

# ITCS 4101: Introduction to NLP

---

## LangChain for Building LLM-powered Applications

Justin Smith, with input from Erfan and Razvan

[jsmit840@charlotte.edu](mailto:jsmit840@charlotte.edu)

Department of Computer Science @ CCI



# Learning Objectives

---

- What is a **multi-agent** setting and when is it useful?
- What is the **LangChain** Library?
- What are LangChain **components**?
  - How do you integrate them with each other?
- How does a LangChain **agent** work?
- For what cases does LangChain work?
- Under what circumstances does LangChain **fail**?

# Agents

---

- An **agent** is a program that can independently interact with its environment.
  - “Independent” referring to not needing step-by-step instructions to complete objectives.
- We can implement an agent via language model, at least to some degree.
  - All input and output of the language model is language.

# Multi-Agent Environments

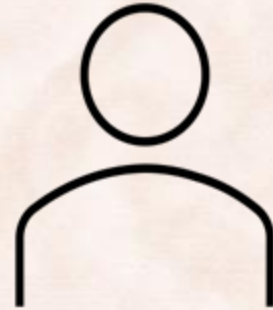
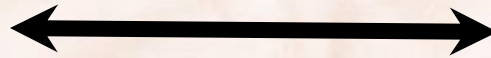
---

- Since agents are independent, these agents can interact with one another.
  - These agents can be trying to accomplish different goals.
- We call these environments *multi-agent environments*.



# Single Agent

---





Data  
scientist



Architect  
/ Planner



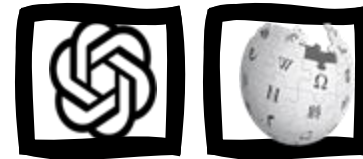
Code  
executor



Storyteller



Content  
extractor



Nested  
tool



# Jupyter Notebook

---

- Jupyter Notebook – We can see an example of two agents interacting in a comedy routine.

# Motivation for Agents

---

- Sometimes a language model is not the best for a task.
  - Most state-of-the-art language models still struggle with:
    - Arithmetic
    - Code execution
    - Very long-context input
- We want to allow our language model agents to interact with other **agents** or **tools** that are better suited for the task. For example:
  - Calculators
  - Compilers/Interpreters
  - Retrieval systems (Vector stores, Graphs, ...)
- Two ways to implement this are with **LangChain** or **AutoGen**.



# What is LangChain?

---

- A library that is designed to ease the development of applications that utilize both an **LLM** and **external tools**.
  - More powerful applications than a single prompt: **dialogue systems** (next homework), **data science** (query a database), **multimodal input** (send preprocessed audio/image/video to the LLM).
- **Central idea:** “Chain” together the LLM and, optionally, additional components.
  - Similar to a linked list, the “chain” is sequential.
  - Dissimilar to a linked list, the input to the next **component** is cumulative

# LangChain Core Components

---

- **LLMs:** Large language models, e.g. GPT-3.5, GPT-4, Llama3, Gemini.
- **Prompt templates:** A template for building a chain of prompts
  - The output of one prompt can be input into a different prompt.
- **Memory:** Keeps track of past messages and information from external tools.
- **Agents:** Abstraction of the chain of LLM prompts and tools.
  - To the user, a chain of prompts is single system to interact with.
- **Vector Stores:** Databases that store the meaning of various text documents.

# Common Components Beyond the LLM

<https://python.langchain.com/docs/integrations/components>

- **Vector Stores:**

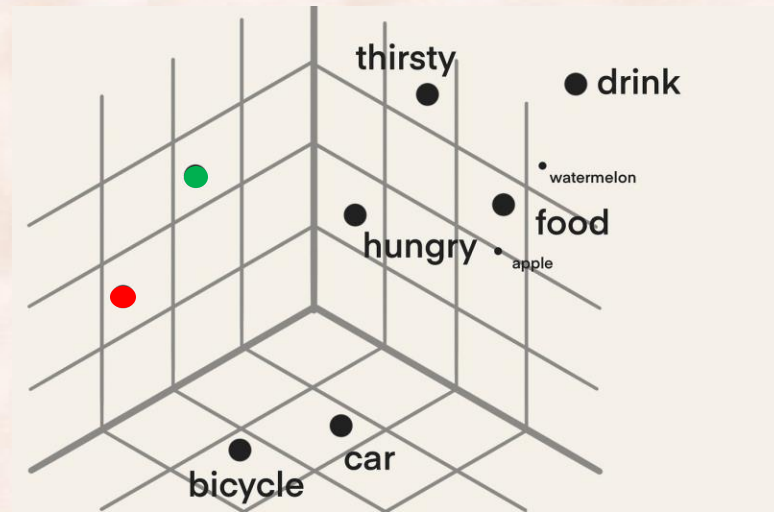
- A type of database that stores data as high-dimensional vectors
- This data can be text, images, video, and more.

