

AI Agents

Introduction

AI Agents represent an exciting development in Generative AI, enabling Large Language Models (LLMs) to evolve from assistants into agents capable of taking actions. AI Agent frameworks enable developers to create applications that give LLMs access to tools and state management. These frameworks also enhance visibility, allowing users and developers to monitor the actions planned by LLMs, thereby improving experience management.

Learning goals

After taking this lesson, you'll be able to:

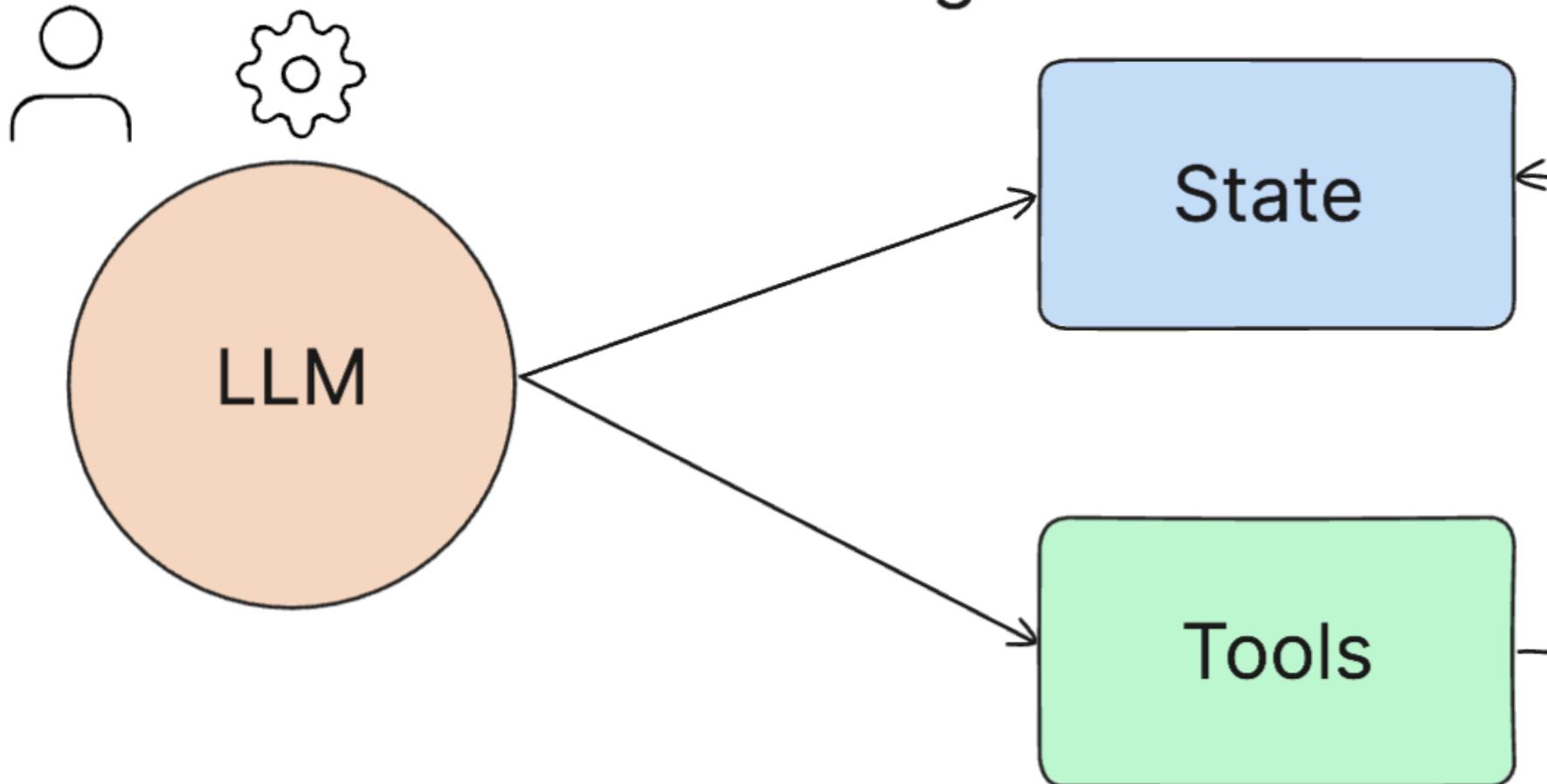
- Explain what AI Agents are and how they can be used.
- Have an understanding of the differences between some of the popular AI Agent Frameworks, and how they differ.
- Understand how AI Agents function in order to build applications with them.

What Are AI Agents?

AI Agents are a very exciting field in the world of Generative AI. With this excitement comes sometimes a confusion of terms and their application. To keep things simple and inclusive of most of the tools that refer to AI Agents, we are going to use this definition:

AI Agents allow Large Language Models (LLMs) to perform tasks by giving them access to a **state and tools**.

