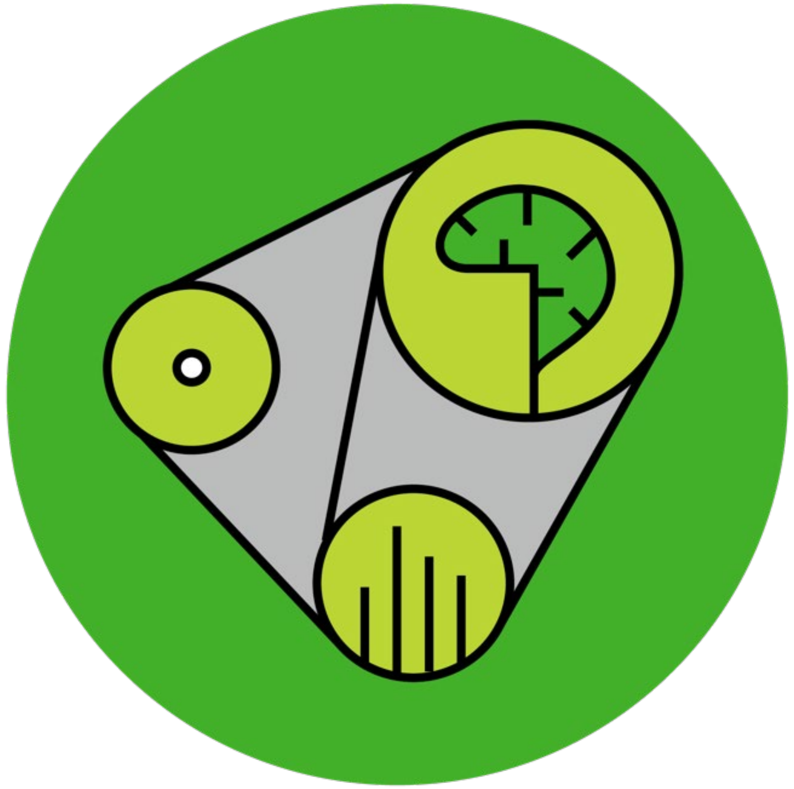
Deloitte.



AI Guild | GenAI Practicum

March 2025

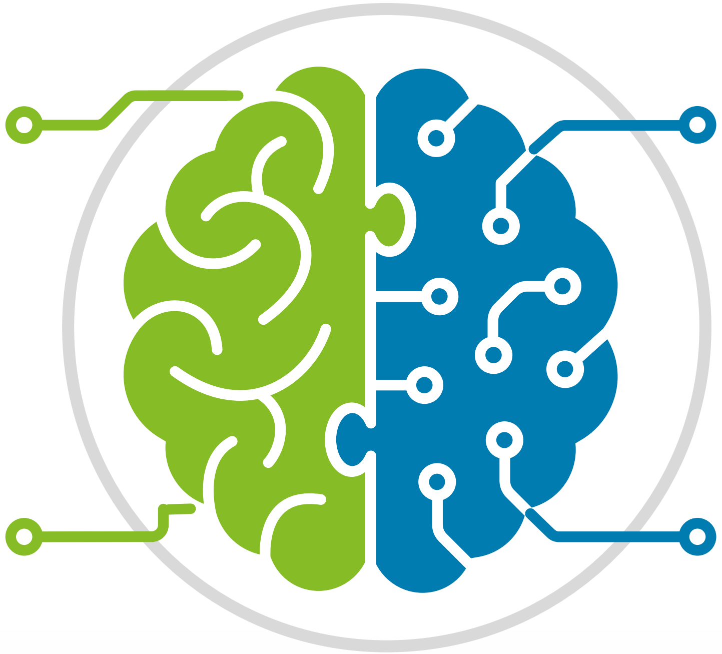
Week Agenda

Topic	Content
01 Section 1 - Langchain	• What is Langchain • What are its Core Features and Modules
02 Modules Part 1	• I/O, Model • Retriever, Vector, RAG • Chains
03 Modules Part 2	• Memory • Agents • Callback handler
03 Build an App	• Put together an LLM, Langchain, and RAG in an App

Section 1 - Langchain

Learning objectives

By the end of session, you should be able to



01 Understand LangChain

02 Its Core Features and Modules

03 Build Simple GenAI Apps Using Langchain

04 Use Cases Where The Different Modules Can Help

LangChain

Framework for developing applications powered by LLMs, in python and javascript/ typescript that are:

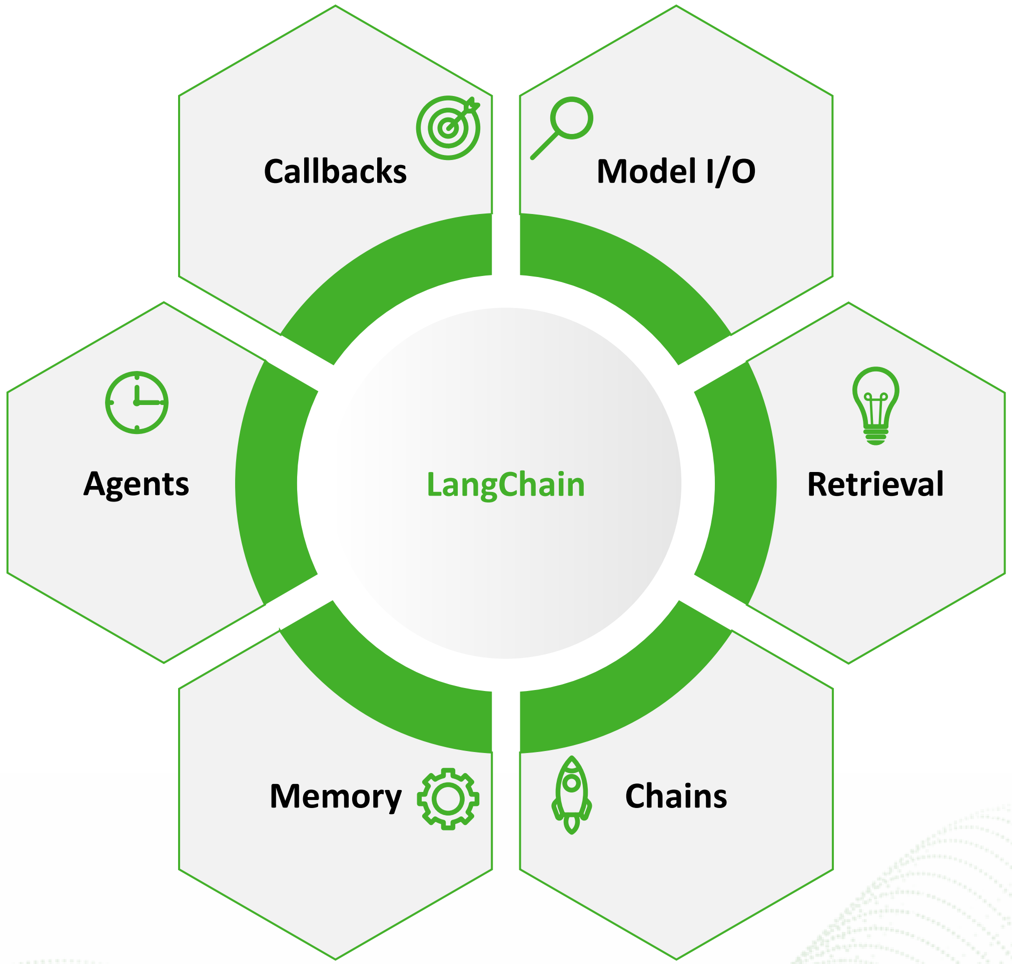
- **Data-aware:** connect a language model to other sources of data
- **Agentic:** allow a language model to interact with its environment