- **Why do we use Vector Stores?**

- Find most similar:
  - *documents* for a query.
  - training *examples* for a test example.

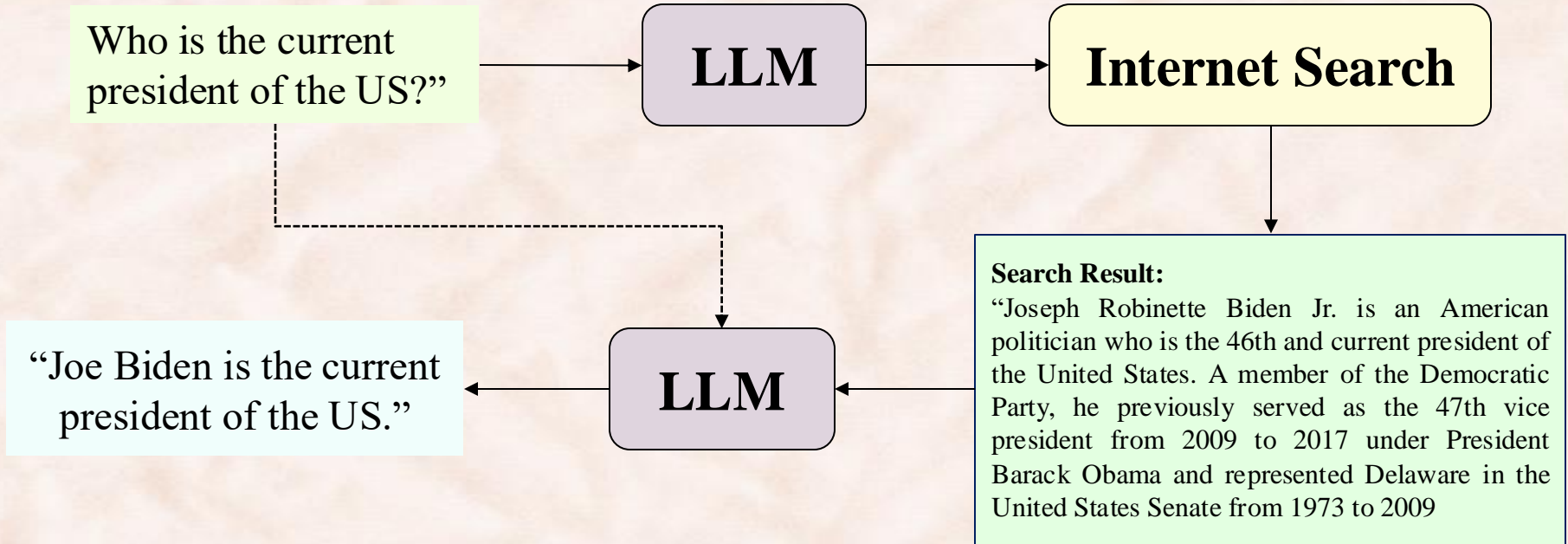
- **Tools:**

- Web Search (this lecture).
- Calculators, Compilers, Interpreters, ...
- Image Generation, File Access, ...



# The Chain in LangChain

---



# The Chain in LangChain

---

- **PromptTemplate** class:
  - Sets up a prompt with input variables, like a formatted string.
  - Make an instance of the prompt with the method `format(inputs)`.
- **LLM** class:
  - Contains a method to call an LLM.
- **LLMChain** class:
  - Takes in an LLM as an object, along with a prompt template.
  - Has a method, `invoke(inputs)`, that will get the LLM response.
- **Agent** class:
  - Abstract an LLM, along with other components and a prompt, into a single object.
- **AgentExecutor** class:
  - Method, `invoke(inputs)`, will get the response of the entire chain.

