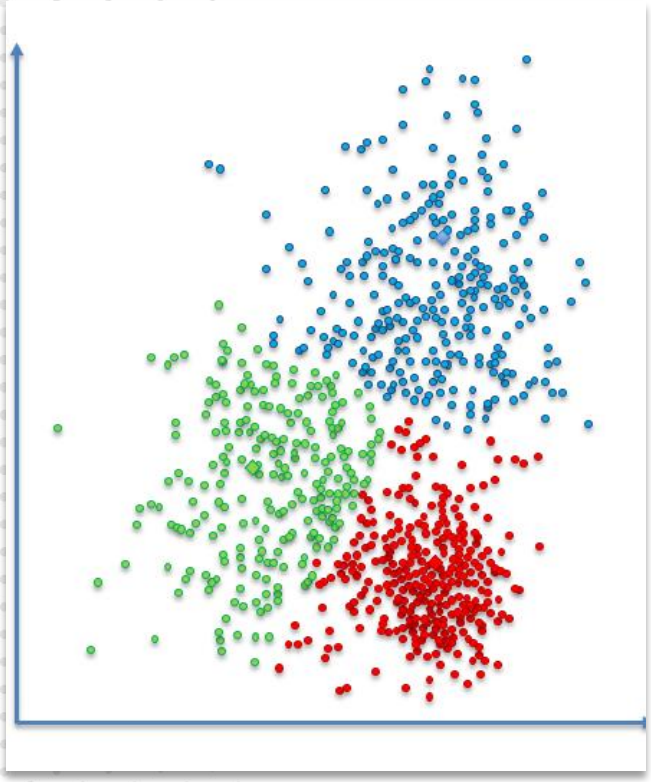
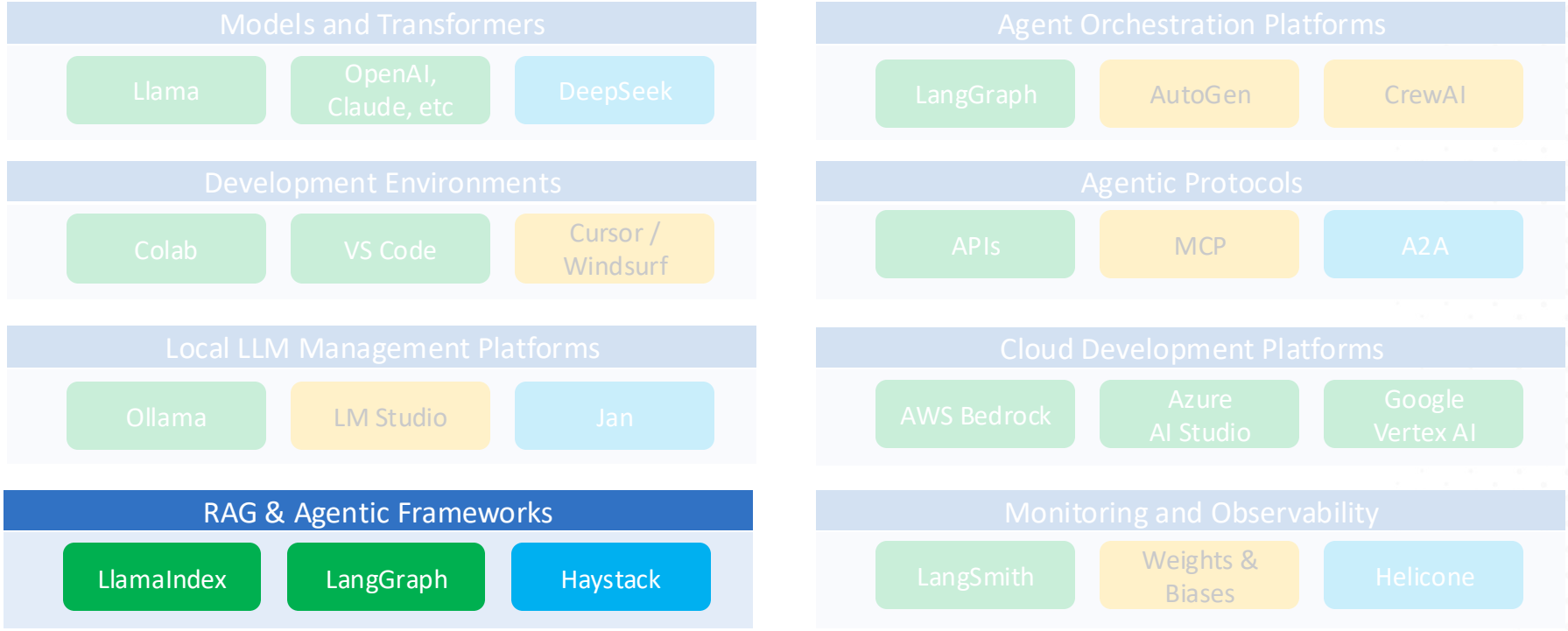


