Introduction to Large Language Models

Spring 2026

Agentic AI III

Agentic AI Frameworks

(Some slides adapted from Ralph Grishman at NYU,
Yejin Choi at UWashington, N. Tomura at UDepaul, Jurafsky and Martin,
CS224N, CS224, CME295 at Stanford and other resourses on the web)

Agentic AI Frameworks

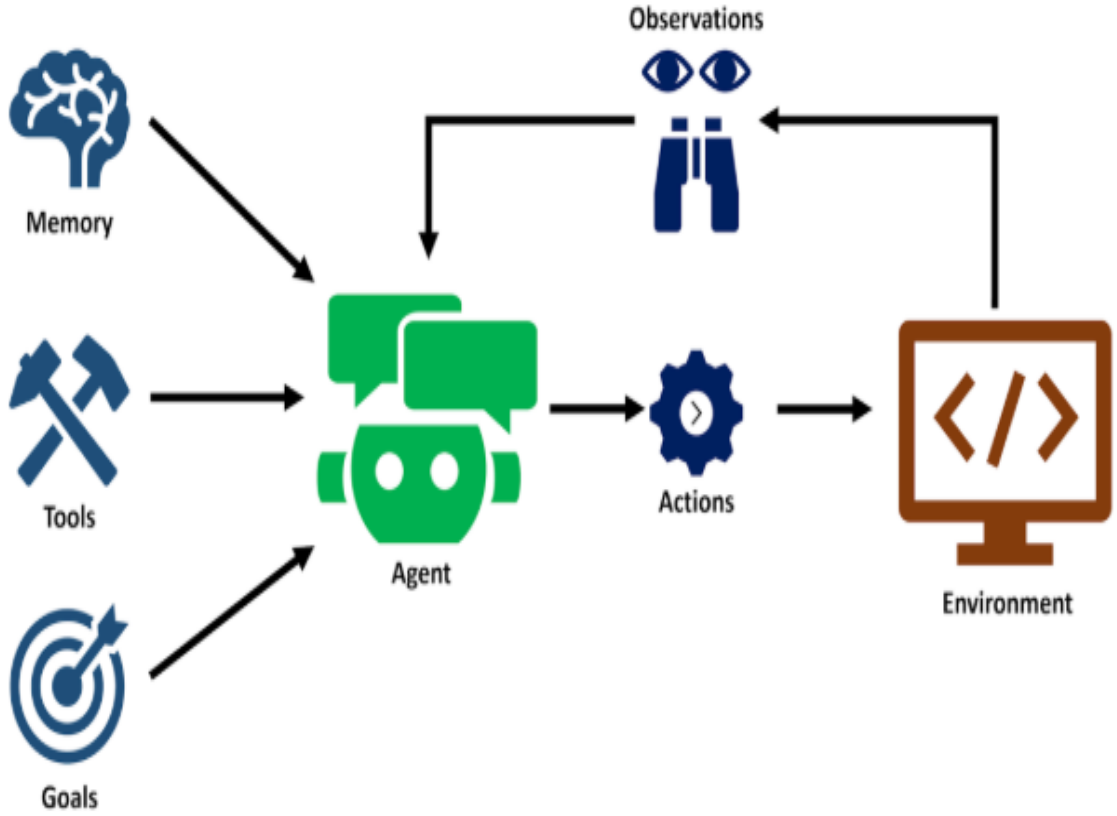
Agentic AI Frameworks

Agentic AI frameworks are software platforms that help developers **build, orchestrate, and manage autonomous AI agents**. They provide reusable components for reasoning, planning, memory, tool use, and multi-agent coordination—so you don't have to build everything from scratch.

What Agentic AI Frameworks Provide

Most frameworks support some or all of the following:

- **Agent loop** (observe → reason → act → reflect)
- **Planning strategies** (ReAct, plan-and-execute, hierarchical)
- **Memory management** (short-term, long-term, vector stores)
- **Tool integration** (APIs, search, code execution)
- **Multi-agent coordination**
- **Evaluation and control** (constraints, retries, guardrails)



https://www.var-meta.com/blog/top-5-agentic-ai-frameworks-in-2025#core-components-of-agentic-ai-in-building-intelligent-applications

Agentic AI Frameworks

Agentic AI Frameworks (List)

Below is a **commonly used list of agentic AI frameworks**, grouped by focus and use case.