The main value props of LangChain are:

- **Components:** abstractions for working with LLMs, along with a collection of implementations for each abstraction. Components are modular and easy-to-use, whether you are using the rest of the LangChain framework or not
- **Off-the-shelf chains:** a structured assembly of components for accomplishing specific higher-level tasks. Off-the-shelf chains make it easy to get started. For more complex applications and nuanced use-cases, components make it easy to customize existing chains or build new ones.

🦜🔗 LangChain

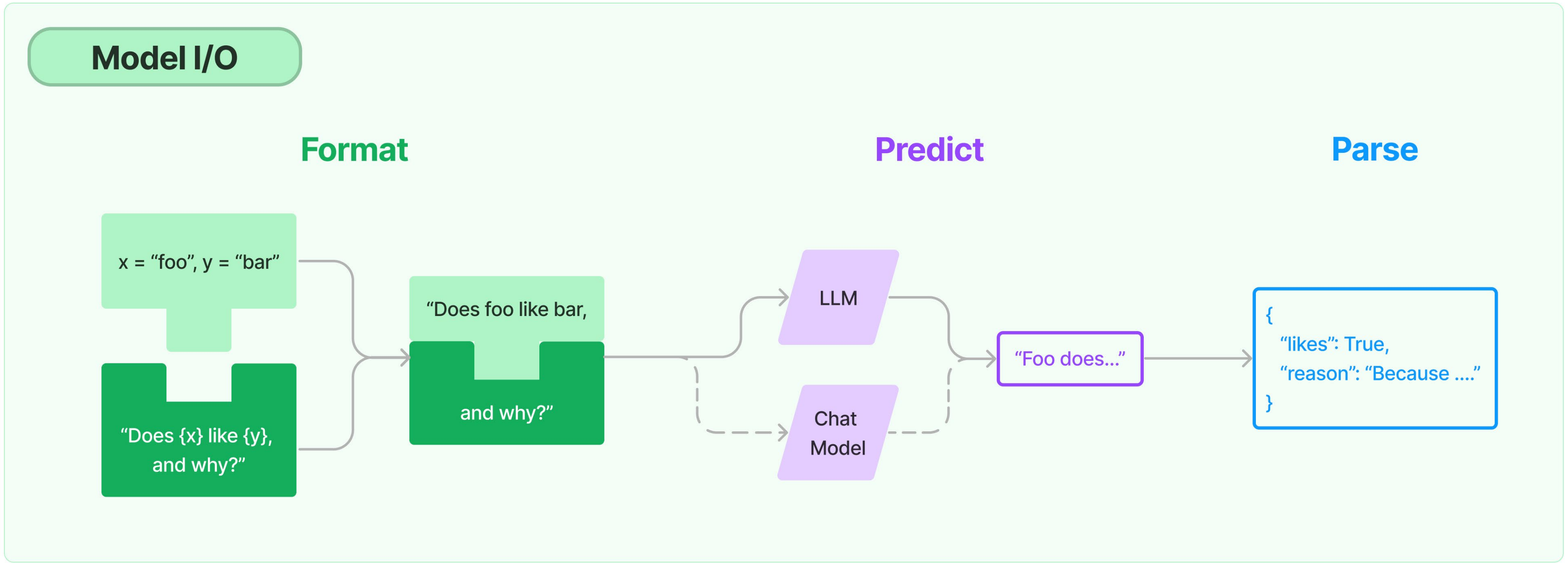
Modules


LangChain
Callbacks
Model I/O
Agents
Retrieval
Memory
Chains

Model I/O

The core element of any LLM application is...the model. LangChain gives you the building blocks to interface with any LLM.

- **Prompts:** Templatize, dynamically select, and manage model inputs
- **Language models:** Make calls to language models through common interfaces
- **Output parsers:** Extract information from model outputs



