Artificial Intelligence Part 3: RAG, Chains and Agents

Charles S. Givre CISSP

Outline

- Part 1: Introduction and Theory: How Generative Models Work
- Part 2: Prompt Engineering: How to get the most out of Generative Models
- **Part 3: Architecture: How to build AI driven applications**
- Part 4: Red Teaming: How to Attack & Defend AI Applications

Module Outline

- Retrieval Augmented Generation (RAG)
- LangChain
- Building AI Agents

Retrieval Augmented Generation (RAG)

Big Problem with LLMs: Building an LLM to work with proprietary data.

Problems with LLMs

- Training takes a long time and is expensive.
- It is a practical impossibility to keep models up to date with the latest information, especially for areas with fast moving data.
- If you are using public models, you cannot use proprietary data in queries.

1. Train a custom model

2. Fine Tune a Foundational Model

Fine Tuning

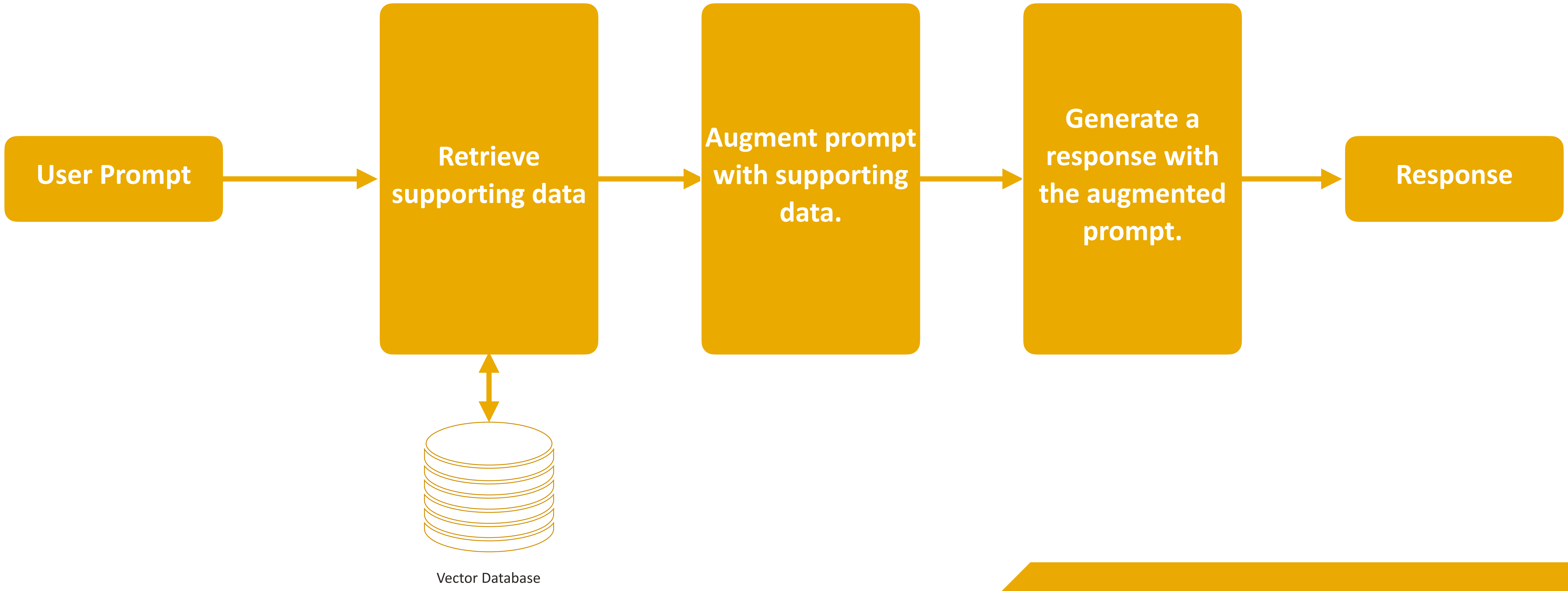
- Fine tuning allows you to tune an existing model for your specific use cases.
- Drawbacks are that it can be time consuming and expensive.
- You will need a decent amount of data for fine tuning.