General-Purpose & Modular Frameworks

- **LangChain** – Modular framework for LLM apps and tool-using agents
- **LangGraph** – Stateful, graph-based orchestration for complex agents
- **Semantic Kernel** – Enterprise-oriented agent and plugin framework
- **Haystack Agents** – Agent framework focused on RAG and document workflows

Autonomous / Single-Agent Frameworks

- **AutoGPT** – Early autonomous agent with plan–execute–reflect loop
- **BabyAGI** – Task-driven autonomous agent with memory
- **SuperAGI** – Autonomous agent platform with planning and execution

Agentic AI Frameworks

Agentic AI Frameworks (List)
Multi-Agent Frameworks

- **Microsoft AutoGen** – Conversational multi-agent coordination
- **CrewAI** – Role-based cooperative multi-agent systems
- **MetaGPT** – Structured multi-agent collaboration inspired by software teams
- **ChatDev** – Simulated AI software development team

Research-Oriented / Simulation Frameworks

- **OpenAI Gym / Gymnasium** – Multi-agent learning environments
- **PettingZoo** – Multi-agent reinforcement learning environments
- **MAgent** – Large-scale multi-agent simulations

Workflow & Orchestration Platforms

- **Apache Airflow (LLM-augmented)** – Task orchestration with agents
- **Prefect + LLM agents** – Workflow-driven agent execution

Top Frameworks Compared

Framework	Core Focus	Agent Type	Strengths	Limitations	Best Use Cases
LangChain	Agent components & chaining	Single-agent (primary)	Modular, large ecosystem, strong tool & RAG support	Limited control flow, implicit state	RAG apps, simple agents, prototyping
LangGraph	Agent orchestration via graphs	Single & multi-agent	Explicit state, loops, branching, production-grade control	Higher complexity	Long-running agents, complex workflows
Microsoft AutoGen	Multi-agent communication	Multi-agent	Structured agent conversations, human-in-the-loop	Less suited for simple tasks	Collaborative reasoning, agent teams
CrewAI	Role-based multi-agent teams	Multi-agent	Clear roles, parallel execution, easy delegation	Less flexible control logic	Task decomposition, team-style workflows
Semantic Kernel	Enterprise agent orchestration	Single & multi-agent	Planning, memory, plugins, enterprise readiness	Steeper learning curve	Business & enterprise systems
AutoGPT	Autonomous agents	Single-agent	End-to-end autonomy, planning + reflection	Hard to control, brittle	Exploratory autonomy, demos
Haystack Agents	RAG-centric agents	Single-agent	Strong document pipelines, retrieval	Less general agent behavior	QA systems, knowledge agents

Top Frameworks Compared - Architectural Perspective

Aspect	LangChain	LangGraph	AutoGen	CrewAI	Semantic Kernel
Control Flow	Linear	Graph-based	Conversational	Role-driven	Planner-driven
State Management	Implicit	Explicit	Message-based	Shared context	Explicit
Planning Support	Medium	Strong	Medium	Medium	Strong
Multi-Agent	Limited	Native	Native	Native	Native
Production Control	Medium	High	Medium	Medium	High

