Foundations of Building AI Agents

Single LLM features: Simple, one-shot tasks

Imagine you want to quickly summarize a news article or translate a customer review. You simply input the text into a single LLM and instantly receive the summarized or translated output—no further steps required. At the most basic level, you can use LLMs for simple, single-turn tasks with no memory or context across calls.

Key characteristics

Single LLM features have the following key characteristics:

- **Stateless processing**: No retention of information or context across interactions.
- **Direct input-output flow**: Straightforward request-response mechanism.
- **Predefined tasks**: Suitable only for clearly defined, single-step actions.

Examples

This paradigm is appropriate for:

- Text summarization
- Sentiment classification
- Information extraction
- Translation

Advantages

Using single LLM features offers:

- **Speed and simplicity**: Fastest to build and run
- **Deterministic output**: Same input, same output
- **Low cost**: Minimal compute and orchestration overhead

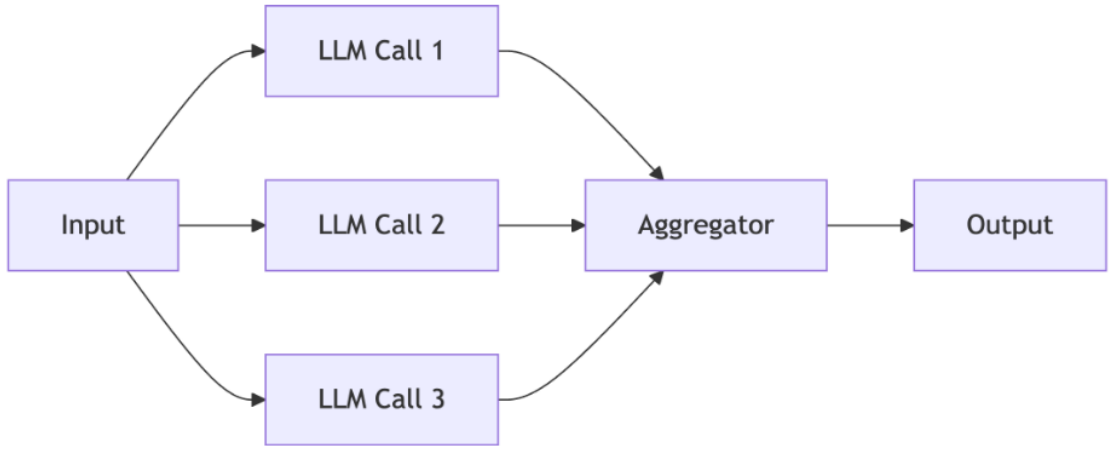
Limitations

When using a single LLM, you will likely encounter the following limitations:

- **No adaptability**: Cannot handle context or dynamic decision-making
- **No memory**: Each input is processed independently

Structured workflows: Multi-step, predictable processes

Structured workflows orchestrate LLM and tool calls through explicit, deterministic code paths. They're ideal for repetitive, multi-step, or compliance-heavy tasks. Consider processing insurance claims, where each document is scanned, information is extracted, validated, and stored. Each step must follow a precise, predictable order, making structured workflows ideal.



Key characteristics

Structured workflows have the following key characteristics:

- **Deterministic execution**: Inputs produce consistent outputs.
- **Explicit control flow**: All steps and decisions are predefined.
- **Predefined tool chains**: Tool use is fixed and transparent.

Best uses

Structured workflows work well for the following needs:

- Repetitive, multi-step tasks with clear logic and minimal ambiguity
- Regulatory or compliance-driven applications
- Scenarios requiring consistency, traceability, and auditability

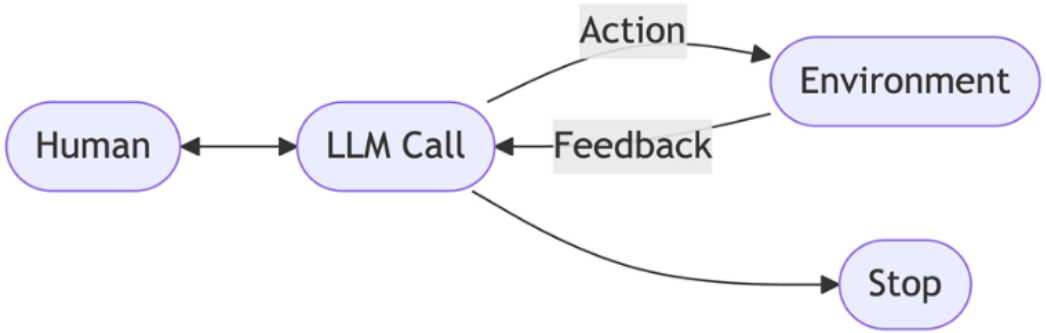
Examples

You'll find structured workflows work well for the following scenarios:

- Document and data pipelines (Optical Character Recognition (OCR) → extraction → validation → storage)
- Batch report generation
- Financial and healthcare transaction processing

Autonomous agents: Flexible, context-aware reasoning

Autonomous agents allow LLMs to plan sequence actions and adapt as conditions change. Agents choose which tools to use and how to achieve their goals based on real-time context and feedback. Imagine an AI-driven virtual assistant helping a user plan a vacation. It dynamically gathers user preferences, researches destinations, suggests accommodations, and adapts recommendations based on feedback. This requires an autonomous agent capable of planning, context-awareness, and iterative improvement.



Core capabilities

Autonomous agents have the following core capabilities:

- **Dynamic planning**: Decomposes goals and adjusts steps as needed
- **Contextual awareness**: Remembers past steps and adapts to user and environment feedback
- **Tool orchestration**: Selects tools and changes strategies dynamically

Best uses

You can use autonomous agents for the following needs:

- Complex, open-ended tasks with unclear solution paths
- Scenarios requiring real-time adaptation and reasoning
- Environments with high variability or need for personalization

Examples

Consider implementing autonomous agents for the following uses:

- Research agents synthesizing new information
- Adaptive customer support and troubleshooting
- Automation that iteratively refines results based on feedback

Advantages

Autonomous agents offer the following advantages:

- **Highly adaptable**: Handles unforeseen situations
- **Dynamic decision-making**: Iterates and improves over time
- **Reduces human intervention**: Manages complexity autonomously

Limitations

Autonomous agents can also encounter the following challenges:

- **Unpredictable outcomes**: Requires robust monitoring and safeguards
- **Higher complexity and cost**: More difficult to debug and guarantee compliance

Summary table

The following table compares three AI systems, their processes, use cases, and pros and cons.

AI System type	Process	Use Case	Pros	Cons
Single LLM	Input → LLM → Output	Summarization, classification	Simple, fast, low cost	Not adaptable, lacks context
Workflow	Parallel LLMs → Aggregation → Output	Structured multi-step tasks	Predictable, easy to audit	Rigid, not dynamic
Agent	Plan → Act → Observe → (repeat agent loop)	Complex, adaptive automation	Flexible, learns from feedback	Unpredictable, complex, pricier

Real-world implementation practices

In practice, hybrid architectures are common. They combine workflow reliability with agent flexibility to achieve the best results.

Recent standards, including Model Context Protocol, or MCP, from Anthropic, and Agent Communication Protocol, or ACP, from IBM, ease integration, monitoring, and governing both approaches at scale.

Key takeaways

When selecting an AI agent, reflect on the following considerations:

Recap

You now know that:

- **AI agents** sit at the highest end of the AI complexity spectrum, excelling at tasks that require autonomous decision-making, adaptation, and strategy.
  Use the four-step decision framework—task ambiguity, step flexibility, tool variety, and failure impact—to decide if an agent is the right fit for the task.
- **Avoid using agents** for simple, repeatable, or high-risk tasks where errors are costly or predictable; tools can perform better.
- **Today's agents** struggle with reliability, high compute costs, and often need human oversight to avoid hallucinations or missteps.
- **Manage risk** by setting boundaries, using logs, monitoring outcomes, and keeping a human-in-the-loop for oversight.
- **Effective agent architecture** includes modular components like memory, tool use, planning strategies, and clear reasoning paths.