# The Chain in LangChain

---

```
from langchain_core.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain_openai import ChatOpenAI
```

```
llm = ChatOpenAI(api_key = "aewndfoa1235123", base_url = base_url,
                 model = model_name, temperature= 0, max_tokens = 1000)
```



# The Chain in LangChain

```
setup = "# Character Description\n\n {character}\n\n # Scene\n\n {scene}\n\n # Context\n\n {context}"

scene = get_completion_from_messages(storyteller_messages)
print(scene)

mac_desc = "You are Mac, a character in a comedy sketch. He will be the \"Funny Man\" in a Straight Man and Funny Man comedy sketch." \
           "He uses a lot of double-speak, where he says metaphors that can be misinterpreted humorously, and gets frustrated when his co-star" \
           " Jim does not get it. He never explains what he actually means, only doubling down on his double-speak. Do not respond as Jim. Always " \
           "prepend your messages with 'Mac:'."

jim_desc = "You are Jim, a character in a comedy sketch. He will be the \"Straight Man\" in a Straight Man and Funny Man comedy sketch." \
           "He misinterprets all metaphors in a humorous manner, showing that he is naive. He does not understand that Mac, the other character, is" \
           " ever mad at him. Do not respond as Mac. Always prepend your messages with 'Jim:'."

context = ""

prompt = PromptTemplate.from_template(setup)

chain = LLMChain(llm=llm, prompt = prompt)
```

# AutoGen

---

- [AutoGen](#) has two major classes that control how it is used:
  - The **Chat** class:
    - This class controls the chat history.
  - The **ConversableAgent** class:
    - This class describes an agent that can be chatted with.
- There are two main ConversableAgents.
  - **UserProxyAgent**:
    - This is an agent that allows for user interaction with the agents.
  - **AssistantAgent**:
    - This is a wrapper around the typical way to interact with LLMs.

# ReAct Prompting

---

- [ReAct prompting](#) is an approach for enhancing the decision making of LLMs, for example, choosing which tools to use for a given situation.
- This utilizes a format that is simple to parse:
  - **Question:** The input to the model.
  - **Thought:** A place for the model to state intentions, which tends to increase performance.
  - **Action:** The tool (if any) does the model wants to use.
  - **Action Input:** The input to the tool.
  - **Observation:** The output of the tool.
- The final answer is given in the format:
  - **Thought:** “I believe I have the final answer.”
  - **Final Answer:** The answer the model gives to the user.

# ReAct Prompting

---

- The structure of the ReAct Loop:

– <question> (<thought> <action> <action input> <observation>)\* <thought> <answer>

**Question:** Who is the current president of the US?

**Thought:** I should use a search engine to find relevant data.

**Action:** Open Google

**Action Input:** “Who is the current president of the US?”

**Observation:** The current president is Joe Biden.

**Thought:** I now know the final answer.

**Answer:** Joe Biden

Option 1: *You do not have enough information to answer the question.*

Option 2: *You have enough information to answer the question.*

# ReAct Prompting

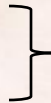
---

- The structure of the ReAct Loop:
  - `<question>` (`<thought>` `<action>` `<action input>` `<observation>`)\* `<thought>` `<answer>`

**Question:** What is 2 + 2?

**Thought:** I now know the final answer.

**Final Answer:** 4



Option 2: *You have enough information to answer the question.*

# LangChain for ReAct Prompting

```
instructions = """
Answer the following questions as best you can. You have access to the following tools:

{tools}

Use the following format:
Question: <the input question>

Following this, use one of the following options:
```

```
Option 1 - You do not have enough information to answer the question:
Thought: <plans for which tool you will utilize and how you will use it>
Action: <the action to take, should be one of [{tool_names}]>
Action Input: <the input to the action>
Observation: <the result of the action>
```

```
Option 2 - You have enough information to answer the question:
Thought: I now know the final answer
Final Answer: <the final answer to the original input question>
```

For example:

Question: Who composed the Four Seasons?

```
Thought: I should utilize serpapi to search the internet for this information.
Action: serpapi
Action Input: Who composed The Four Seasons?
Observation: Antonio Vivaldi
```

```
Thought: I now know the final answer
Final Answer: Antonio Vivaldi
```

In another example:

```
Question: What is the chemical formula of water?
Thought: I now know the final answer
Final Answer: H2O.
```

Begin!

```
Question: {input}
Thought: {agent_scratchpad}
```



# The Agent of LangChain

---

- **ChatPromptTemplate** class:
  - Very similar to the OpenAI chat; however, it utilizes tuples instead of dictionaries.
- **Agent** class:
  - Abstract an LLM, along with other components and a prompt, into a single object.
- **AgentExecutor** class:
  - Method, `invoke(inputs)`, will get the response of the entire chain.