# Lesson 3:
# Agentic AI Development Tools

Objectives

- What is an agent, agent vs. workflow
- Agentic frameworks
    - LangChain/LangGraph, CrewAI, AutoGen, …
- A first manually coded agent in Python
- Multi-tool agent in Python
- Multi-tool agent with LangChain, Llamaindex and CrewAI
- Automated agent with LangChain
- Coding your own Agent

# RAG and Agentic Frameworks




Mature | Strong Contenders | Emerging

Pearson

# Agents vs. Workflows

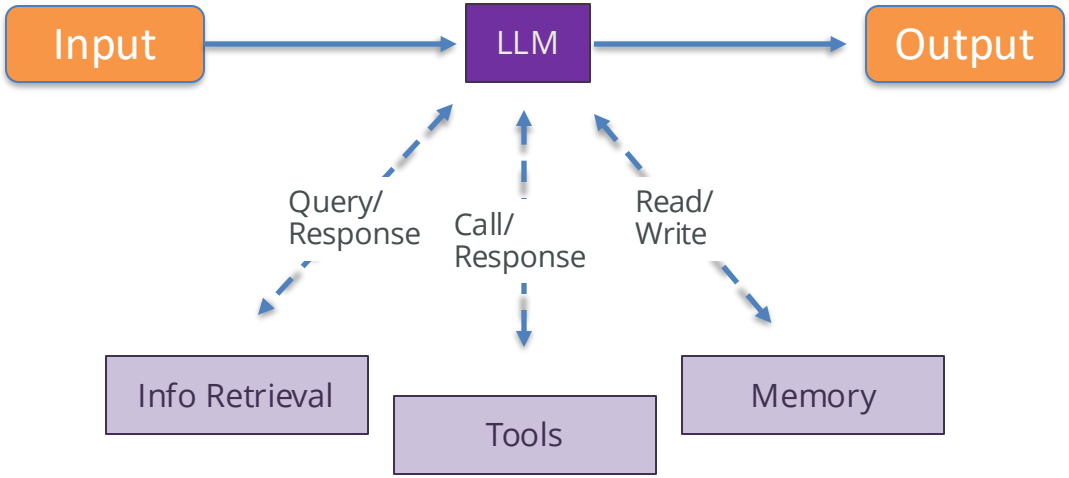
Terminology is fluid, but let's define:

- Workflow: predefined sequence of steps or logic that governs how tasks are performed. Often labeled "orchestrator" (governs how agents/tools are used)
- Agents: An autonomous decision-making entity that uses tools, memory, and planning to act. Sometimes labeled "actor"

- Workflows combine agents and their tasks
  - workflows are static (actions are defined at design time), usually in code, invoke agents and their actions
  - agents are dynamic (behavior decided at runtime), powered by LLMs to observe, reason, and act – more flexible but slower and with risk of compounding errors
    - Confusion risk: agents can be used to build or modify workflows!