RAG to the Rescue!

What is RAG?

- RAG stands for Retrieval Augmented Generation and bridges the gap between static models and dynamic systems.
- RAG allows an organization to use AI to generate responses based on either proprietary or current information without having to retrain a model or fine tune it.

How RAG works


User Prompt
Retrieve supporting data
Augment prompt with supporting data.
Generate a response with the augmented prompt.
Response
Vector Database

Components of a RAG workflow

- **Embedding Model**: Converts the prompt into embeddings
- **Vector Database**: A live repository which contains live, up to date information for the model to retrieve. Vector databases are specialty databases for this purpose and quickly match your query vector to the most similar vectors in the database.
- **LLM**: A generative large language model generates the actual response.

Be consistent with your embeddings

RAG in Cyber

- RAG is a great tool for any AI driven applications which involve proprietary or changing information.
- Compliance is an excellent use case.
- Explaining MITRE ATT&CK techniques and comparing them to log info.

Vector Databases

- Vector databases are purpose built databases for retrieval of semantically similar vectors. They implement nearest neighbor algorithms.
- Popular examples now are ChromaDB and Pinecone.
- Many databases can be used as a Vector store including MySQL, MongoDB, Oracle, Postgres and others.

Vector Databases

- We will be using Chroma for our examples and labs.
- Chroma is an OSS vector database which has become quite popular for AI use cases.
- Docs available here: https://docs.trychroma.com/docs/overview/introduction.
- For production use cases, you will need to set up Chroma in the traditional manner, but for our labs, that is not necessary.

Chroma Basic Usage

Chroma organizes data into key/values and is capable of rapid retrieval and efficient storage.

```
import chroma
# Create the client
chroma_client = chromadb.Client()

# Create a document collection if necessary
collection = chroma_client.create_collection(name="collection1")
```

Chroma Basic Usage

```
# You can provide an embedding function or have chroma do it.

collection.add(
    ids=["id1", "id2"],
    documents=[
        "This is a document about pineapple",
        "This is a document about oranges"
    ]
)
```

https://docs.trychroma.com/docs/overview/getting-started

Chroma Basic Usage

```
# Here's where the magic happens...
results = collection.query(
    # Chroma will embed this for you
    query_texts=["I really like citrus"],
       n_results=1 # how many results to return
)
```

```
{'ids': [['id2']],
   'embeddings': None,
   'documents': [['This is a document about oranges']],
   'uris': None,
   'included': ['metadatas', 'documents', 'distances'],
   'data': None,
   'metadatas': [[None]],
   'distances': [[1.0216939449310303]]
}
```

https://docs.trychroma.com/docs/overview/getting-started

Building The Prompt for RAG

- Once you have relevant documents from the vector database, you still have to craft a prompt that includes the contextual documents from the vector database.
- The prompt might look something like this:

```
template = f"""Use the following pieces of context to answer the question at
the end. If you don't know the answer, just say that you don't know. Use three
sentences maximum. Keep the answer as concise as possible.

{context}

Question: {question}

Helpful Answer:"""
```

Remember to keep your prompts short and to the point.

RAG Tips

- RAG queries can have the advantage of providing sourcing information. Seeing sourcing is a good indicator that an AI application is using a RAG pipeline.
- Inputs to RAG pipelines can be poisoned. (More on this later)
- You will have to develop a strategy for chunking documents if you are using large documents.
- RAG can be expensive because it can blow up the input prompt size.

Chunking

- If you have large documents, you will have to divide them up into chunks.
- A good chunking strategy can mean the difference between success and failure for a RAG application.

Implementing a RAG Pipeline

Privacy Still Matters with RAG

- RAG pipelines put the similar documents in the prompt.
- If you are using a public model, those documents are being sent to the model owner in the prompt.