When to Choose What

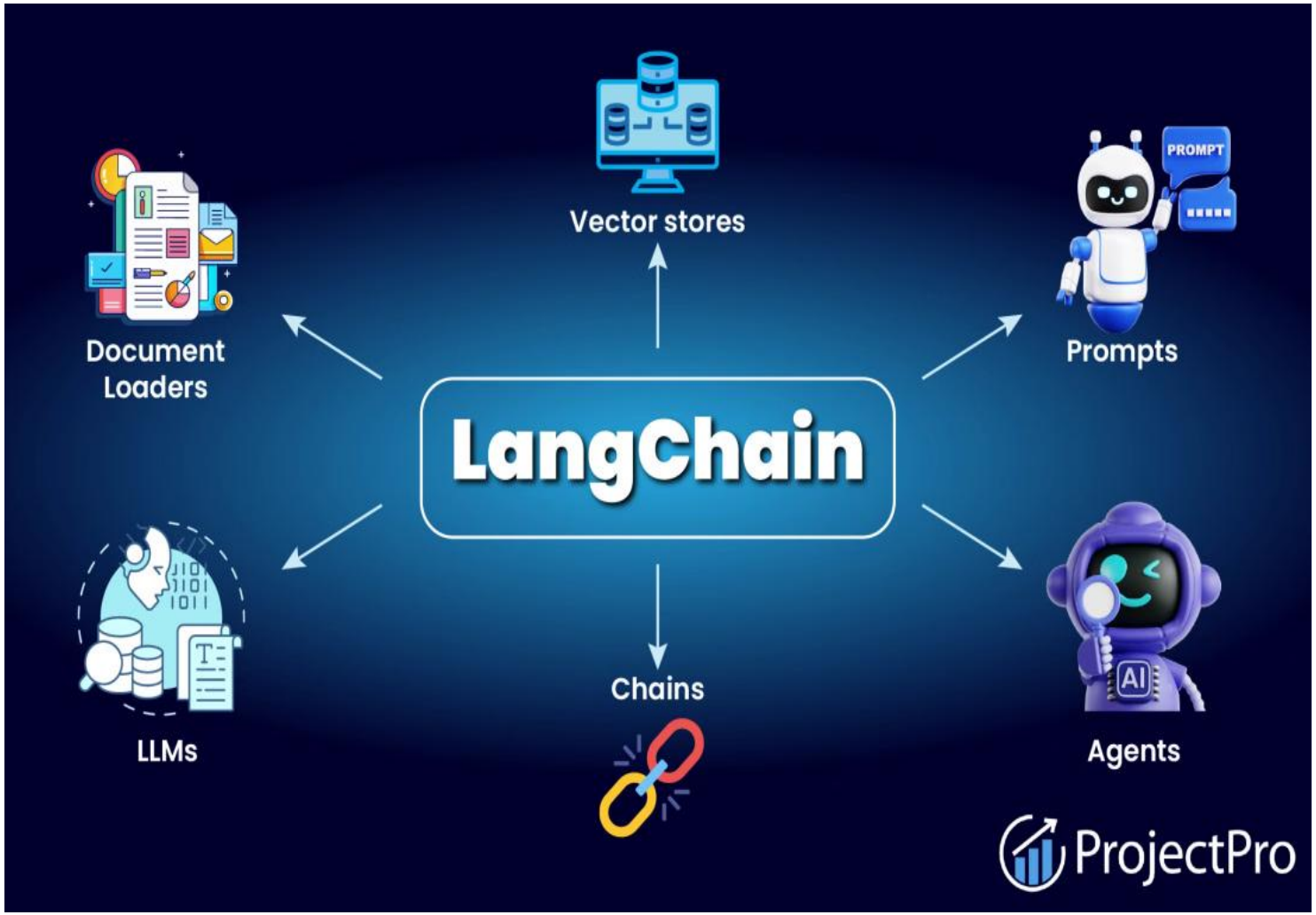
- **LangChain** → Simple agents, RAG, fast prototyping
- **LangGraph** → Complex, long-horizon, stateful or multi-agent systems
- **AutoGen** → Collaborative reasoning and agent dialogue
- **CrewAI** → Task delegation with clear agent roles
- **Semantic Kernel** → Enterprise-grade agent platforms
- **AutoGPT** → Autonomy experiments (not tight control)

Key Insight:

- LangChain builds agents.
- LangGraph controls agents.
- AutoGen and CrewAI scale agents sideways (teams).
- Semantic Kernel hardens agents for enterprise use.

LangChain

LangChain is an open-source framework for building **LLM-powered applications and agents** by composing modular components such as prompts, models, tools, and memory.



What LangChain Is For

LangChain helps developers:

- Build **LLM applications** quickly
- Create **tool-using agents**
- Chain multiple reasoning steps together
- Integrate **external data** (RAG)
- Manage **conversation history and memory**

https://www.projectpro.io/article/langchain/894

LangChain

Core Components

1. **Models**
   - Wrappers for LLMs, chat models, and embeddings
2. **Prompt Templates**
   - Reusable, parameterized prompts
3. **Chains**
   - Linear pipelines of steps

```
Input → Prompt → LLM → Output
```

5. **Tools**
   - APIs, search, databases, code execution
6. **Agents**
   - LLMs that decide **which tool to use and when**
   - Often use the **ReAct** pattern
7. **Memory**
   - Conversation history
   - Vector-based or structured memory