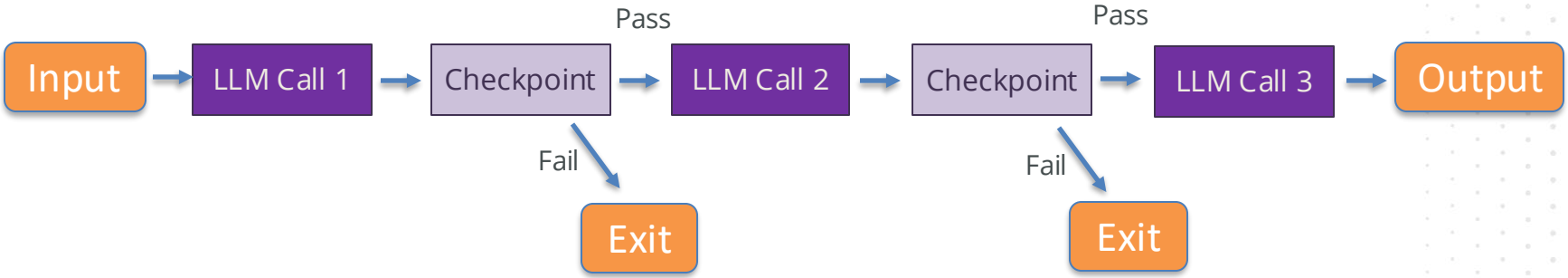
# Starting Point – Augmented LLM

- A basic LLM only interacts with you
- An Augmented LLM can access different elements (RAG, Internet, summary of previous exchanges etc.), directly or through APIs or connection platforms (e.g. MCP)



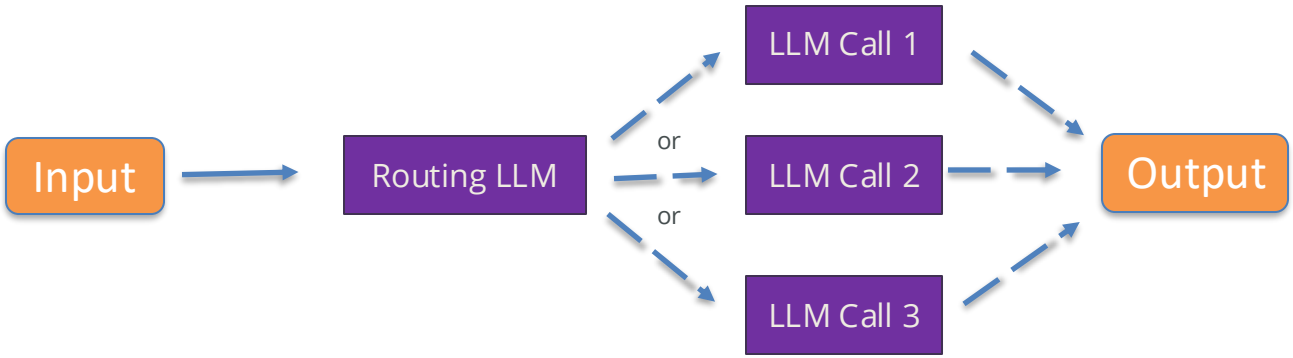
# Prompt Chaining Workflow

- Each LLM call processes the output of the previous one
- Programmatic checkpoints validate that the output is toward the end goal
  - Examples: generate a summary in languages 1, translate into language 2; find keywords in a text, generate definitions, etc.



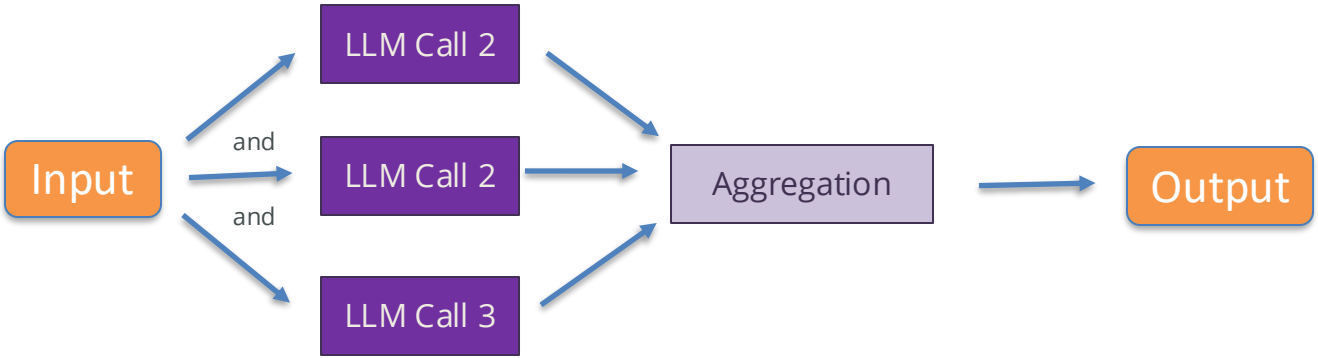
# Routing Workflow

- User input is processed by a first LLM (or LLM call)
- Keywords results in processing by specific second LLM
  - Example: customer service query (-> marketing, technical etc.)