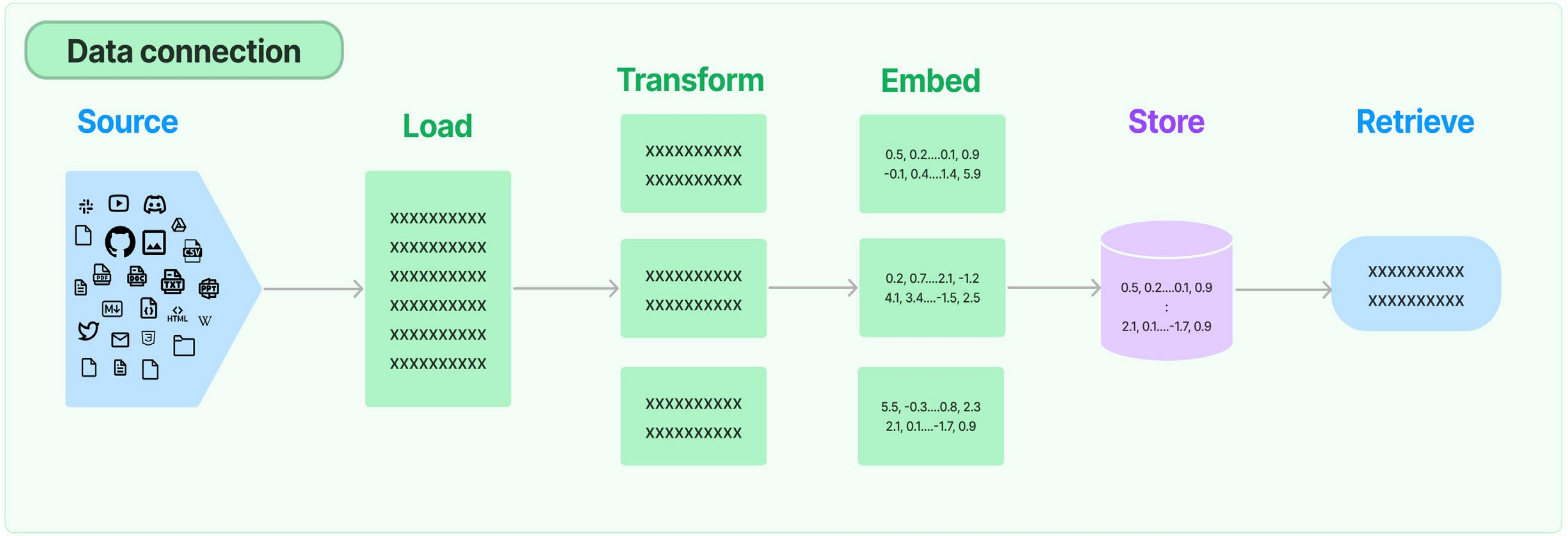
Model I/O
Format
Predict
Parse
x = "foo", y = "bar"
"Does {x} like {y}, and why?"
"Does foo like bar, and why?"
LLM
Chat Model
"Foo does..."
{ "likes": True, "reason": "Because ...." }

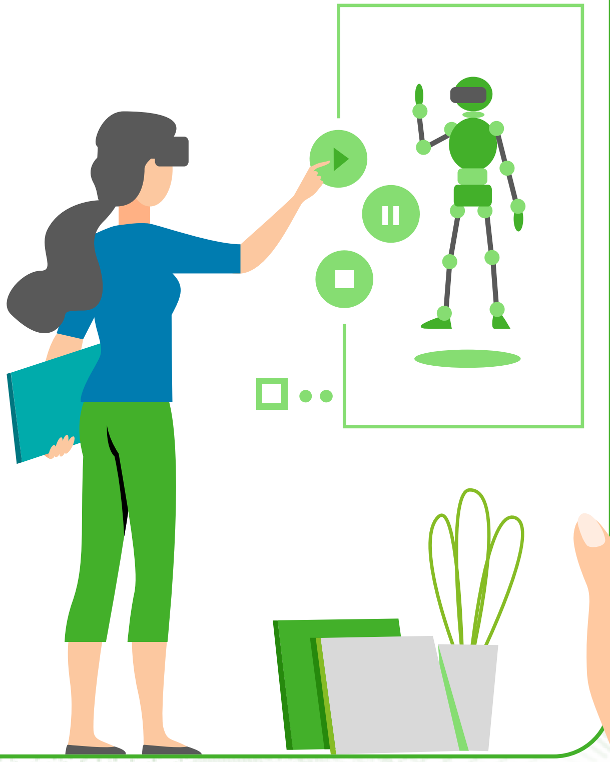
Labs Pre Check

- Please make sure all have forked repo(https://github.com/Deloitte-US-DEP-Training/genai-practicum-track1-labs) and is in sync with latest and greatest we have in Session3 folder.
- Please note Lab-1 Talking to LLMs where we have a module to interact with model is something we all will try to run, if able to talk to LLM remaining labs are appending to that, rest of it because of time constraints will be readouts, please take your time to run and explore post session as you will have access.

Retrieval

Many LLM applications require user-specific data that is not part of the model's training set. The primary way of accomplishing this is through **Retrieval Augmented Generation** (RAG). In this process, external data is retrieved and then passed to the LLM when doing the generation step. LangChain provides all the building blocks for RAG applications – from simple to complex.





Retrieval contd…

Document loaders

- Load documents from many different sources.
- 100 different document loaders as well as integrations with other major providers in the space, like AirByte and Unstructured.
- Load all types of documents (HTML, PDF, code) from all types of locations (private s3 buckets, public websites).

Document transformers

- Fetching only the relevant parts of documents. This involves several transformation steps in order to best prepare the documents for retrieval. One of the primary ones here is splitting (or chunking) a large document into smaller chunks.
- Several different algorithms for doing this, as well as logic optimized for specific document types (code, markdown, etc).

Retrieval contd…

Text embedding models

- Creating embeddings for documents. Embeddings capture the semantic meaning of the text, allowing you to quickly and efficiently find other pieces of text that are similar.
- 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.
- Standard interface, allowing you to easily swap between models.

Vector stores

- With the rise of embeddings, there has emerged a need for databases to support efficient storage and searching of these embeddings.
- Integrations with over 50 different vector stores, from open-source local ones to cloud-hosted proprietary ones, allowing you to choose the one best suited for your needs.
- Standard interface, allowing you to easily swap between vector stores.

