents(
    docs,
    embedding=embedding_function,
    persist_directory="path/to/directory"
)

retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 2}
)
```

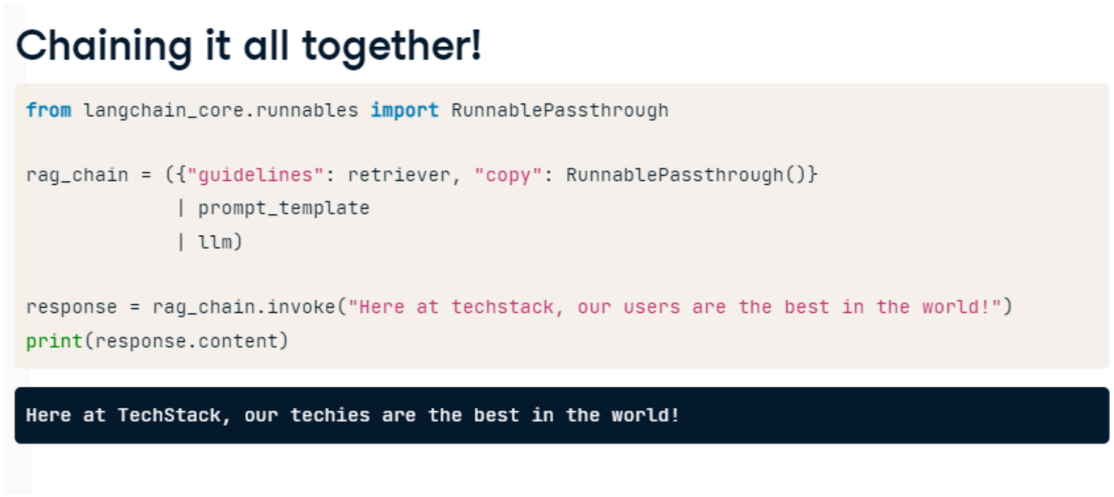
6.2.2 Building a retrieval prompt Template:



Building a prompt template

```
message = """
Review and fix the following TechStack marketing copy with the following guidelines in consideration:

Guidelines:
{guidelines}

Copy:
{copy}

Fixed Copy:
"""

prompt_template = ChatPromptTemplate.from_messages([("human", message)])
```

6.2.3 Creating a RAG Chain:



Chaining it all together!

```
from langchain_core.runnables import RunnablePassthrough

rag_chain = ({"guidelines": retriever, "copy": RunnablePassthrough()}
            | prompt_template
            | llm)

response = rag_chain.invoke("Here at techstack, our users are the best in the world!")
print(response.content)
```

```
Here at TechStack, our techies are the best in the world!
```

Ex:

```
vectorstore = Chroma.from_documents(
    docs,
    embedding=OpenAIEmbeddings(api_key='<OPENAI_API_TOKEN>',
model='text-embedding-3-small'),
    persist_directory=os.getcwd()
)
retriever = vectorstore.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 3}
)
# Create a chain to link retriever, prompt_template, and llm
rag_chain = ({"context": retriever, "question": RunnablePassthrough()}
            | prompt_template
            | llm)
# Invoke the chain
response = rag_chain.invoke("Which popular LLMs were considered in the paper?")
print(response.content)
```

Preparing the documents and vector database