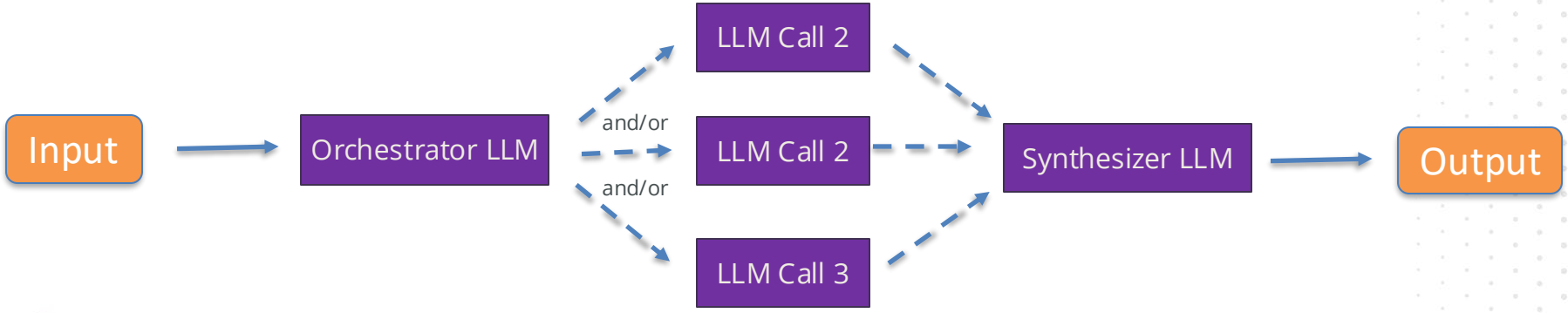
# Parallelization Workflow

- Several LLMs work in parallel on the same task:
  - Voting system (get outputs from several LLMs) – example: code debugging
  - Sectioning system (split task into subtasks)- example: guardrails (one LLM processes query, other checks for inappropriate content, aggregator applies 2 onto 1)



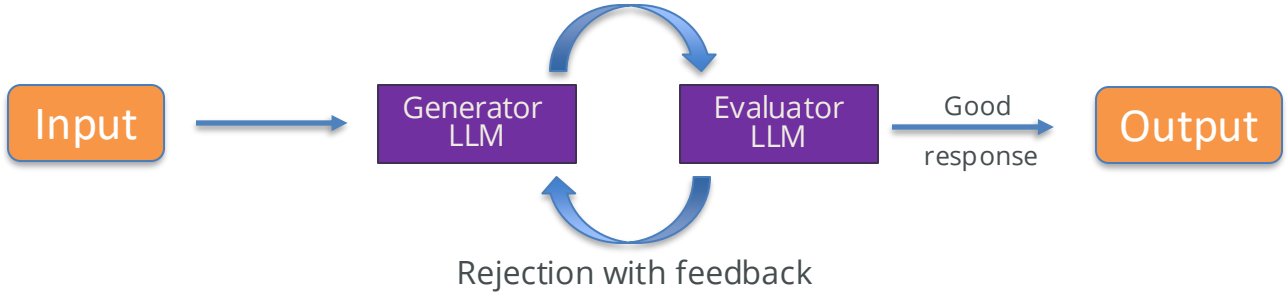
# Orchestrator/ Workers  Workflow

- A first (orchestrator) LLM breaks down the query into tasks, then sends to other LLMs
  - Similar to parallelization, but the breakdown is LLM-generated instead of static
  - Useful when next tasks depends on the query (e.g. Write a literature review on the efficacy of LLMs in clinical diagnostics -> break into radiology, pathology, workers deal with each individual field)