Retrievers

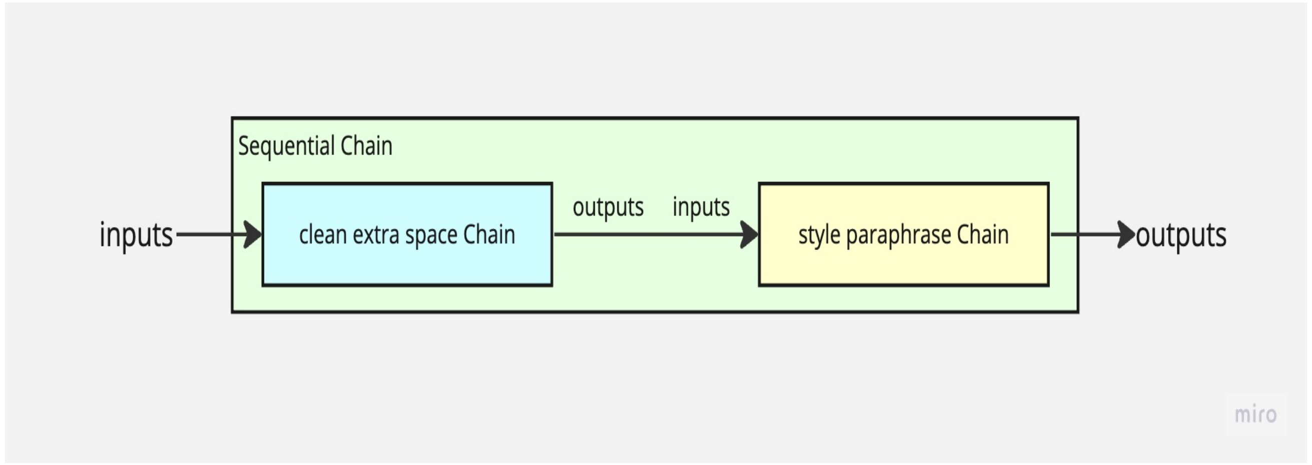
Different retrieval from basic methods that are easy to get started - namely simple semantic search. A collection of algorithms on top of this to increase performance. These include:

- **Parent Document Retriever:** This allows you to create multiple embeddings per parent document, allowing you to look up smaller chunks but return larger context.
- **Self Query Retriever:** User questions often contain a reference to something that isn't just semantic but rather expresses some logic that can best be represented as a metadata filter. Self-query allows you to parse out the semantic part of a query from other metadata filters present in the query.
- **Ensemble Retriever:** Sometimes you may want to retrieve documents from multiple different sources, or using multiple different algorithms. The ensemble retriever allows you to easily do this.

Chains

Using an LLM in isolation is fine for simple applications, but more complex applications require chaining LLMs – either with each other or with other components. LangChain provides the Chain interface for such "chained" applications. A chain can be defined as a sequence of calls to components, which can include other chains

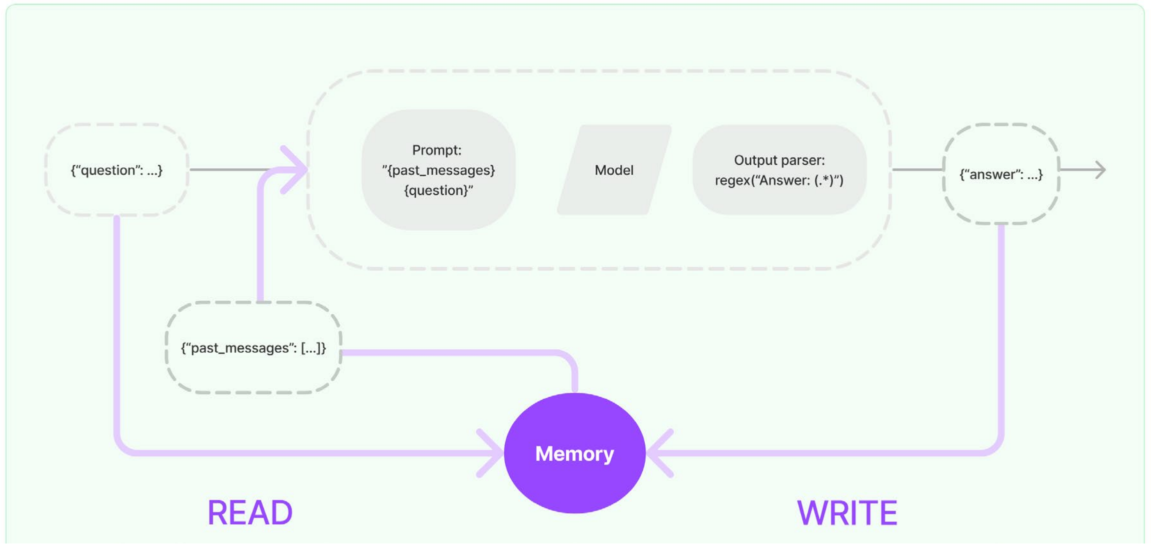
- **Generic Chains:** used for more generic use cases like chaining prompt, with query inputs to LLM, sequential chaining etc
- **Utility Chains:** summarization, bash, conversation, Q&A etc, these chains perform specialized tasks



Memory

At bare minimum, a conversational system should be able to access some window of past messages directly. This ability to store information about past interactions is "memory” here. A memory system needs to support two basic actions: reading and writing. Recall that every chain defines some core execution logic that expects certain inputs. Some of these inputs come directly from the user, but some of these inputs can come from memory. A chain will interact with its memory system twice in a given run.

- AFTER receiving the initial user inputs but BEFORE executing the core logic, a chain will **READ** from its memory system and augment the user inputs.
- AFTER executing the core logic but BEFORE returning the answer, a chain will **WRITE** the inputs and outputs of the current run to memory, so that they can be referred to in future runs.