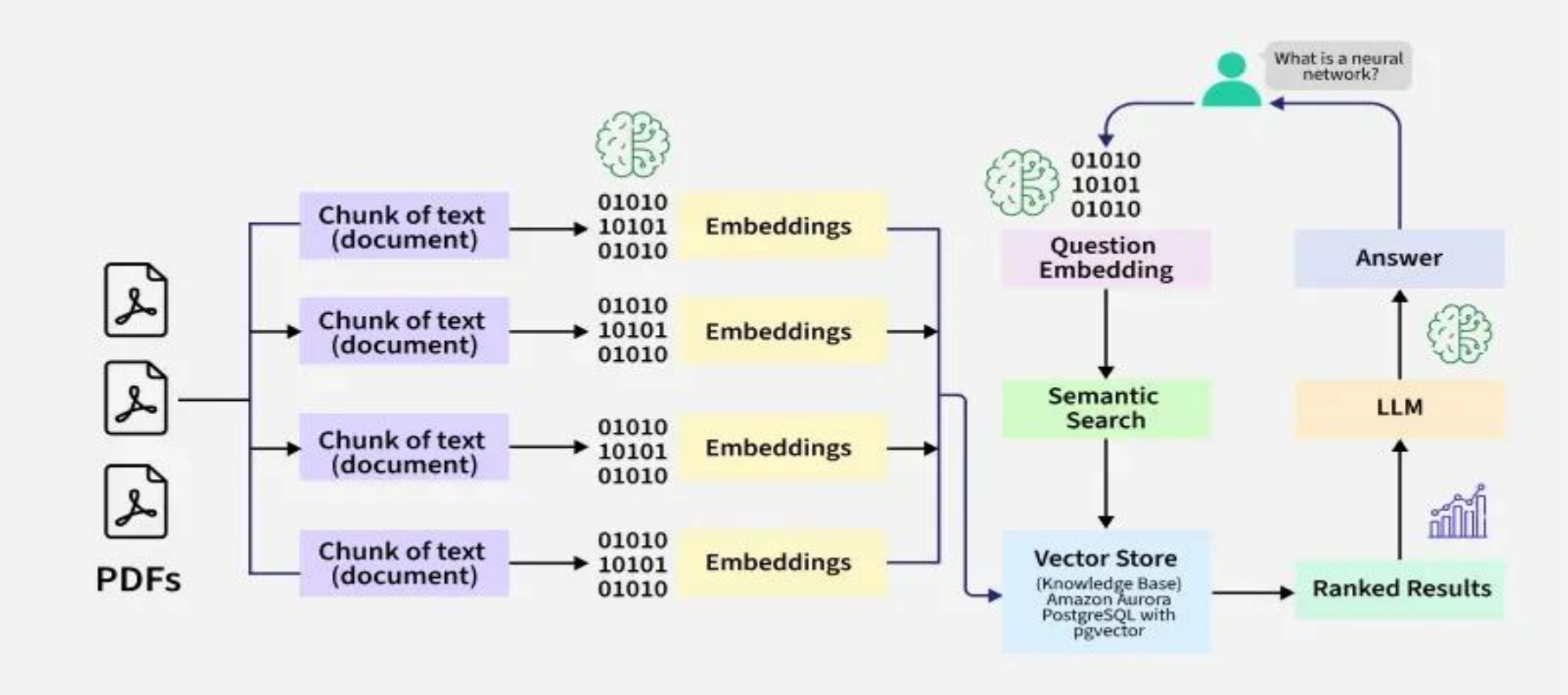
https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangChain

How LangChain Works?

LangChain enables Retrieval-Augmented Generation (RAG) by combining document processing, vector storage and LLMs to generate accurate, context aware responses. It connects embeddings, vector databases and models into a smooth workflow.

RAG pipeline using LangChain



https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangChain

RAG pipeline using LangChain:

1. Document Processing

Documents (e.g., PDFs) are split into smaller chunks so they can be processed efficiently.

2. Embeddings Creation

Each chunk is converted into embeddings that capture its semantic meaning.

3. Vector Store

These embeddings are stored in a vector database, creating a searchable knowledge base.

4. User Query

The process starts when a user submits a question or request as input.

For example, a user might ask, “What’s the weather like today?” This query serves as the input to the LangChain pipeline.

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangChain

RAG pipeline using LangChain:

5. Vector Representation

Once the query is received, LangChain converts it into a vector representation using embeddings. This vector captures the semantic meaning of the query.

6. Similarity Search

This vector is compared with vectors stored in a database to find the most relevant matches based on meaning.

7. Fetching Relevant Information

The system retrieves the most relevant data or context from the vector database to support the response.

8. Generating a Response

The retrieved context is passed to a language model, which processes it and generates a meaningful answer.

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangChain Example

Implementation
Let's implement a model using LangChain and OpenAI API:

Step 1: Install the dependencies

We will install all the required dependencies for our model.

- **langchain:** the core LangChain framework (chains, prompts, tools, memory, etc.).
- **langchain-openai:** OpenAI model wrapper for LangChain (GPT-3.5, GPT-4, etc.).
- **python-dotenv:** to securely manage our API keys inside a .env file.

```
!pip install langchain langchain-openai python-dotenv
```

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangChain

Step 2: Import Libraries

We will import all the required libraries.