LLMs without tools

Tools:

- Allow LLMs to interact with real world
- Perform math, retrieve facts, and take actions
- Help move from ‘text generation’ to ‘problem solving’

Capabilities provided by tools

Access private data

- Retrieval of information unavailable in training data
- Enables retrieval-augmented generation (RAG)

Process multiple modalities

- Analyzes images, audio, and non-text inputs
- Enables vision capabilities, voice understanding, and multimodal reasoning

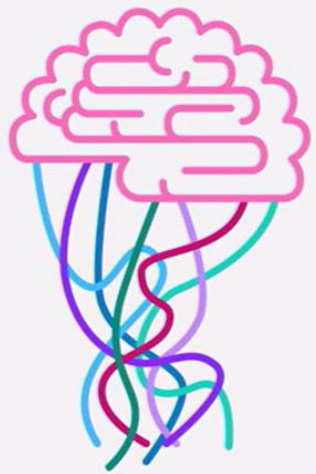
Overcome LLM limitations

- Extends beyond built-in constraints
- Maintains conversation memory across sessions
- Processes information that exceeds context window size

Control external systems

- Interacts with APIs, software, and digital services
- Allows the LLM to take actions

Hallucinations



LLMs without tools:

Rely on patterns in training data

Lead to hallucinations

Are noticeable in tasks like math or logic

Introducing tools

LLM tools

- Help take actions beyond text generation
- Interact with real-world data
- Perform complex tasks
- Extend the AI’s capabilities

Introducing tools

No more guessing with a calculator tool

Expanding capabilities

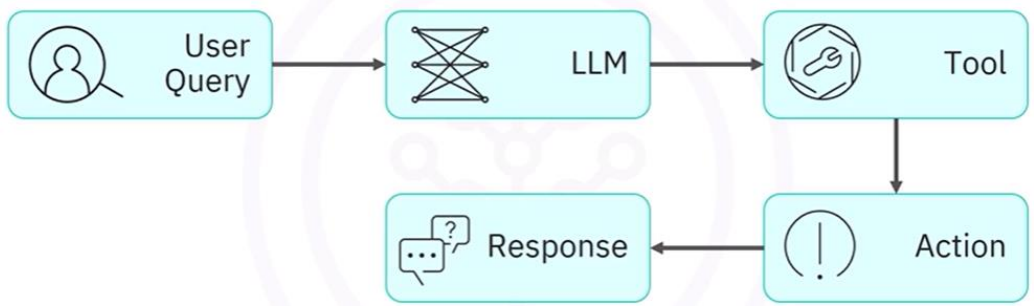
Web tool:
Fetch real-time weather

Code tool:
Write and execute Python code

Search tool:
Look up recent news

SQL tool:
Query data from a business database

The agentic future



Recap

- Understand the limitations of LLMs without tools and the impact of missing tools on accuracy
- Identify the powerful capabilities that tools provide to LLMs, including accessing private data, processing multiple modalities, overcoming LLM limitations, and controlling external systems
- Recognize how tools transform LLMs from guessers into intelligent agents capable of interacting with real-world data
- Explore how different tools, like calculators, web access, and databases, enable LLMs to perform tasks beyond text generation
- Learn how the agentic process works—LLMs selecting the right tool, taking action, and providing meaningful responses
- Understand how tools are essential for improving precision and moving from “text generation” to “problem solving”

Introduction

Empower LLMs with tools

- Access live data
- Perform actions
- Ensure mathematical and logical precision
- Retrieve private or enterprise-specific data
- Enhance multistep reasoning

What is a tool

- Does one job well
- Is a Python function
- Debugging is more art than science
- LangChain evolves rapidly, so version control is essential
- LLMs and agents differ in how they handle inputs and outputs
- Certain combinations require specific formats to work reliably

Understand the LLM's response

- Inputs a query
- Extracts the parameters
- Determines which tool to call
- Passes the extracted inputs
- Receives the output

Anatomy of a tool

A descriptive name: Use intuitive names such as `add_numbers`

```
def add_numbers(inputs: str) -> dict:
    """
    Tool: add_numbers
    Description: Adds numeric values extracted from a string input.
    Input: str (e.g., "Add 10, 20 and 30")
    Output: dict (e.g., {"result": 60})

    """
    numbers = [int(x) for x in inputs.replace(",", "").split() if x.isdigit()]
    result = sum(numbers)
    return {"result": result}
```

Anatomy of a tool

Standardized input: Use an easy-to-parse input such as a text string or JSON

```
def add_numbers(inputs: str) -> dict:
    """
    Tool: add_numbers
    Description: Adds numeric values extracted from a string input.
    Input: str (e.g., "Add 10, 20 and 30")
    Output: dict (e.g., {"result": 60})

    """
    numbers = [int(x) for x in inputs.replace(",", "").split() if x.isdigit()]
    result = sum(numbers)
    return {"result": result}
```