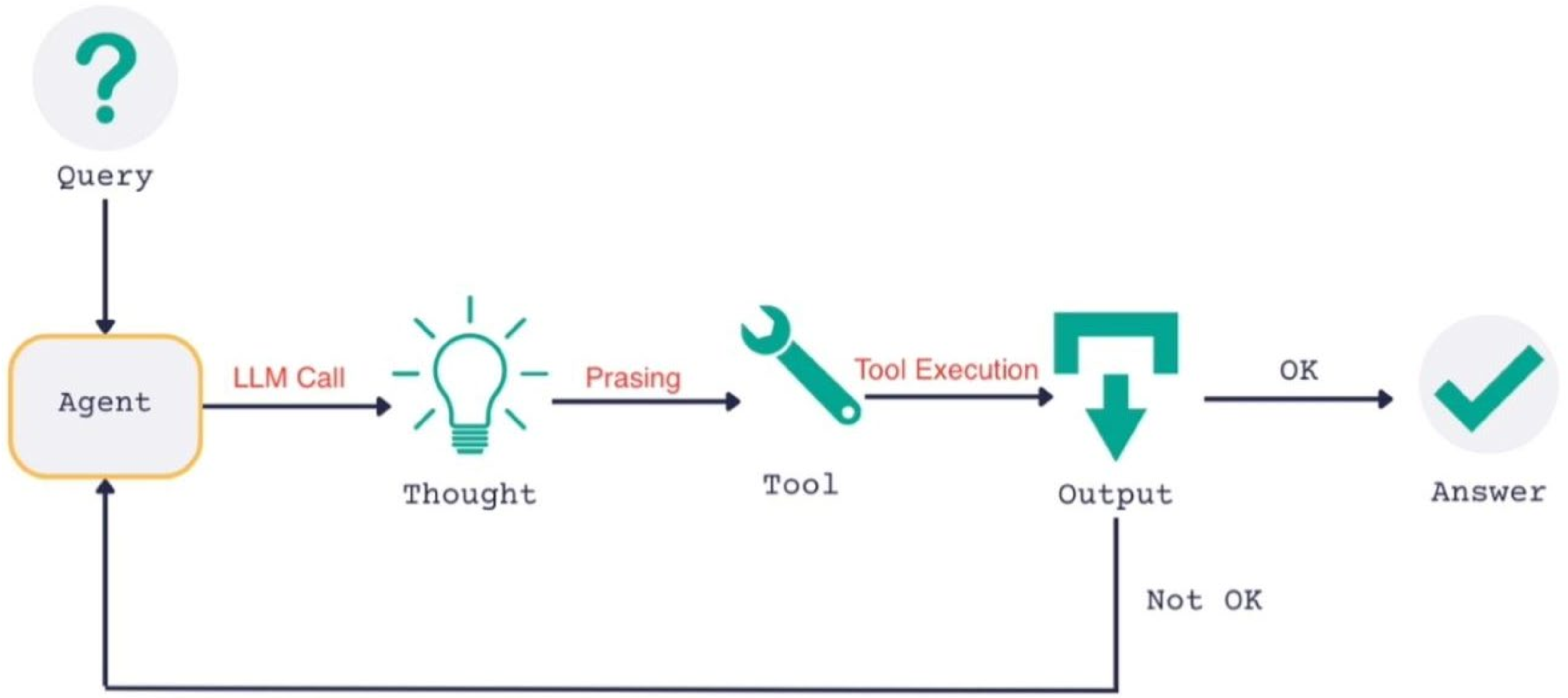


Agents

The core idea of agents is to use an LLM to choose a sequence of actions to take. In chains, a sequence of actions is hardcoded (in code). In agents, a language model is used as a reasoning engine to determine which actions to take and in which order.

- **Agent:** responsible for deciding what step to take next. This is powered by a language model and a prompt which can include
  - The personality of the agent (useful for having it respond in a certain way)
  - Background context for the agent (useful for giving it more context on the types of tasks it's being asked to do)
  - Prompting strategies to invoke better reasoning (the most famous/widely used being ReAct)
- **Tools:** functions that an agent calls. There are two important considerations here:
  - Giving the agent access to the right tools
  - Describing the tools in a way that is most helpful to the agent
- **Toolkit:** Often the set of tools an agent has access to is more important than a single tool. For this LangChain provides the concept of toolkits – groups of tools needed to accomplish specific objectives.
- **Agent Executor:** the runtime for an agent. This is what actually calls the agent and executes the actions it chooses.

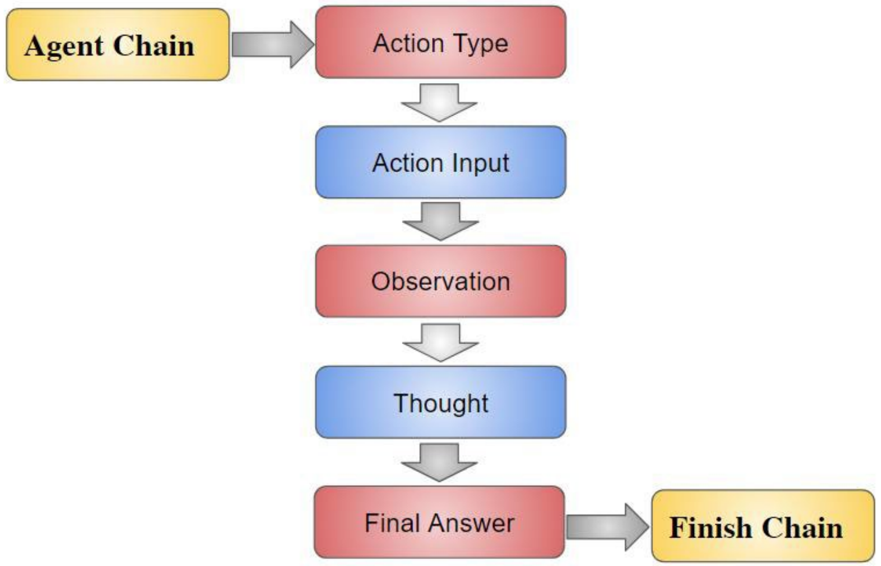
Agents


Query
Agent
LLM Call
Thought
Prasing
Tool
Tool Execution
Output
OK
Answer
Not OK

Agents vs Chains

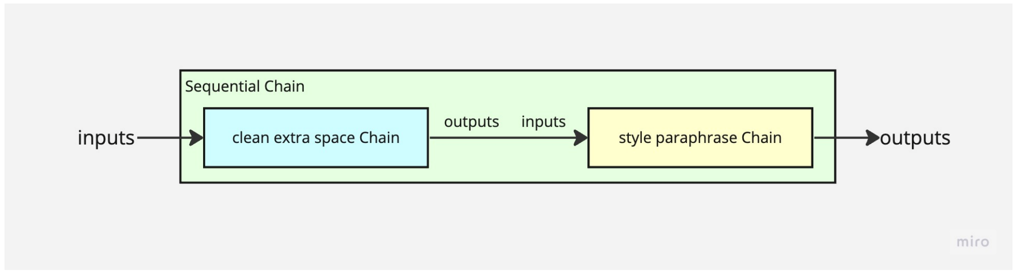
Agents

- Agents use a language model to decide what actions to take. Agents would use a language model to generate their responses based on the user's input and the available tools.
- Agents allow a step-by-step thought process.



Chains

- Chains have a fixed sequence of actions defined by the developer.
- Chain defines an immediate input/output process.