What is an Agent?



LLM

- "Choose Your Own Adventure"

Let's define these terms:

Large Language Models - These are the models referred throughout this course such as GPT-3.5, GPT-4, Llama-2, etc.

State - This refers to the context that the LLM is working in. The LLM uses the context of its past actions and the current context, guiding its decision-making for subsequent actions. AI Agent Frameworks allow developers to maintain this context easier.

Tools - To complete the task that the user has requested and that the LLM has planned out, the LLM needs access to tools. Some examples of tools can be a database, an API, an external application or even another LLM!

These definitions will hopefully give you a good grounding going forward as we look at how they are implemented. Let's explore a few different AI Agent frameworks:

LangChain Agents

[LangChain Agents](#) is an implementation of the definitions we provided above.

To manage the **state**, it uses a built-in function called the `AgentExecutor`. This accepts the defined `agent` and the `tools` that are available to it.

The `Agent Executor` also stores the chat history to provide the context of the chat.

LangChain Agents



```
from langchain_openai import  
ChatOpenAI  
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)
```

Context
System Message

You Are a Helpful Assistant

State

AgentExecutor
Chat History

```
agent = create_openai_functions_agent(llm,  
tools, prompt)
```

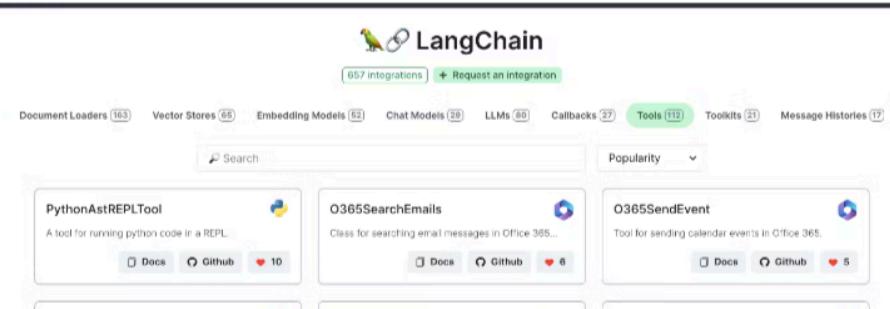
Tools

Search,
Services,
Databases

```
agent_executor =  
AgentExecutor(agent=agent,  
tools=tools, verbose=True)
```

```
agent_executor.invoke({"input":  
"hi! my name is bob",  
"chat_history": []})
```

```
from  
langchain_community.tools.tavily_search import TavilySearchResults  
  
tools = [search, retriever_tool]
```



AutoGen

The next AI Agent framework we will discuss is [AutoGen](#). The main focus of AutoGen is conversations. Agents are both **conversable** and **customizable**.

Conversable - LLMs can start and continue a conversation with another LLM in order to complete a task. This is done by creating `AssistantAgents` and giving them a specific system message.

```
autogen.AssistantAgent( name="Coder", llm_config=llm_config, ) pm = autogen.AssistantAgent( name="Product_manager", system_message="Creative in software product ideas.", llm_config=llm_config, )
```

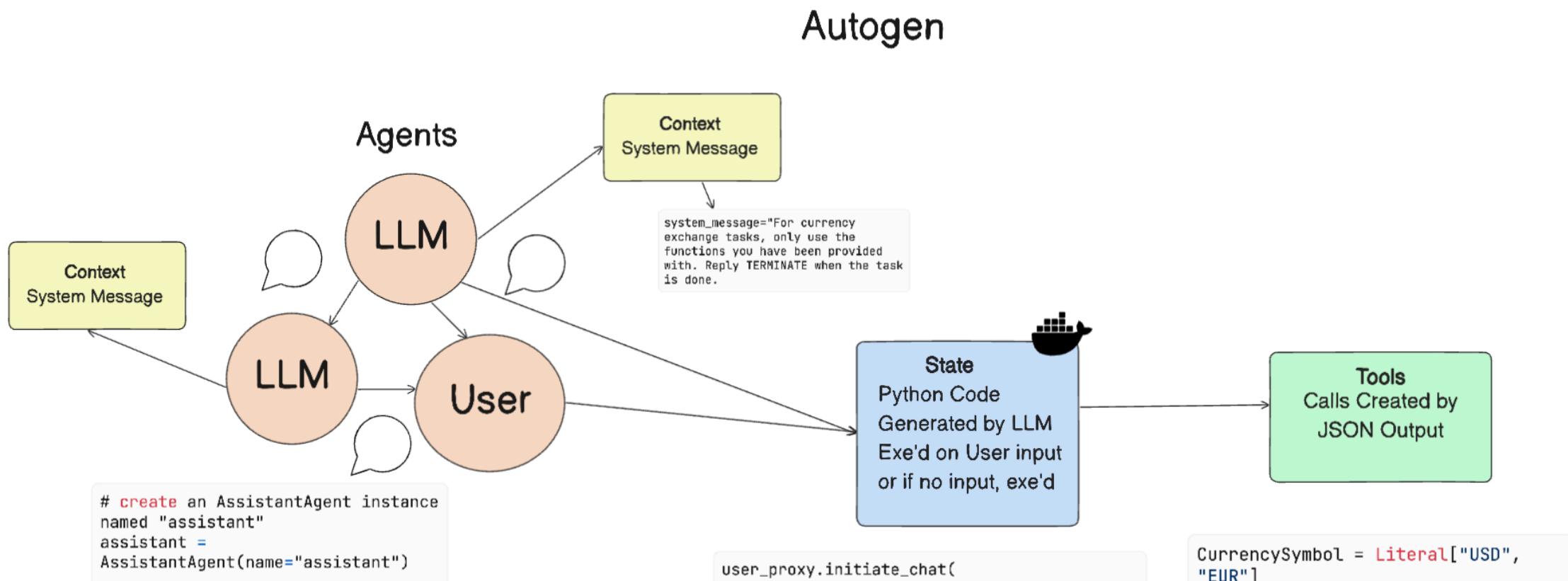
Customizable - Agents can be defined not only as LLMs but be a user or a tool. As a developer, you can define a `UserProxyAgent` which is responsible for interacting with the user for feedback in completing a task. This feedback can either continue the execution of the task or stop it.

```
user proxy = UserProxyAgent(name="user proxy")
```