```
loader = PyPDFLoader('rag_vs_fine_tuning.pdf')
data = loader.load()
# Split the document using RecursiveCharacterTextSplitter
splitter = RecursiveCharacterTextSplitter(
    # separators =["\n\n", "\n", " ", ""],
    chunk_size=50,
    chunk_overlap=50
)
docs = splitter.split_documents(data)
# Embed the documents in a persistent Chroma vector database
embedding_function = OpenAIEmbeddings(api_key='<OPENAI_API_TOKEN>',
model='text-embedding-3-small')
vectorstore = Chroma.from_documents(
    docs,
    embedding=embedding_function,
    persist_directory=os.getcwd()
)
# Configure the vector store as a retriever
retriever = vectorstore.as_retriever(
     search_type="similarity",
    search_kwargs={"k:3"}
)
```

Vector databases allow fast retrieval of relevant information based on embedding similarity, which is crucial for large-scale information retrieval applications.

7 LangChain Ecosystem

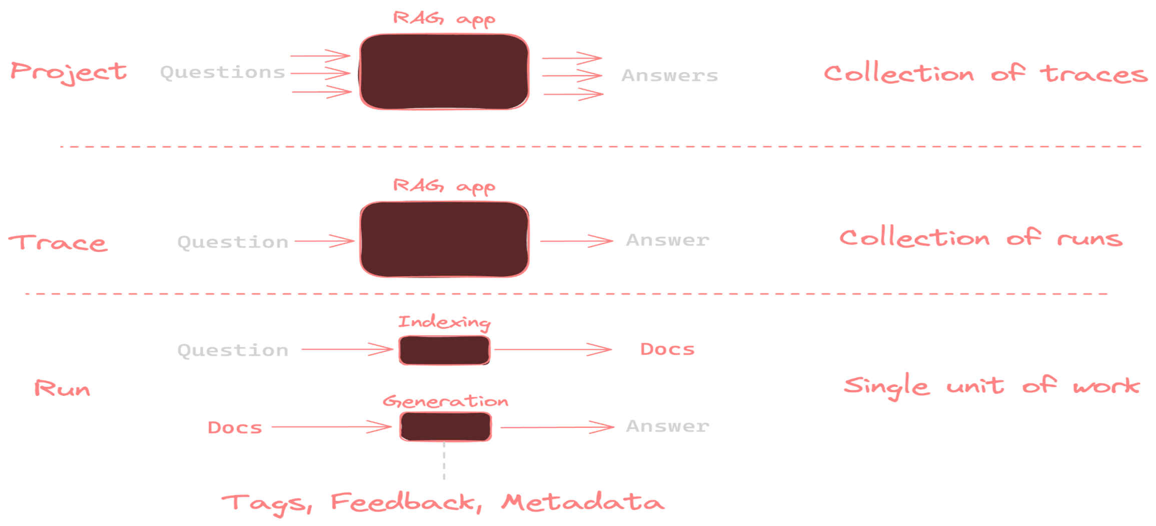
7.1 Langsmith

Langsmith is a debugging and evaluation tool within LangChain, helping developers identify bottlenecks or errors in their workflows.

LangSmith is a platform for building production-grade LLM applications. It allows you to closely monitor and evaluate your application, so you can ship quickly and with confidence. Use of LangChain is not necessary - LangSmith works on its own!

Install LangSmith

Pip install -U langsmith openai



Concepts in langSmith

7.1.1 Observability: This conceptual guide covers topics that are important to understand when logging traces to LangSmith. A Trace is essentially a series of steps that your application takes to go from input to output. Each of these individual steps is represented by a Run. A Project is simply a collection of traces. The following diagram displays these concepts in the context of a simple RAG app, which retrieves documents from an index and generates an answer.

Run A Run is a span representing a single unit of work or operation within your LLM application. This could be anything from single call to an LLM or chain, to a prompt formatting call, to a runnable lambda invocation. If you are familiar with OpenTelemetry, you can think of a run as a span.

Traces A Trace is a collection of runs that are related to a single operation. For example, if you have a user request that triggers a chain, and that chain makes a call to an LLM, then to an output parser, and so on, all of these runs would be part of the same trace. If you are familiar with OpenTelemetry, you can think of a LangSmith trace as a collection of spans. Runs are bound to a trace by a unique trace ID.