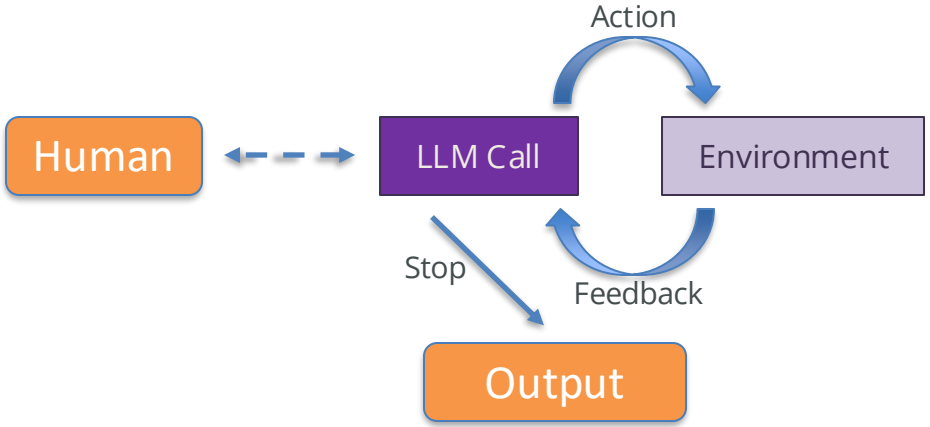
# Evaluator / Generator  Workflow

- Inspired by GAN:
  - First LLM produces an answer, evaluator LLM uses criteria to validate the output
  - Examples: movie storyboard, evaluator critiques age/plot etc.; technical plan writing, evaluator critiques plan coherence, step sequencing etc.



# Agent

- Often LLMs using tools with environmental feedback in a loop
- Start with command or instruction from human
  - Can then pause for feedback or when facing errors
  - Stop when task completes (or when condition, e.g. max iterations) is reached
- Easy to implement, even for complex tasks



# Agentic Landscape

- You can code agentic behavior by hand (Python)
  - Transparent, but manual context retrieval, no abstraction, hard to scale
- Frameworks help you scale, reduce the need for boilerplate code
- Multiple contenders

| Core Open Source<br><br>(foundation tools for devs) | Enterprise platforms<br><br>(scalable deployment & infra) | No-code builders<br><br>(fast prototyping for non-tech teams) |
|---|---|---|
| LangChain, LlamaIndex, AutoGen, CrewAI, … | Bedrock, Azure Agentic Tools, Vellum, … | Rivet, Flowise, Dust,… |

# Agentic with LangChain

- Purpose:
  - A general-purpose framework to build chains, agents, and tool-augmented LLM applications.
- Architecture:
  - Modular: Chains, tools, memory, retrievers, and agents are separate, composable elements.
  - Agents: Use a loop of reasoning → action → observation.
  - Tools: Can be any callable (Python function, API call).
  - Memory: Includes support for chat history, summarization memory, and custom memory modules.

# Agentic with LangChain

- Agent Capabilities:
  - Supports zero-shot agents, reactive agents, and conversational agents.
  - Tool calling via function schemas or OpenAI function calling.
  - Limited native support for multi-agent coordination (you have to wire it manually).
- Best Use Cases:
  - Building tool-augmented chatbots.
  - Creating custom pipelines for document Q&A, web scraping, or search.
  - Flexible, ideal for end-to-end systems involving RAG + tools + user memory.

# Agentic with LlamaIndex

- Purpose:
  - A data framework designed to make your data (documents, databases, APIs) accessible to LLMs.
- Architecture:
  - Core abstractions:
    - Documents
    - Nodes
    - Indices (vector, keyword, composable graphs)
    - Engines (query engines, agents)
  - Deep integration with LangChain, OpenAI, and Chroma, but can work independently.
  - Has support for agentic querying through "query transformers" and "retrieval agents"

# Agentic with LlamaIndex

- Agent Capabilities:
  - Data-aware agents: Query over documents, refine responses, navigate document graphs.
  - Supports tool use, RAG loops, and memory at the data interaction level.
  - Newer versions support task decomposition and planning agents.
- Best Use Cases:
  - Anything involving structured or semi-structured document search.
  - Retrieval-augmented generation (RAG) pipelines.
  - Adding memory and planning to document interactions.

# Agentic with CrewAI

- Purpose:
  - A lightweight but powerful multi-agent orchestration framework with roles, collaboration, and task flow.
- Architecture:
  - Core abstractions:
    - Agent: Each with a role, goal, tools, and backstory.
    - Task: Assigned to agents, can be sequential or parallel.
    - Crew: A team of agents, managed by an orchestrator.
  - Inspired by real-world workflows (engineer, reviewer, planner, etc.).

# Agentic with CrewAI

- Agent Capabilities:
  - Agents work sequentially or concurrently on a task list.
  - Each agent can call tools, query documents, or respond to others.
  - Supports orchestrator-worker, review loop, and role-based planning natively.
  - Minimal overhead: relies on LangChain or LlamaIndex tools.
- Best Use Cases:
  - Simulating collaborative teams (e.g., dev + reviewer + PM).
  - Implementing the Orchestrator–Worker and Evaluator–Optimizer workflows.
  - Fast prototyping of agent teams.