Challenges Building AI Applications

- Models have different SDKs and constructing pipelines.
- As you build out more complex applications, there are an increasing number of steps in that pipeline.
- To build agentic applications, your LLM needs to be able to interact with data and other downstream steps.
- How do we do this?

Introducing LangChain

- LangChain is a framework designed to facilitate building AI applications by abstracting the common components, and having a lot of different implementations of these components.
- This is a VERY fast moving area and there are some other new frameworks such as n8n which are also designed to facilitate building AI driven applications.
- There are commercial products such as Make.com, String.com, Zapier and others.

LangChain Modules

- LangChain features 6 top-level abstractions:
  - **Model I/O**: Facilitates inputs/ outputs to the model
  - **Retrieval**: Retrieving relevant text from a vector database.
  - **Chains**: Enable the contraction of sequences of LLM operations.

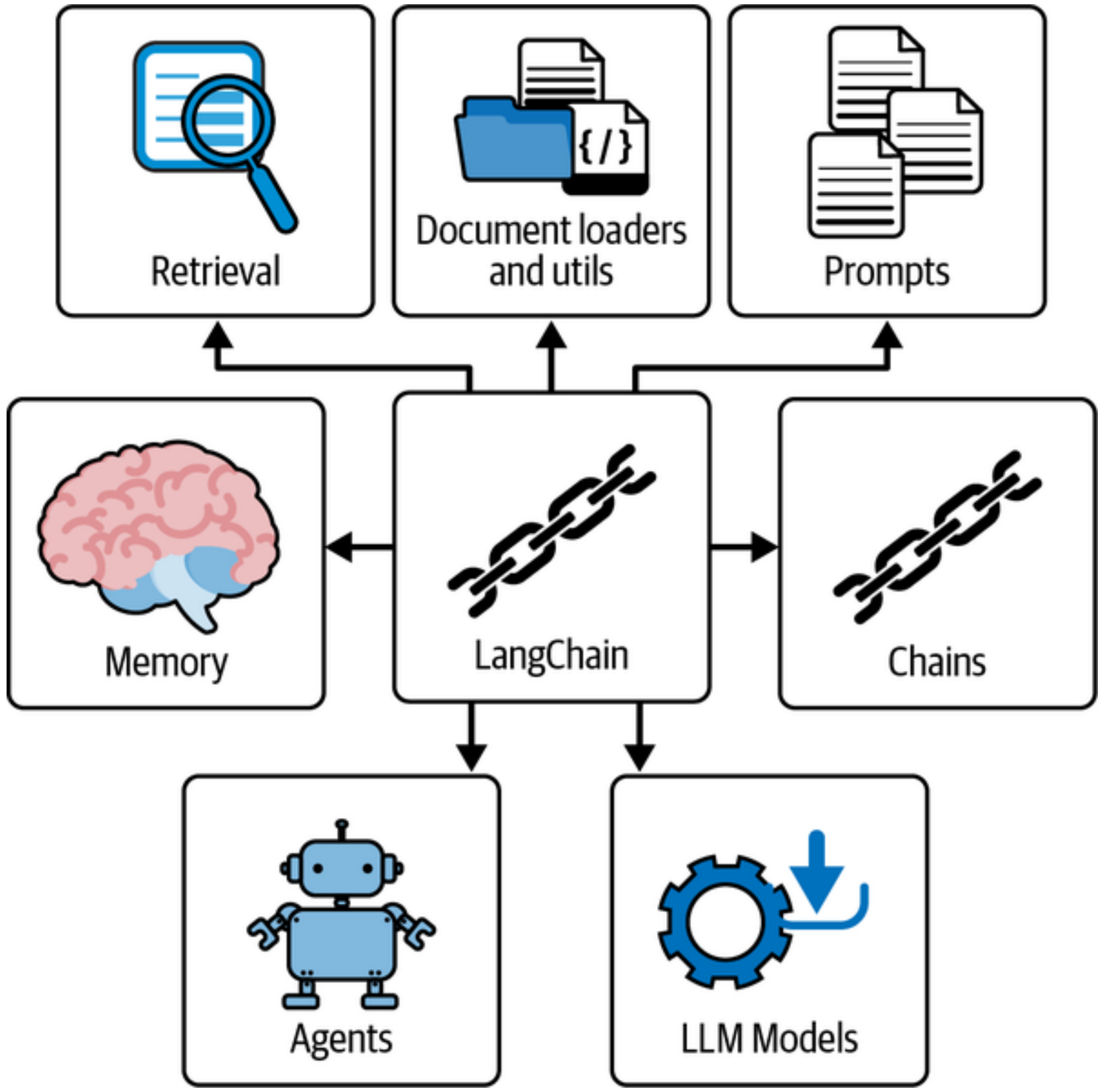


Image from *Prompt Engineering for Generative AI* by James Phoenix and Mike Taylor

LangChain Modules

- **Agents:** Allow chains to make decisions on which tools to use based on high level instructions.
- **Memory**: Saves the state of the application between LLM interactions.
- **Callbacks:** Used to run additional code on an event happening
- **Models:** Abstracts a LLM