Callbacks

Callbacks system allows you to hook into the various stages of your LLM application. This is useful for logging, monitoring, streaming, and other tasks. You can subscribe to these events by using the callbacks argument available throughout the API. The callbacks argument is available on most objects throughout the API (Chains, Models, Tools, Agents, etc.) in two different forms:

Constructor callbacks

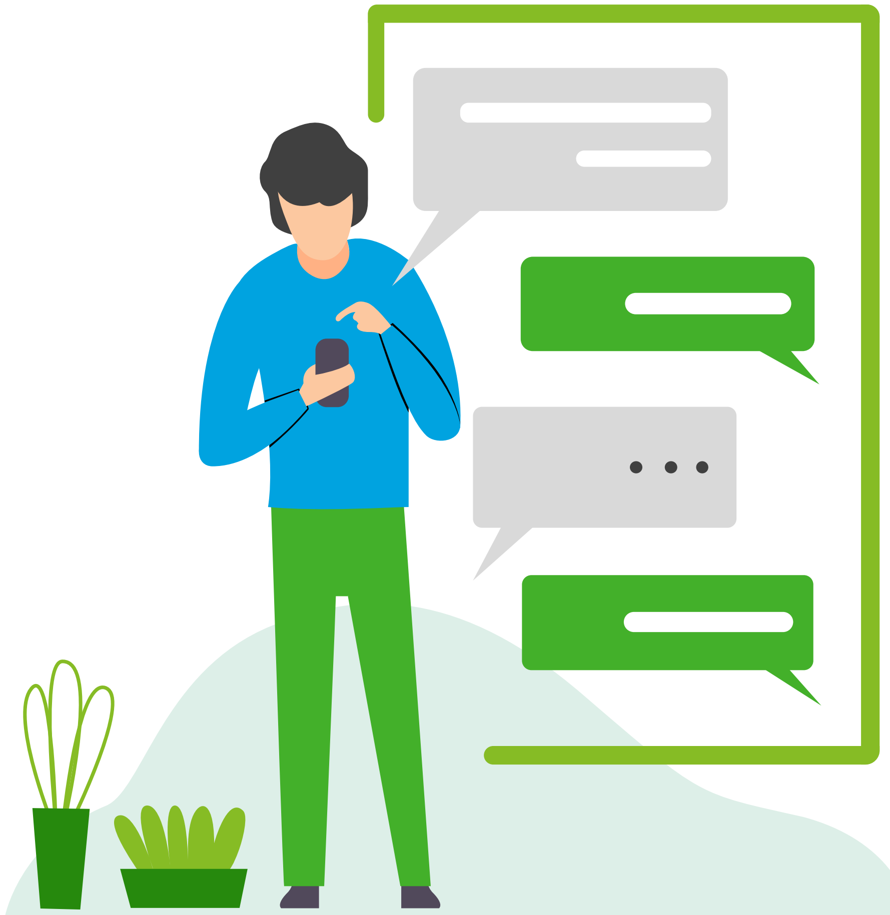
- In the constructor, e.g. LLMChain (callbacks=[handler], tags=['a-tag']), which will be used for all calls made on that object, and will be scoped to that object only, e.g. if you pass a handler to the LLMChain constructor, will not be used by the Model attached to that chain..
- For use cases such as logging, monitoring, etc., which are not specific to a single request, but rather to the entire chain. For example, if you want to log all the requests made to an LLMChain, you would pass a handler to the constructor.

Request callbacks

- In the run()/apply() methods used for issuing a request, e.g. chain.run (input, callbacks=[handler]), which will be used for that specific request only, and all sub-requests that it contains (e.g. a call to an LLMChain triggers a call to a Model, which uses the same handler passed in the call() method)
- For use cases such as streaming, where you want to stream the output of a single request to a specific websocket connection, or other similar use cases. For example, if you want to stream the output of a single request to a websocket, you would pass a handler to the call() method

Callbacks contd…

The verbose argument is available on most objects throughout the API (Chains, Models, Tools, Agents, etc.) as a constructor argument, e.g. LLMChain(verbose=True), and it is equivalent to passing a ConsoleCallbackHandler to the callbacks argument of that object and all child objects. This is useful for debugging, as it will log all events to the console.



LangChain Expression Language

LangChain Expression Language, or LCEL, is a declarative way to easily compose chains together. LCEL was designed from day 1 to support putting prototypes in production, with no code changes, from the simplest “prompt + LLM” chain to the most complex chains (we’ve seen folks successfully run LCEL chains with 100s of steps in production). To highlight a few of the reasons you might want to use LCEL:

Streaming support

When you build your chains with LCEL you get the best possible time-to-first-token (time elapsed until the first chunk of output comes out). For some chains this means e.g., we stream tokens straight from an LLM to a streaming output parser, and you get back parsed, incremental chunks of output at the same rate as the LLM provider outputs the raw tokens.

Async support

Any chain built with LCEL can be called both with the synchronous API (e.g., in your Jupyter notebook while prototyping) as well as with the asynchronous API (e.g., in a LangServe server). This enables using the same code for prototypes and in production, with great performance, and the ability to handle many concurrent requests in the same server.

Optimized parallel execution

Whenever your LCEL chains have steps that can be executed in parallel (eg if you fetch documents from multiple retrievers) we automatically do it, both in the sync and the async interfaces, for the smallest possible latency.

LangChain Expression Language

Seamless LangSmith tracing integration: As your chains get more and more complex, it becomes increasingly important to understand what exactly is happening at every step. With LCEL, all steps are automatically logged to LangSmith for maximum observability and debuggability.

Seamless LangServe deployment integration: Any chain created with LCEL can be easily deployed using LangServe.

Retries and fallbacks

Configure retries and fallbacks for any part of your LCEL chain. This is a great way to make your chains more reliable at scale. We're currently working on adding streaming support for retries/fallbacks, so you can get the added reliability without any latency cost.

Access intermediate results

For more complex chains it's often very useful to access the results of intermediate steps even before the final output is produced. This can be used let end-users know something is happening, or even just to debug your chain. You can stream intermediate results, and it's available on every LangServe server.

Input and output schemas

Input and output schemas give every LCEL chain Pydantic and JSONSchema schemas inferred from the structure of your chain. This can be used for validation of inputs and outputs, and is an integral part of LangServe.

App to query multiple data sources

How Langchain can help us to query multiple data sources like PDF, then SQL database, then TXT file. How Langchain understands where to extract meaningful information, how Langchain do the right selection of data sources.

Time to build a sample app, where an employee salary data is given in a csv and employee profile information in a pdf, then create a GenAI based chatbot that can aggregate information form both sources and give the final answer.