- **os:** interact with environment variables.
- **load_dotenv:** loads .env file values into our environment.
- **OpenAI:** lets us call OpenAI's GPT models in LangChain.
- **PromptTemplate:** define structured prompts with placeholders.
- **StrOutputParser:** ensures model response is returned as clean string text.

```
import os
from dotenv import load_dotenv
from langchain_openai import OpenAI
from langchain.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser
```

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangChain

Step 3: Load API Key

We need to load the OpenAI API Key, but first we create a .env file to store our API key.

```
OPENAI_API_KEY = your_openai_api_key_here
```

Now we use the os.getenv() function to securely fetch the API key.

```
load_dotenv()
api_key = os.getenv("OPENAI_API_KEY")
```

Step 4: Initialize the OpenAI LLM

We initialize the LLM model:

- **temperature=0.7:** controls creativity (0 = deterministic, 1 = very creative).

- **openai_api_key=api_key:** authenticates with OpenAI.

```
llm = OpenAI(
    temperature=0.7,
    openai_api_key=api_key
)
```

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangChain

Step 5: Run a Simple Prompt

We will check by running a simple prompt.

•**.invoke():** sends prompt to LLM and returns text output.

```
prompt = "Suggest me a skill that is in demand?"
response = llm.invoke(prompt)
print(" Suggested Skill:\n", response)
```

Output:

```
 Suggested Skill:


Data analysis and data science skills are in high demand in various industries, including technology, finance, healthcare, and marketing. With the increasing amount of data being generat
ed, companies are looking for professionals who can collect, organize, and analyze data to make informed business decisions. Additionally, skills in programming languages such as Python,
 R, and SQL are also highly sought after in the data analysis field.
```

Step 6: Create a Prompt Template

We create a dynamic prompt where {year} can be replaced with input values.

```
template = "Give me 3 career skills that are in high demand in {year}."
prompt_template = PromptTemplate.from_template(template)
```

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangChain

Step 7: Build a Chain

LCEL (LangChain Expression Language): It's a new way to compose LLM workflows using a simple, chainable syntax with the | (pipe) operator.

1. prompt_template

•Fills placeholders (like {year}) with actual inputs.

•Example: "Give me 3 career skills in 2025."

2. llm

•Sends the formatted prompt to the OpenAI model.

•Example input: "Give me 3 career skills in 2025."

•Example output: "1. Data Analytics\n2. AI/ML\n3. Cybersecurity"

3. StrOutputParser()

•Cleans up and ensures the LLM's response is returned as a string.

```
chain = prompt_template | llm | StrOutputParser()
```

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangChain

Step 8: Run the Chain

We run the chain to fetch results.

- **.invoke({"year": "2025"})** replaces {year} with 2025 in the prompt.

- **Final formatted prompt:** "Give me 3 career skills that are in high demand in 2025."

```
response = chain.invoke({"year": "2025"})
print("\n Career Skills in 2025:\n", response)
```

Output:

```
 Career Skills in 2025:


1. Data Analysis and Artificial Intelligence: With the increasing use of technology and data in various industries, the demand for professionals who can analyze and interpret large amoun
ts of data and develop AI solutions will continue to rise in 2025.

2. Digital Marketing: As more businesses shift towards online platforms, the need for professionals who can effectively market products and services through digital channels will be in h
igh demand. This includes skills such as social media marketing, search engine optimization, and content creation.

3. Cybersecurity: With the rise of cyber threats and data breaches, the demand for cybersecurity professionals will continue to grow in 2025. These professionals are responsible for prot
ecting sensitive information and ensuring the security of computer systems and networks.
```

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangChain

When to Use LangChain

Use LangChain when you need:

- RAG applications
- Simple or single-agent systems
- Tool-using LLMs
- Fast experimentation and prototyping

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangGraph

- **LangGraph** is a **stateful, graph-based orchestration framework** (built on top of LangChain) for building **robust, controllable, long-running AI agents**. It is designed for agentic systems that require **loops, branching, memory, and multi-agent coordination**.

1. What LangGraph Is

LangGraph models agent behavior as a **directed graph**:

- **Nodes** → functions or agents (LLM calls, tools, evaluators)
- **Edges** → rules for what runs next
- **State** → shared, persistent data passed between steps

Instead of a linear chain, you explicitly control **how and when agents think and act**.

2. Why LangGraph Exists

LangChain agents work well for simple loops, but they struggle with:

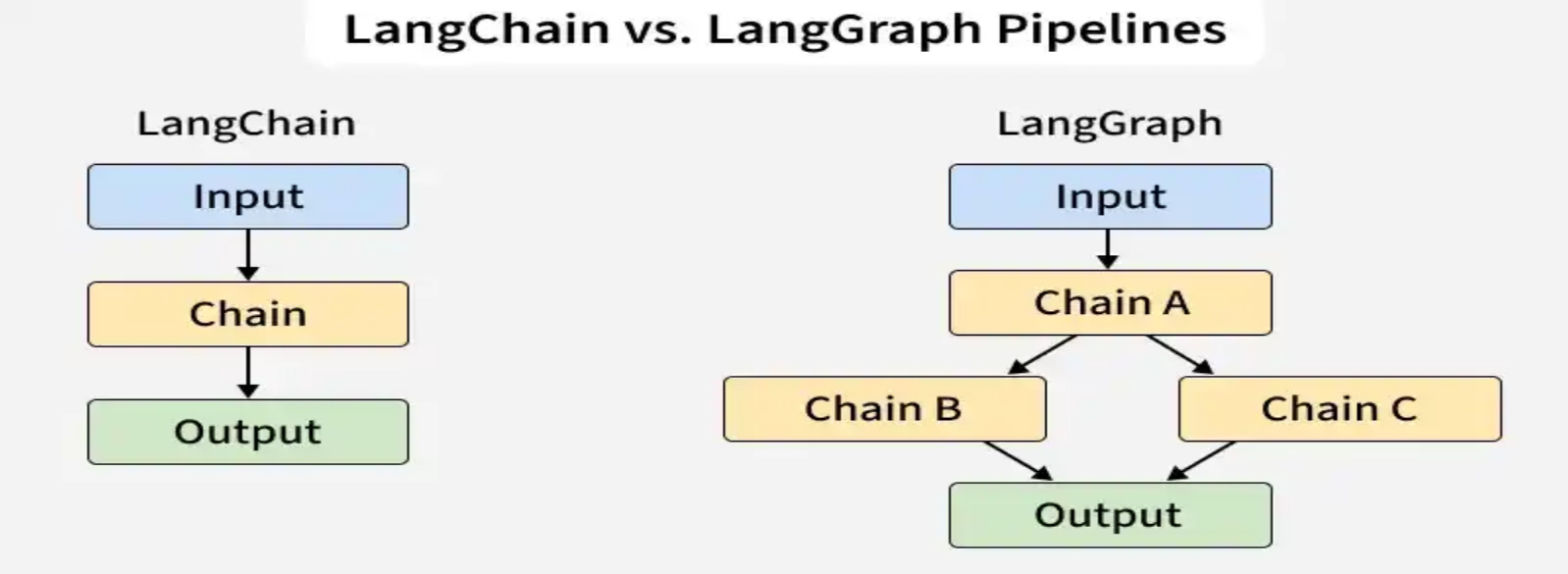
- Complex control flow
- Explicit state management
- Long-running, fault-tolerant agents
- Multi-agent systems

LangGraph solves this by making agent logic explicit and inspectable.

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangGraph

- LangGraph allows developers to define workflows as nodes and edges, making complex agent interactions more structured, scalable and easier to control.
  - Organises AI agent workflows using graph-based architectures
  - Supports both simple use cases (chatbots) and complex multi-agent systems
  - Integrates LLMs with external tools, APIs and memory
  - Enables modular and customizable workflow design



LangChain uses a straightforward step-by-step pipeline where each task follows a fixed sequence from input to output while LangGraph allows flexible workflows with branching and parallel steps, making it better suited for complex, decision-based applications.

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangGraph

Building a Simple Chatbot with LangGraph : LangGraph makes it easy to build structured, stateful applications like chatbots. In this example we'll learn how to create a basic chatbot that can classify user input as either a greet, search query and respond accordingly.

Step 1: Install the Dependencies

Installs the required dependencies,

- **langgraph:** Framework for building graph-based AI workflows.
- **langchain:** Popular toolkit for LLM-powered AI applications.
- **google-generativeai:** Google's API for Generative AI (Gemini models).

To know how to access Gemini API refer to : How to Access and Use Google Gemini API Key (with Examples)

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangGraph

Step 2: Setup Gemini API

- Imports the Google Generative AI Python SDK.
- Configures the API with our private key for authentication.
- Initializes the Gemini 1.5 Flash model for fast, multimodal LLM responses.
- Defines an ask_gemini function that takes a prompt (user question) and generates a response from Gemini and handles errors gracefully by returning an apologetic message if the API fails.

To know how to access Gemini API refer to : How to Access and Use Google Gemini API Key (with Examples)

```
import google.generativeai as genai

genai.configure(api_key="YOUR_API_KEY")

model = genai.GenerativeModel("gemini-1.5-flash")

def ask_gemini(prompt: str) -> str:
    try:
        response = model.generate_content(prompt)
        return response.text
    except Exception as e:
        return "Sorry, something went wrong with the Gemini API."
```

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangGraph

Step 3: Define Chatbot State

We will import Optional and TypedDict for strict type checking and creates a GraphState type:

- Holds the current question, its classification (greeting/search) and the final response.
- Ensures clarity and structure in state handling during workflow execution.

```
from typing import Optional
from typing_extensions import TypedDict

class GraphState(TypedDict):
    question: Optional[str]
    classification: Optional[str]
    response: Optional[str]
```

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangGraph

Step 4: Classify Input

We define classify, which takes the workflow state and analyzes the user's question.