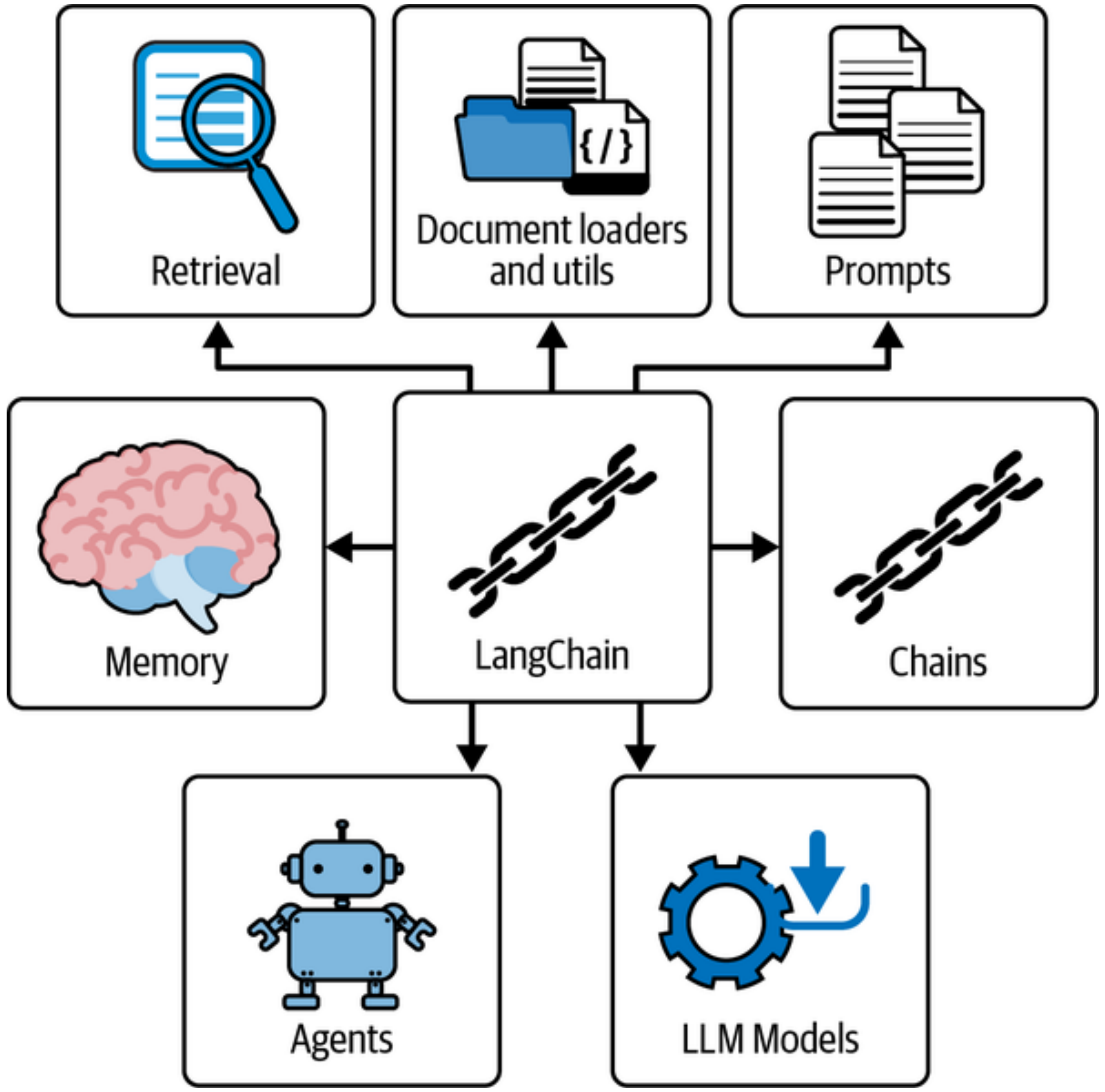


Image from *Prompt Engineering for Generative AI* by James Phoenix and Mike Taylor

LangChain Messages

- There are three message abstractions in LangChain:
  - SystemMessage: This is the message which tells the model how it should behave
  - HumanMessage: This is information coming from the user that the AI responds to
  - AIMessage: The responses from the AI.

LangChain Example 1

Note: All the LangChain examples can be found in the `ai_examples` folder in the class repo.

```
from langchain_openai.chat_models import ChatOpenAI
from langchain.schema import HumanMessage, SystemMessage

chat = ChatOpenAI(temperature=0.75,
                  model="gpt-3.5-turbo",
                  api_key=os.getenv("OPENAI_KEY"))


messages = [
    SystemMessage(content="You are a helpful assistant that generates ASCII art.  You must only
respond with a witty greeting and a cheerful ASCII art based on the user's input."),
    HumanMessage(content="Hello, how are you?  It sure is beautiful outside!"),
]

response = chat.invoke(messages)
print(response)
```

LangChain Example 1

```
{'additional_kwargs': {'refusal': None},
 'content': 'Hello there! 🌞\n'
            '```\n'
            '     / \\ \n'
            '    / _ \\\n'
            '   | / \\ |\n'
            '   ||   || _______\n'
            '   ||   || |\\     \\\n'
            '   ||   || ||\\     \\\n'
            '   ||   || || \\    |\n'
            '   ||   || ||  \\__/\n'