Projects A Project is a collection of traces. You can think of a project as a container for all the traces that are related to a single application or service. You can have multiple projects, and each project can have multiple traces.

Feedback Feedback allows you to score an individual run based on certain criteria. Each feedback entry consists of a feedback tag and feedback score, and is bound to a run by a unique run ID. Feedback can currently be continuous or discrete (categorical), and you can reuse feedback tags across different runs within an organization.

Tags Tags are collections of strings that can be attached to runs. They are used to categorize runs and make it easier to search for them in the LangSmith UI. Tags can be used to filter runs in the LangSmith UI, and can be used to group runs together for analysis.

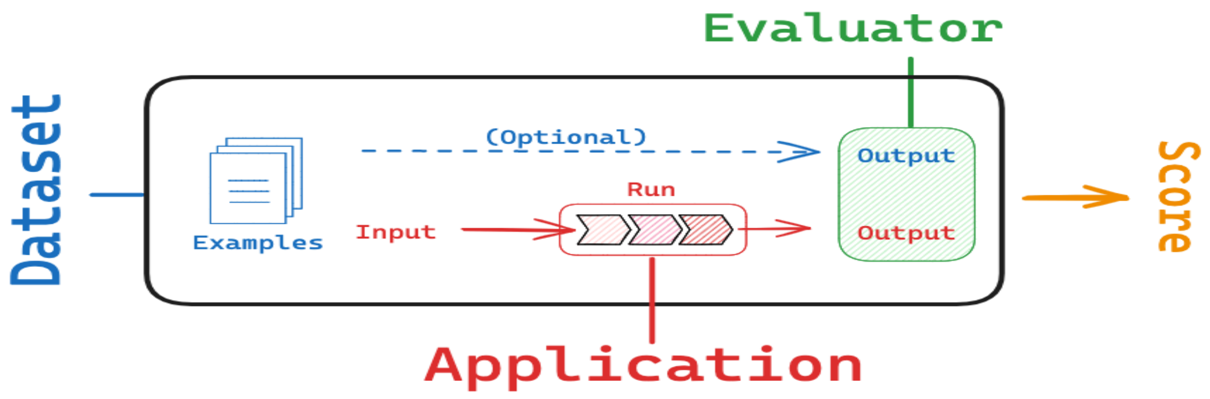
MetaData Metadata is a collection of key-value pairs that can be attached to runs. Metadata can be used to store additional information about a run, such as the version of the application that generated the run, the environment in which the run was generated, or any other information that you want to associate with a run. Similar to tags, you can use metadata to filter runs in the LangSmith UI, and can be used to group runs together for analysis

7.1.2 Evaluation: The pace of AI application development is often rate-limited by high-quality evaluations because there is a paradox of choice. Developers often wonder how to engineer their prompt or which LLM best balances accuracy, latency, and cost. High quality evaluations can help you rapidly answer these types of questions with confidence.

LangSmith allows you to build high-quality evaluations for your AI application. This conceptual guide will give you the foundations to get started. First, let's introduce the core components of LangSmith evaluation:

Dataset: These are the inputs to your application used for conducting evaluations.

Evaluator: An evaluator is a function responsible for scoring your AI application based on the provided dataset.



Evaluation steps:

1. Install langsmith
2. Create an api key
3. Set up environment

    export LANGCHAIN_TRACING_V2=true
    export LANGCHAIN_API_KEY=<your-api-key>

4. Run Evaluation
5. View Experiments UI

Sample RUN Evaluation:

```
from langsmith import evaluate, Client
from langsmith.schemas import Example, Run
# 1. Create and/or select your dataset
client = Client()
dataset                                                                        =
client.clone_public_dataset("https://smith.langchain.com/public/a63525f9-bdf2-
4512-83e3-077dc9417f96/d")

# 2. Define an evaluator
#    For     more     info     on     defining     evaluators,     see:
https://docs.smith.langchain.com/evaluation/how_to_guides/evaluation/evaluate_
llm_application#use-custom-evaluators
def is_concise_enough(root_run: Run, example: Example) -> dict:
                 score    =    len(root_run.outputs["output"])    <    3    *
len(example.outputs["answer"])
    return {"key": "is_concise", "score": int(score)}

# 3. Run an evaluation
evaluate(
    lambda x: x["question"] + " is a good question. I don't know the answer.",
    data=dataset.name,
    evaluators=[is_concise_enough],
    experiment_prefix="my first experiment"
)
```

7.2 LangServe

LangServe allows easy deployment of LangChain applications to production environments, providing scalability and robust performance.

LangServe helps developers deploy LangChain runnables and chains as a REST API.

Features

- Input and Output schemas automatically inferred from your LangChain object, and enforced on every API call, with rich error messages
- API docs page with JSONSchema and Swagger (insert example link)
- Efficient /invoke, /batch and /stream endpoints with support for many concurrent requests on a single server
- /stream_log endpoint for streaming all (or some) intermediate steps from your chain/agent
- new as of 0.0.40, supports /stream_events to make it easier to stream without needing to parse the output of /stream_log.
- Playground page at /playground/ with streaming output and intermediate steps
- Built-in (optional) tracing to LangSmith, just add your API key (see Instructions)
- All built with battle-tested open-source Python libraries like FastAPI, Pydantic, uvloop and asyncio.
- Use the client SDK to call a LangServe server as if it was a Runnable running locally (or call the HTTP API directly)
- LangServe Hub

7.3 LangGraph

LangGraph facilitates the creation of multi-agent knowledge graphs, which are useful for applications requiring multi-agent collaboration or complex decision-making.

LangGraph Platform is a commercial solution for deploying agentic applications to production, built on the open-source LangGraph framework. Here are some common issues that arise in complex deployments, which LangGraph Platform addresses:

- **Streaming support**: LangGraph Server provides multiple streaming modes optimized for various application needs
- **Background runs**: Runs agents asynchronously in the background
- **Support for long running agents**: Infrastructure that can handle long running processes
- **Double texting**: Handle the case where you get two messages from the user before the agent can respond
- **Handle burstiness**: Task queue for ensuring requests are handled consistently without loss, even under heavy loads

Installation

pip install -U langgraph

step-by-step breakdown for implementing a `LangGraph` structure:

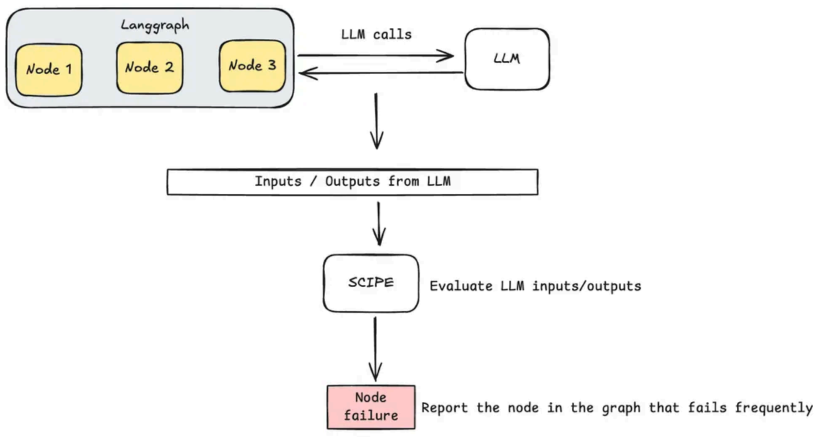
1. Initialize the Model and Tools
2. Initialize the Graph with State
3. Define Graph Nodes
4. Define Entry Point and Graph Edges
5. Compile the Graph
6. Execute the Graph

7.3.1 SCIPE (Systematic Chain Improvement and Problem Evaluation)

https://blog.langchain.dev/scipe-systematic-chain-improvement-and-problem-evaluation/

Demo -

https://colab.research.google.com/drive/1INuL-6cQ-R9z4Clx9L8416ykv6XsRWwg?ref=blog.langchain.dev#scrollTo=33z20rSze8CK


Langgraph
Node 1
Node 2
Node 3
LLM calls
LLM
Inputs / Outputs from LLM
SCIPE
Evaluate LLM inputs/outputs
Node failure
Report the node in the graph that fails frequently