# Agentic with AutoGen (Microsoft)

- Purpose:
  - An advanced framework for multi-agent conversations, task-solving, and reflection loops.
- Architecture:
  - Core concepts:
    - Agents as functions: UserProxyAgent, AssistantAgent, CriticAgent, CodeAgent, etc.
    - GroupChat: agents communicate in turn-based fashion.
    - Self-reflection loop and auto-correction via “evaluator” roles.
  - Strong integration with OpenAI APIs and code evaluation.

# Agentic with AutoGen (Microsoft)

- Agent Capabilities:
  - Evaluator–Optimizer is native: agents can critique and revise in a loop.
  - Agents can delegate, revise, terminate, and rerun tasks.
  - Supports code execution, debugging, and memory reflection.
  - Easily model cooperative, adversarial, or supervisory agent interactions.
- Best Use Cases:
  - Complex workflows with feedback loops.
  - Code generation & debugging (e.g., agents that write and fix Python scripts).
  - Autonomous multi-agent simulation environments (research-oriented).

# Agentic LLM Frameworks Comparison

| Feature | LangChain | LlamaIndex | CrewAI | AutoGen |
|---|---|---|---|---|
| Agent type support | Chain & tool agents | Data agents | Role-based agents | Self-reflective agents |
| Multi-agent support | Manual | Limited | Native | Native (chat-based) |
| Tool Integration | Strong (modular) | Uses LangChain or native | Via LangChain tools | Native or custom tools |
| Maturity / Stability | High | High | Mid | Advanced / research grade |
| Best for | General AI pipelines, modular chains & tools-augmented agents | RAG & Data exploration, data-document retrieval and query agents | Team-like workflows, native multi-agent | Evaluator-optimizer loops, multi-agent coordination with evaluator optimizer loops |

# Simple Agent Logic

DEMO

- Agents are scripted LLM calls
  - Usually in a loop, until the LLM computes its answer as satisfactory
- The effect is an autonomous, goal-directed system that typically includes:
  - Perception – Interprets inputs or environment.
  - Planning – Chooses among actions or tools.
  - Action – Performs a task or responds to a user.
- Implementing these actions can be fairly basic, e.g.,:
  - Perception – agent loads RAG data (builds an internal representation of its world)
  - Planning – decides which chunk is most relevant
  - Action – answer the user question

# Enhanced Agent Logic

DEMO

- A basic agentic structure with RAG is agentic, but does not display all properties of agents, lacking:
  - Tool selection (e.g., fallback to Wikipedia or different retrievers).
  - Memory of past steps or dynamic adaptation.
  - Multi-step reasoning or looping behavior.
  - Explicit goals beyond answering a single question.
- We can improve by using a script and conditions to choose one tool or another:
  - Climate.txt if it includes info on climate, Wikipedia otherwise
  - This is even more agentic (planning/tool selection, evaluation then answer based on best sources)

# Enhanced Agent Logic with frameworks

**DEMO**

- Frameworks like LangChain, LlamaIndex, or CrewAI integrate agentic scripts – scripts that include the loop function, (and/or) the tool selection function, the memory of previous responses, an evaluator for the answers obtained etc.
- The effect is that a command replaces a full function or loop coded manually
- Each framework has specific focus -> is better/easier than the other for a particular agentic structure
  - All three can fulfill our basic demo requirements

# Deeper Agentic with LangChain

DEMO

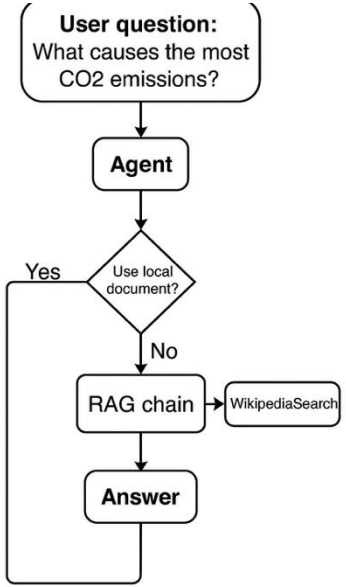
- LangChain includes many agentic functions, e.g.:

| Agent Type | Description | Pattern/Strategy Used |
|---|---|---|
| zero-shot-react-description | **ReAct-style agent** that picks a tool and uses its description to decide — no examples needed. | **ReAct (Reasoning + Acting)** |
| self-ask-with-search | LLM decomposes question and invokes a search tool for sub-questions. | **Self-Ask with Google**-like |
| structured-chat-zero-shot-react-description | ReAct agent that expects structured tool inputs and outputs. | Structured ReAct |
| conversational-react-description | Like ReAct but maintains conversational memory between steps. | ReAct + Chat Memory |
| openai-functions | Uses OpenAI's **function calling** system for tool execution. | OpenAI-native function calling |
| chat-zero-shot-react-description | Like zero-shot-react, but optimized for chat models. | ReAct for chat-optimized LLMs |
| plan-and-execute | Splits planning and acting: planner builds a plan, executor executes it step-by-step. | **Plan-then-Act** pattern (BabyAGI-inspired) |
| chat-conversational-react-description | Adds memory and chat context to structured ReAct for OpenAI chat models. | Memory-augmented ReAct |

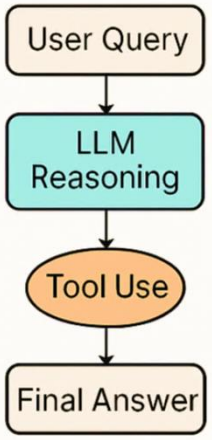
# Building a LangChain Agentic Workflow

**DEMO**

- Our workflow so far uses a zero-shot ReAct model
  - No prior example provided, tool selects tool, evaluates if it provides the answer, switch to another tool if needed, then provides the answer
- The LangChain Zero_shot_react agent provides the same function
  - No need to code the loop manually
  - +"agentic cool factor"




Agentic logic




**Zero-Shot ReAct**

- Chosen tool based on ReAct reasoning
- One-time reasoning + action

# LangChain Agentic Example

What happens in the background:

- When calling:

```
agent_executor.run("What causes the most CO2 emissions?")
```

- LangChain implements ReACT (Reason, then Act), and executes a loop coordinated by the AgentExecutor that looks like this:

```
while not done:
    # Step 1: Build input prompt for the LLM
    prompt = format_with_tools(
        input_question,
        tools_available,
        past_tool_uses_and_observations
    )
```

# LangChain Agentic Example

```
# Step 2: Send prompt to LLM and get output
   llm_output = llm(prompt)

# Step 3: Parse LLM output
   if output includes Action:
       run the specified Tool with Action Input
       save Observation
   elif output includes Final Answer:
       return that and break loop
   else:
       raise error
```

# LangChain Agentic Example

In practice, these steps are all handled by the class langchain.agents.AgentExecutor``` (you can print it with `agent_executor.agent.llm_chain.prompt.template`)

The LLM actually sees something like this (example):

```text
Answer the following question as best you can using the tools available.

You have access to the following tools:
Search: useful for finding information on the internet
Calculator: useful for math
```

# LangChain Agentic Example

```
Use the following format:

Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [Search,
Calculator]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Observation can repeat) ...
Thought: I now know the final answer
Final Answer: the answer to the original question

Begin!
```

# LangChain Agentic Example

```
Question: What is the square root of 256?
Thought: I need to use the calculator to find the square root.
Action: Calculator
Action Input: sqrt(256)
Observation: 16
Thought: I now know the final answer
Final Answer: 16
```

# LangChain Agentic Example

- With our agent definition:

```
agent_executor = initialize_agent(
    tools=[LocalDocQA, WikipediaSearch],
    llm=Ollama(model="llama3"),
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True
)
```

- LangChain creates a prompt template (ReAct-style), feeds the tool names + descriptions, sends to Ollama LLM, parses output to extract:
  - Action → matches it to a tool
  - Action Input → feeds that to the tool
  - Observation → appends result to prompt
- Loops until the LLM says “Final Answer: …”

# Building your own Agent

**DEMO**

- You can use LangChain, CrewAI, LlamaIndex etc. pre-made agents
- You can also use community resources to find agents that fit your needs:
  - https://www.langchain.com/hub (community-driven registry of LangChain components - chains, agents, prompts, toolkits)
  - (https://github.com/langchain-ai/langgraph plenty of LangGraph agents and examples, from simple routers to planner + executors + multi-agent collaboration grpahs, etc.)
- You can also create your own agent with LangGraph
  - Logic is simple: define states (what input/result the agent needs to consider), routing function (where to go next, depending on the state) and what action to take… until the agent gets the result it thinks you want