- Checks if the question is a greeting. For example keywords like hi, hello, etc.
- Tags the question as either "greeting" or "search" for branching logic later.
- Returns the updated state with the new classification.

```
def classify(state: GraphState) -> GraphState:
    question = state.get("question", "").lower()
    if any(word in question for word in ["hello", "hi", "hey", "good morning", "good evening"]):
        classification = "greeting"
    else:
        classification = "search"

    return {
        **state,
        "classification": classification
    }
```

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangGraph

Step 5: Respond Using Gemini (or Greeting)

This define respond which generates appropriate output based on classification.

- For greetings, returns a friendly welcome message.
- For search questions, calls Gemini via ask_gemini and fetches an AI-generated answer.
- Handles unknown classifications with a safety fallback response.
- Updates and returns the state with the generated reply.

```
def respond(state: GraphState) -> GraphState:
    classification = state.get("classification")
    question = state.get("question")

    if classification == "greeting":
        response = "Hello! How can I help you today?"
    elif classification == "search":
        response = ask_gemini(question)
    else:
        response = "I'm not sure how to respond to that."

    return {
        **state,
        "response": response
    }
```

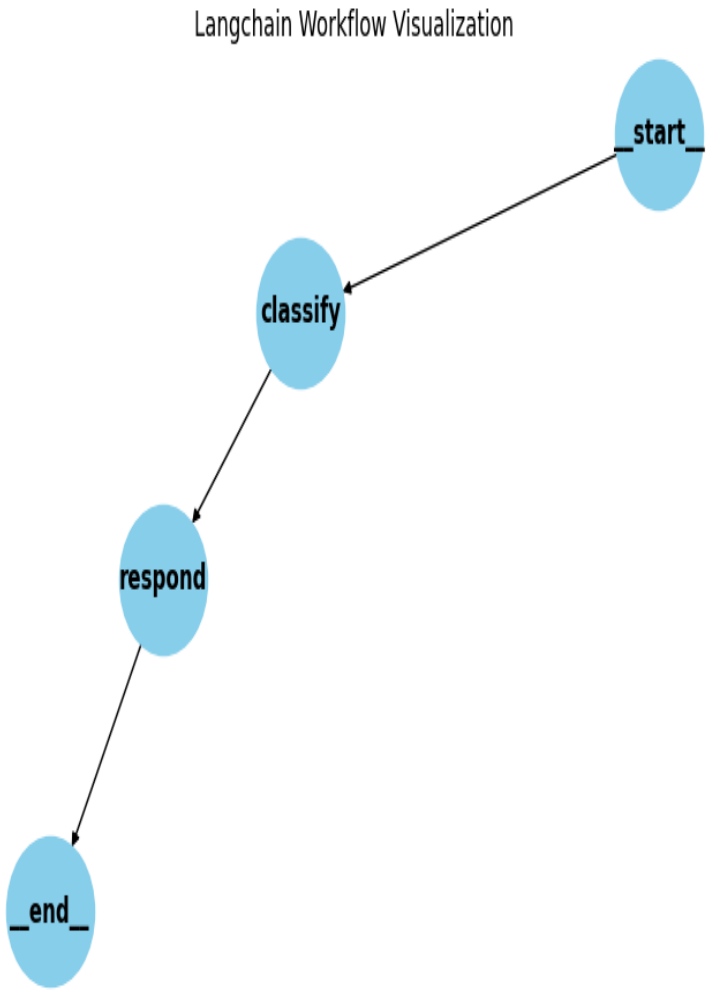
https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangGraph

Step 6: Build LangGraph Workflow

- Import tools for network graph creation and visualization.
- Build the workflow graph using LangGraph, adding nodes for classification and response, connecting them with edges and compiling the app.
- Include a function to visually display the workflow using networkx and matplotlib, aiding understanding and troubleshooting.

```python
import networkx as nx
import matplotlib.pyplot as plt
from langgraph.graph import StateGraph

builder = StateGraph(GraphState)
builder.add_node("classify", classify)
builder.add_node("respond", respond)
builder.set_entry_point("classify")
builder.add_edge("classify", "respond")
builder.set_finish_point("respond")
app = builder.compile()

def visualize_workflow(builder):
    G = nx.DiGraph()

    for node in builder.nodes:
        G.add_node(node)
    for edge in builder.edges:
        G.add_edge(edge[0], edge[1])

    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_size=3000,
            node_color="skyblue", font_size=12, font_weight="bold", arrows=True)

    plt.title("Langchain Workflow Visualization")
    plt.show()

visualize_workflow(builder)
```