State and Tools

To change and manage state, an assistant Agent generates Python code to complete the task.

Here is an example of the process:



Taskweaver

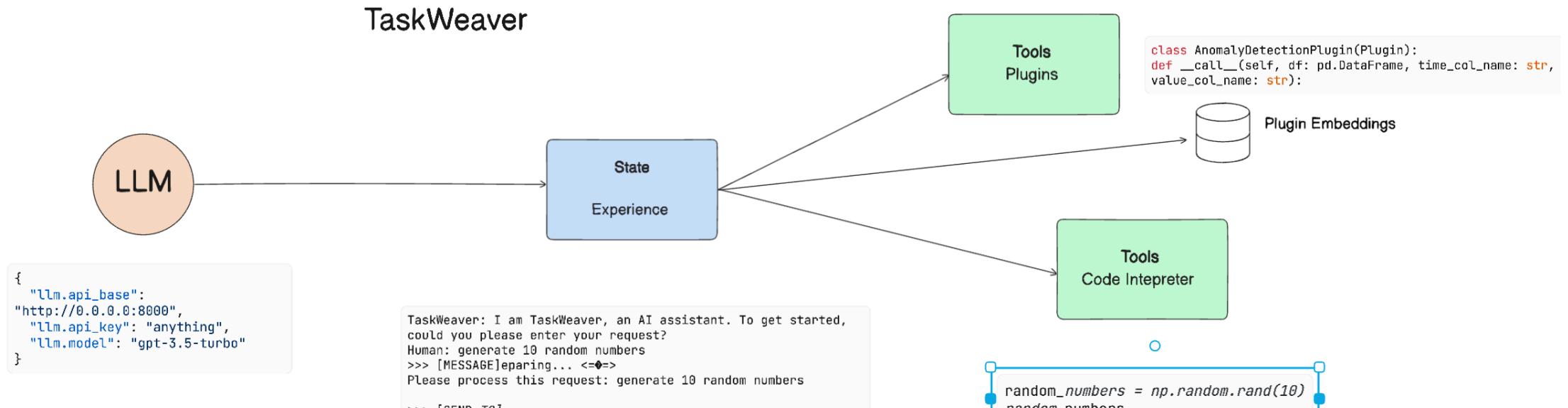
The next agent framework we will explore is [Taskweaver](#). It is known as a "code-first" agent because instead of working strictly with `strings`, it can work with `DataFrames` in Python. This becomes extremely useful for data analysis and generation tasks. This can be things like creating graphs and charts or generating random numbers.

State and Tools

To manage the state of the conversation, TaskWeaver uses the concept of a `Planner`.

The `Planner` is a LLM that takes the request from the users and maps out the tasks that need to be completed to fulfill this request.

To complete the tasks the `Planner` is exposed to the collection of tools called `Plugins`. This can be Python classes or a general code interpreter. These plugins are stored as embeddings so that the LLM can better search for the correct plugin.



JARVIS

The last agent framework we will explore is **JARVIS**. What makes JARVIS unique is that it uses an LLM to manage the `state` of the conversation and the `tools` are other AI models. Each of the AI models are specialized models that perform certain tasks such as object detection, transcription or image captioning.