            '   ||   || ||   ||\n'
            '    \\\\_/ \\_/ \\_//\n'
            '   /   _     _   \\\n'
            '  /               \\\n'
            '  |    O     O    |\n'
            '  |   \\  ___  /   |      \n'
            ' /    \\ \\_/ /     \\\n'
            '/  -----  |  --\\    \\\n'
            '|     \\__/|\\__/ \\   |\n'
            '\\       |_|_|      /\n'
            ' \\_____      _____/\n'
            '      \\     /\n'
            '       |     |\n'
            '```\n'
            'Enjoy the beautiful day!',
 'example': False,
 'id': 'run--b34ed13b-632c-4fe9-a2f1-d9a50ce8f6b5-0',
 'invalid_tool_calls': [],
 'name': None,
 'response_metadata': {'finish_reason': 'stop',
                       'id': 'chatcmpl-Bx22zJjyCJoj3jV6saD20yPqU0kxn',
                       'logprobs': None,
```

```
'usage_metadata': {'input_token_details': {'audio': 0,
'cache_read': 0},
                    'input_tokens': 55,
                    'output_token_details': {'audio': 0,
'reasoning': 0},
                    'output_tokens': 159,
                    'total_tokens': 214}}
```

LangChain and Streaming

- Generative models generate one token at a a time. To reduce latency, you can stream the results from the model instead of waiting for the entire completion.
- LangChain supports this, but it adds some complexity to your code and is beyond the scope of this class.
- You can also send batches (in parallel) to LangChain if you want to generate multiple responses.