https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

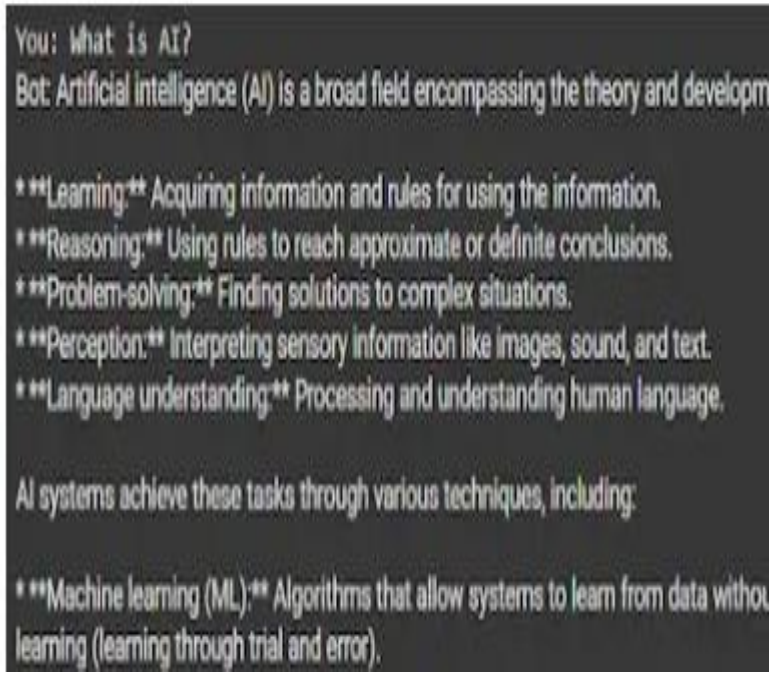
LangGraph

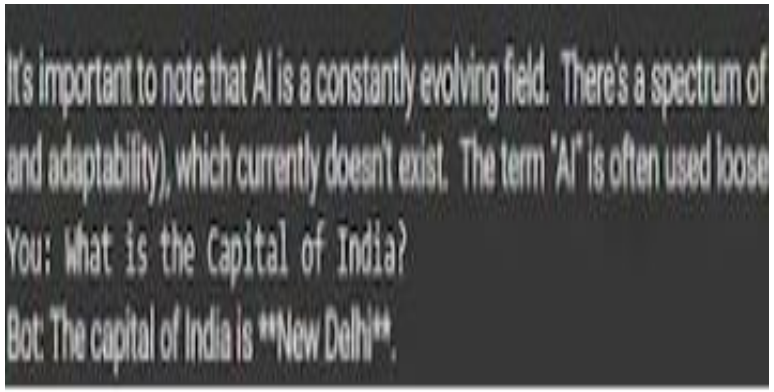
Step 7: Interactive Chat Interface

- Create a command-line chatbot that processes user inputs until “exit” or “quit” is typed.
- Send each input through the workflow graph and returns the bot’s response, either a greeting or an AI-powered answer.

```
print("=== Gemini-Powered Chatbot ===")
print("Type your question below. Type 'exit' to quit.\n")

while True:
    user_input = input("You: ")
    if user_input.strip().lower() in ['exit', 'quit']:
        print("Bot: Goodbye!")
        break

    state = {"question": user_input}
    result = app.invoke(state)
    print("Bot:", result["response"])
```

Output:





https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangChain vs LangGraph

Features	LangGraph	LangChain
Architecture	Graph-based (nodes and edges with memory and branching).	Sequential decision-act loop.
Workflow Control	Fully customizable paths, loops and conditions.	Limited control, follows predefined tool-usage cycle.
State Management	Built-in persistent state across the entire graph.	Implicit or external memory required.
Support for Loops	Yes, supports cyclical flows and iteration.	Not designed for loops or retries.
Human-in-the-Loop	Built-in support for pausing and resuming with human input.	Requires custom implementation.
Debugging and Observability	High observability with tools like LangSmith.	Limited transparency, harder to debug.

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

LangChain vs LangGraph

LangChain vs LangGraph (Quick Contrast)

Rule of thumb:

- *LangChain builds agent components*
- *LangGraph controls agent behavior*

When to Use LangGraph

Use LangGraph when you need:

- Long-horizon reasoning
- Reliable replanning
- Complex workflows
- Multi-agent collaboration
- Debuggable, auditable agent logic

https://www.geeksforgeeks.org/artificial-intelligence/introduction-to-langchain/

Thank You for Your Attention