It’s Time to Build an App

Q & A

SessionTakeaways

Share key Takeaways

01 Understanding of LangChain and its core features and different modules

02 LangChain is a framework that can help to accelerate the development of GenAI applications built on top of LLMs

03 LangChain has a vast number of OOTB integrations with most of the components needed in a GenAI application ecosystem

04 LangChain modules are built with extensibility in mind so anything you would like to add to the existing integrations across modules can be easily developed

Session Summary

Share key Takeaways

05 RAG is a good strategy for large document storage and retrieval

06 Agents can be used to allow LLM driven interfaces to execute actions

07 Applications typically utilize a variety of tools, including the LLM, LangChain, RAG, and Agents to get the job done.

Appendix: LlamaIndex

LlamaIndex (formerly GPT Index) is a data framework for LLM applications to ingest, structure, and access private or domain-specific data.

Why LlamaIndex?

At their core, LLMs offer a natural language interface between humans and inferred data. Widely available **models come pre-trained on huge amounts of publicly available data, from Wikipedia and mailing lists to textbooks and source code**. Applications built on top of LLMs often require **augmenting these models with private or domain-specific data.** Unfortunately, that data can be distributed across siloed applications and data stores. It's behind APIs, in SQL databases, or trapped in PDFs and slide decks. That's where LlamaIndex comes in.

How can LlamaIndex help?

LlamaIndex provides the following tools:

Data connectors ingest your existing data from their native source and format. These could be APIs, PDFs, SQL, and (much) more.

Data indexes structure your data in intermediate representations that are easy and performant for LLMs to consume.

Engines provide natural language access to your data. For example:

- **Query engines** are powerful retrieval interfaces for knowledge-augmented output.
- **Chat engines** are conversational interfaces for multi-message, "back and forth" interactions with your data.

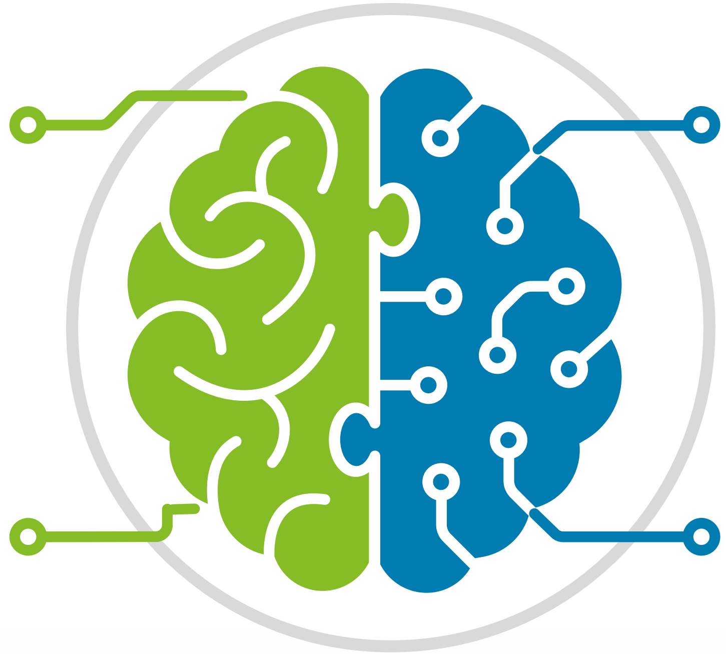
Data agents are LLM-powered knowledge workers augmented by tools, from simple helper functions to API integrations and more.

Application integrations tie LlamaIndex back into the rest of your ecosystem. This could be LangChain, Flask, Docker, ChatGPT, or… anything else!

Section 2 – Retrieval Augmented Generation (RAG)

Learning objectives

By the end of session, you should be able to



01 Understand Vectors and Embeddings

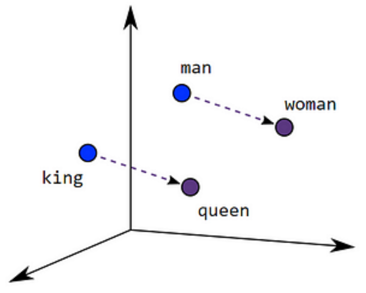
02 Understand Vector Stores

03 Understand Agents

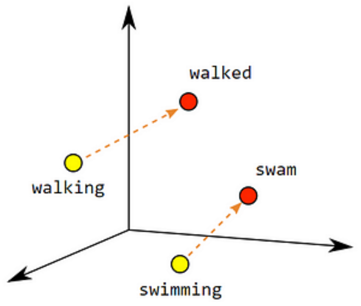
04 Putting it all together in an App

Embeddings

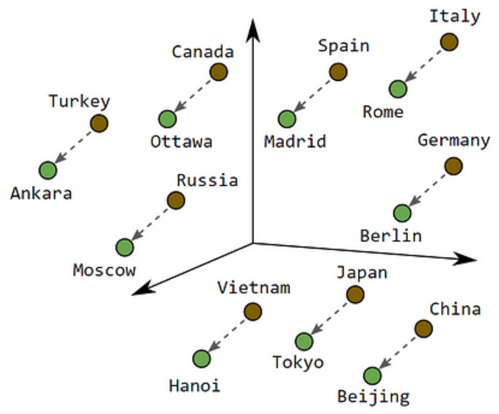
- An embedding is a numerical representation of a piece of information that captures its semantic meaning.
- Embeddings can be created of text, images, documents, audio, etc.