LangChain Prompt Templates

- LangChain has a prompt template object which are templates that accept parameters and construct a prompt for the LLM.
- LangChain prompt templates allow you to easily validate user inputs, combine prompts together and easily create few-shot examples from data.

```
system_template = "Your name is {ai_name}"
system_prompt = SystemMessageTemplate.from_template(template)
chat_prompt = ChatPromptTemplate.from_messages([system_prompt])
chain = chat_prompt | model
result = chain.invoke({ai_name: "Jill"})

# Take a look at risky_sql_queries.py
```

LangChain Expression Language (LCEL)

- LCEL is a declarative language for building complex runnable AI applications
- It's the glue that puts all the pieces together.
- LCEL offers benefits such as optimized parallel execution, asynchronous support, and simplified streaming.
- LCEL runnables are logged automatically and feature a standard API for ease of development.

https://python.langchain.com/docs/concepts/lcel/

Single LLM Calls do not need LCEL

LCEL Runnable Sequences

- LCEL has two types of runnable objects: RunnableSequence and RunnableParallel
- Both can be run synchronously or asynchronously.
- You can call a chain using the `invoke()` method.
- LCEL overloaded the | operator so any runnable objects can be chained together with the | as shown below:

```
chain = step1 | step2 | step3
```

LangChain Output Parsers

- LangChain has a collection of Output Parsers which can extract structured data from LLM responses. Currently available parsers are:
    - List Parser
    - Datetime Parser
    - Auto-fixing Parser
    - Pydantic Parser (Predefined JSON Objects)
    - Retry Parser
    - Structured Output Parser
    - XML Parser
    - More…
- LangChain Parsers can generate output instructions for the LLM!

Tangent: Pydantic

- Pydantic is a data validation library which you can use to define schemata for complex JSON objects.
- The easy way to create Pydantic objects are tools like https://jsontopydantic.com/

Using an Output Formatter

```
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.prompts import SystemMessagePromptTemplate, ChatPromptTemplate
from langchain_core.output_parsers import PydanticOutputParser
from pydantic import BaseModel

class SQL_Query(BaseModel):
    query: str
    is_malicious: bool
    table_count: int

json_output_parser = PydanticOutputParser(pydantic_object=SQL_Query)
json_output_parser.get_format_instructions()
```

Using an Output Formatter

The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {"properties": {"foo": {"title": "Foo", "description": "a list of strings", "type": "array", "items": {"type": "string"}}}, "required": ["foo"]}

the object {"foo": ["bar", "baz"]} is a well-formatted instance of the schema. The object {"properties": {"foo": ["bar", "baz"]}} is not well-formatted.

Here is the output schema:

```
{"properties": {"query": {"title": "Query", "type": "string"}, "is_malicious": {"title": "Is Malicious", "type": "boolean"}, "table_count": {"title": "Table Count", "type": "integer"}}, "required": ["query", "is_malicious", "table_count"]}
```

Using an Output Formatter

```
chain = chat_prompt | model | json_output_formatter
result = chain.invoke({"table": "users",
                       "columns": [...],
                       "format_instructions": json_output_parser.get_format_instructions()
                       },
                      )
```

```
query='SELECT * FROM users WHERE id = 1' is_malicious=True table_count=1
```

The output is parsed into a useable object!

Questions?

Using AI to Call Functions

DANGER

- Calling functions from AI is inherently risky, especially if the AI accepts input from untrusted users.
- When implementing features that call functions, always ask yourself what could happen if this "goes rogue"?
- Remember that the AI can be somewhat unpredictable, so be sure to include safeguards.
- **This is especially true if your AI bot is connected to internal data sources.**

Function Calling Workflow

Future Learning

- LangChain Evaluators: Useful for comparing models.
- Advanced Pydantic usage
- Model fine tuning