# LangChain for ReAct Prompting

```
# Import the necessary classes.
from langchain_core.prompts import ChatPromptTemplate
from langchain.agents import AgentExecutor, create_react_agent, load_tools
from langchain import hub

# Boilerplate code setting up the prompt template, the list of tools, the agent, and the agent executor.
# using the 'system_message' and 'instructions' that were instantiated in the cell above.
prompt = ChatPromptTemplate.from_messages(
    [
        (
            'system',
            system_message
        ),
        (
            'user',
            instructions
        )
    ]
)

# Input either into a .env file or directly into this cell your API key from serpapi
# Where to get your API key: https://serpapi.com/
tools = load_tools(['serpapi'], llm=llm, serper_api_key=serper_api_key)

agent = create_react_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True, handle_parsing_errors=True, max_iterations= 4)
```

# The Chain in LangChain

---

- **How do these LLMs and tools work together?**
  - `input_variable` is a parameter of the `PromptTemplate` class
  - This allows linking the output to one component with the input of another.
- **What format style should we use for exporting information from a tool?**
  - JSON is a great format to output in.
  - Many LMs can comprehend and produce JSON consistently.
  - Other times, just a string is sufficient.

# LangChain Example

---

- Jupyter notebook.

# Zero-Shot

- Zero-shot refers to giving the LLM no examples, only the instructions:
  - This is sufficient for some tasks, but sometimes it is hard to clearly express the intended output in English, e.g. output format.

USER

Given the short story below, extract a graph of all the characters, where characters are represented as nodes, and edges represent relationships that are expressed in the text. The graph should be output in JSON as a list of (node1, node2, relationship) triplets.

Short story: "There was once a hare who was friends with a tortoise. One day, he challenged the tortoise to a race. Seeing how slow the tortoise was going, the hare thought he'd win this easily. So, he took a nap while the tortoise kept on going. When the hare woke, he saw that the tortoise was already at the finish line. Much to his chagrin, the tortoise won the race while he was busy sleeping."

ASSISTANT

```
```json
[
  {"node1": "hare", "node2": "tortoise", "relationship": "friends"},
  {"node1": "hare", "node2": "tortoise", "relationship": "challenged to a race"},
  {"node1": "hare", "node2": "race", "relationship": "took a nap during"},
  {"node1": "tortoise", "node2": "race", "relationship": "kept going"},
  {"node1": "tortoise", "node2": "finish line", "relationship": "at"},
  {"node1": "hare", "node2": "finish line", "relationship": "saw tortoise at"},
  {"node1": "tortoise", "node2": "race", "relationship": "won"},
  {"node1": "hare", "node2": "race", "relationship": "lost"}
]
```
```

OpenAI Playground example here: [Zero-Shot Example with GPT-4-Turbo](#)

# In-context Learning: Few-Shot Examples

[Language Models are Few-Shot Learners](#)

---

- Sometimes it's better to give a few examples to the LLM to explain the problem.
  - For example, 1-shot refers to one example, 2-shot to two examples, and so on.
  - In-context examples tend to increase the efficiency of the model.
- In general, this is referred to as *few-shot*.
- Let's try the same prompt as before, but with examples of what we want the model to produce.

OpenAI Playground example here: [Few-shot example with GPT-4-Turbo](#)



# Examples and Improved Output

## USER

Given a short story, extract a graph of all the characters, where characters are represented as nodes, and edges represent relationships that are expressed in the text. The graph should be output in JSON as a list of (node1, node2, relationship) triplets.