Male-Female



Verb Tense



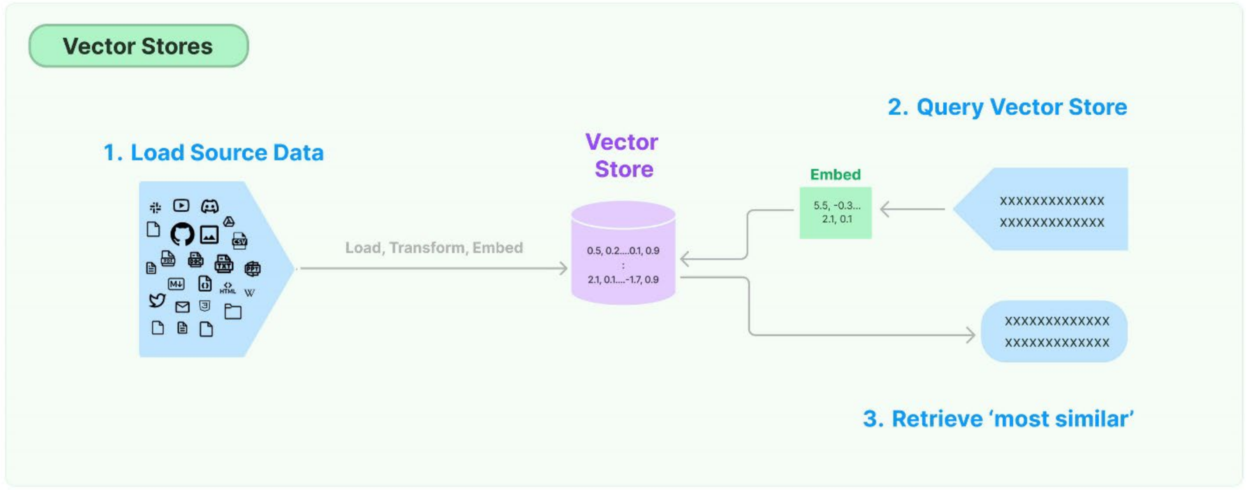
Country-Capital

Vector Stores

- Databases that store embeddings for textual data.

- The vector store not only keeps these vectors together in a vector space, but also quickly and efficiently finds and places similar vectors or identical vectors near one another.

- Vector stores are useful for applications that require fast responses, like chat bots or voice assistants.



Examples of Vector Stores: Pinecone, Chroma, Lance, FAISS

Building vector stores with Langchain

Raw Documents

- Documents with text contents are loaded.

Chunking

- The text content is splitted into chunks of text or a set of characters.

Embedding creation

- Embeddings are created for each chunk.

Loading of Documents into Vector Store

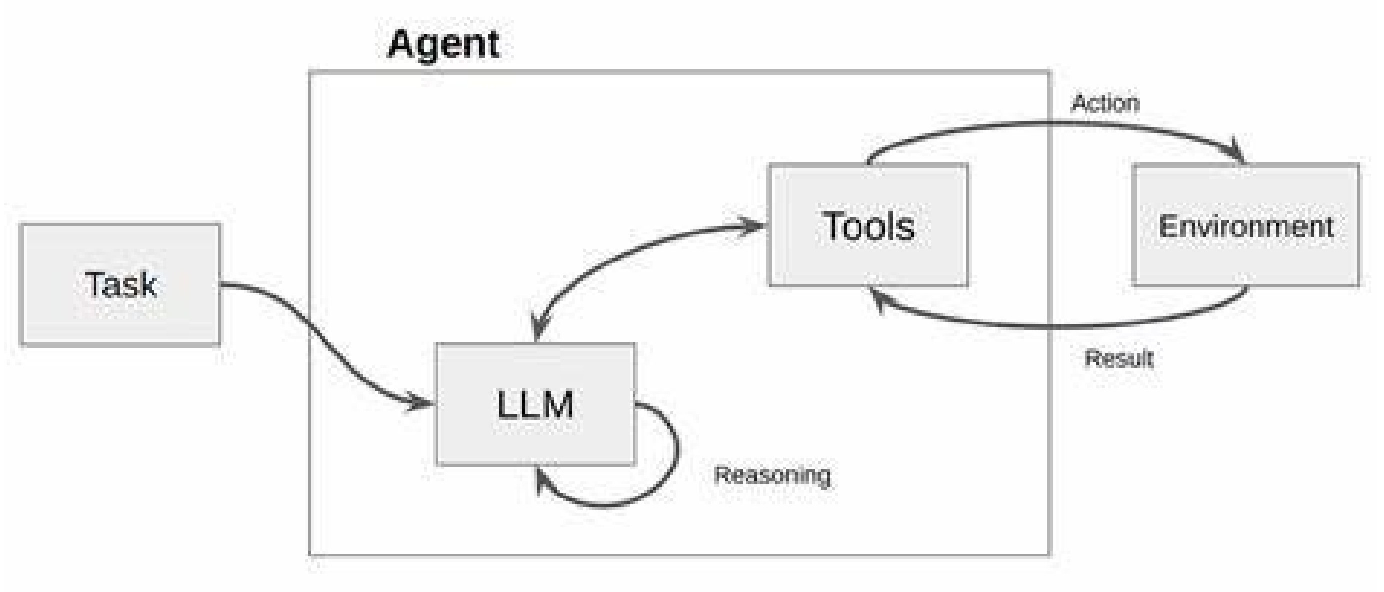
- The documents are then loaded into vector stores.

Retrievers

- A retriever is an interface that returns documents given an unstructured query.
- It is a lightweight wrapper around the vector store class to make it conform to the retriever interface.
- It uses the search methods implemented by a vector store, like similarity search and MMR, to query the texts in the vector store.

Agents

- An agent is a system that decides the sequence of actions to take perceiving the environment.It operates anonymously.
- Here, a LLM is used as a reasoning engine to determine which actions to take and in which order.
- After executing actions, the results can be fed back into the LLM to determine whether more actions are needed, or whether it is okay to finish.



Agents – Key Components

Schema

- **AgentAction** - Dataclass that represents the action an agent should take. It has a ’tool’ and ‘tool_input’ property.
- **AgentFinish** - This represents the result from an agent, when it is ready to return to the user.
- **Intermediate Steps -** These represent previous agent actions and corresponding outputs from this CURRENT agent run.

Agent Executor

- The agent executor is the runtime for an agent. This is what calls the agent, executes the actions it chooses, passes the action outputs back to the agent, and repeats.

Tools

- Tools are functions that an agent can invoke.
  The Tool abstraction consists of two components:
  1. The **input schema for the tool**. This tells the LLM what parameters are needed to call the tool.
  2. The **function to run**. This is generally just a Python function that is invoked.

Agents - Types

Some common agent types are:

- OpenAI Tools – This type of agent is intended for Chat models, supports chat history, parallel function calling and multi-input tools. It should be used when OpenAI models are used.

- ReAct Agent – This type of agent is intended for LLMs and supports chat history. It is used when we are using simple models.

Categorization of agents depends on:

- Intended Model Type
- Chat History Support
- Multi-Input Tools Support
- Parallel Function Calling Support
- Model Parameters Requirement