For example, if the short story is:

"Red Riding Hood had lunch at her grandmother's place. Then she parted ways with her grandmother and walked into the woods. She met and talked with the Wolf, who followed her deep into the forest."

then the output should be:

```
[  
  ["Red Riding Hood", "grandmother", "parted ways with"],  
  ["Red Riding Hood", "Wolf", "met"],  
  ["Red Riding Hood", "Wolf", "talked"],  
  ["Wolf", "Red Riding Hood", "followed"]  
]
```

If the short story is:

"The raccoon brushed his teeth at 5pm, while the elephant got ready at 6pm. The raccoon met with the elephant at 7pm for dinner. The elephant got a steak, the raccoon got mashed potatoes. The elephant went home at 9pm, the raccoon went home at 10pm. Later, the elephant made a phone call to the raccoon.",

then the output should be:

```
[  
  ["raccoon", "elephant", "met at 7pm for dinner"],  
  ["elephant", "raccoon", "made a phone call"]  
]
```

Now output the graph for this short story: "There was once a hare who was friends with a tortoise. One day, he challenged the tortoise to a race. Seeing how slow the tortoise was going, the hare thought he'd win this easily. So, he took a nap while the tortoise kept on going. When the hare woke, he saw that the tortoise was already at the finish line. Much to his chagrin, the tortoise won the race while he was busy sleeping."

```
...  
[  
  ["hare", "tortoise", "friends with"],  
  ["hare", "tortoise", "challenged to a race"],  
  ["hare", "tortoise", "took a nap while"],  
  ["tortoise", "hare", "kept on going"],  
  ["hare", "tortoise", "saw at the finish line"],  
  ["tortoise", "hare", "won the race"]  
]  
...
```

## ASSISTANT

# Critique of LangChain

---

- LangChain is unnecessary when using solely the LLM.
  - Instead, easier to just keep track of the conversation in the code.
- Tools are not always the simplest, most effective solution.
  - It might be enough to provide examples to the LLM and have it accomplish the task.
- LangChain does not improve the base model.
- LangChain cannot improve the context length.
  - No language model can keep an infinitely long conversation going.
  - Sometimes a longer prompt outperforms a long list of function calls.

# Learning Objectives

---

- What is a **multi-agent** setting and when is it useful?
- What is the **LangChain** Library?
- What are LangChain **components**?
  - How do you integrate them with each other?
- How does a LangChain **agent** work?
- For what cases does LangChain work?
- Under what circumstances does LangChain **fail**?

# Frameworks for deploying LLM-based Web Apps

---

- [Streamlit](#):
  - **Pros**: Python-based, more configurable than Chainlit (e.g. authentication), more UI components than Chainlit and Gradio, and easy to deploy for free on a public URL through [share.streamlit.io](https://share.streamlit.io) (through a .edu account)
  - **Cons**: Needs additional work to support token streaming and does not support visualizing LangChain prompt chains.
- [Chainlit](#):
  - **Pros**: Python-based, easily visualizes prompt chains from LangChain (as of version 0.2.0), and supports token streaming.
  - **Cons**: Needs some work to deploy and does not offer free hosting that can be accessed through a public URL. To host on a public URL students would need to use a cloud provider like [Heroku](#).

# Frameworks for deploying LLM-based Web Apps

---

- [Chat UI](#):
  - **Pros**: Easily integratable with a multitude of LLM hosting services and frameworks such as Azure, HuggingFace Inference endpoints, and vLLM. Full UI and deployment configurability. Chat-UI is a replica of HuggingChat.
  - **Cons**: Needs JavaScript to customize, does not visualize LangChain prompt chains.
- [Gradio](#):
  - **Pros**: Python-based, more UI components than Chainlit, and easy to deploy to a public URL through Huggingface spaces.
  - **Cons**: Does not support visualizing LangChain prompt chains.



# Recommended Readings

---

- <https://learn.deeplearning.ai/langchain>
- [https://python.langchain.com/docs/get\\_started/introduction](https://python.langchain.com/docs/get_started/introduction)
- <https://github.com/microsoft/autogen>
  - [https://microsoft.github.io/autogen/docs/Use-Cases/agent\\_chat/](https://microsoft.github.io/autogen/docs/Use-Cases/agent_chat/)
  - [https://microsoft.github.io/autogen/docs/Use-Cases/agent\\_chat/#diverse-applications-implemented-with-autogen](https://microsoft.github.io/autogen/docs/Use-Cases/agent_chat/#diverse-applications-implemented-with-autogen)