Anatomy of a tool

Comprehensive documentation: What the tool does, expected input/output, and known limitations

```
def add_numbers(inputs: str) -> dict:
    """
    Tool: add_numbers
    Description: Adds numeric values extracted from a string input.
    Input: str (e.g., "Add 10, 20 and 30")
    Output: dict (e.g., {"result": 60})

    """
    numbers = [int(x) for x in inputs.replace(",", "").split() if x.isdigit()]
    result = sum(numbers)
    return {"result": result}
```

Anatomy of a tool

Function body: The code that locates digits, converts digits to integers, and calculates their sums

```
def add_numbers(inputs: str) -> dict:
    """
    Tool: add_numbers
    Description: Adds numeric values extracted from a string input.
    Input: str (e.g., "Add 10, 20 and 30")
    Output: dict (e.g., {"result": 60})

    """
    numbers = [int(x) for x in inputs.replace(",", "").split() if x.isdigit()]
    result = sum(numbers)
    return {"result": result}
```

Anatomy of a tool

Consistent output: Returns output in a predictable format, usually a dictionary

```
def add_numbers(inputs: str) -> dict:
    """
    Tool: add_numbers
    Description: Adds numeric values extracted from a string input.
    Input: str (e.g., "Add 10, 20 and 30")
    Output: dict (e.g., {"result": 60})
    """
    numbers = [int(x) for x in inputs.replace(",", "").split() if x.isdigit()]
    result = sum(numbers)
    return {"result": result}
```

Tool interface

Specify the tool's name, function, and a short description

```
from langchain.agents import Tool

add_tool=Tool(
    name="AddTool",
    func=add_numbers,
    description="Adds a list of numbers and returns the result.")

print("tool object",add_tool)
```

The invoke method

Input: "What is the sum of 10, 20, and 30?"

```
test_input = "what is the sum of 10, 20 and 30"

print(add_numbers.invoke(test_input))
{'result': 60}
test_input = "what is the sum of ten, 20 and 30"
print(add_numbers.invoke(test_input))

{'result': 50}
```

The @tool decorator

`@tool` decorator defines tools in modern LangChain applications

```
from langchain_core.tools import tool
import re
@tool
def add_numbers(inputs:str) -> dict:

    """
    Adds a list …
    """

    # Use regular expressions to extract all numbers from the input
    numbers = [int(num) for num in re.findall(r'\d+', inputs)]
    result = sum(numbers)
```

Create a structured tool in LangChain

Args: Defines the expected input schema

```
print("Name: \n", add_numbers.name)
'''add_numbers'''

print("Description: \n", add_numbers.description)
'''
Description:
Adds a list of numbers provided in the input string.
Parameters:
inputs (str):
String, it should contain numbers that can be extracted and summed.
'''

print("Args: \n", add_numbers.args)

'''Args: {'inputs': {'title': 'Inputs', 'type': 'string'}}'''
```

Create a structured tool in LangChain

Each key corresponds to a parameter and the value parameter's input

```
add_numbers_with_options.invoke({"numbers":[-1.1,-2.1,-3.0],"absolute":False})

{'result': -6.2}


add_numbers_with_options.invoke({"numbers":[-1.1,-2.1,-3.0],"absolute":True})


{'result': 6.2}
```

Create a structured tool in LangChain

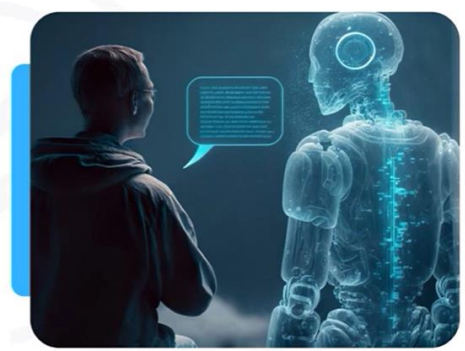
Specify the return type

```
from typing import Dict, Union
@tool
def sum_numbers_with_complex_output(inputs: str) -> Dict[str, Union[float,str]]:
```

Final thoughts

LangChain

- Tools are essentially Python functions
- Evolves rapidly, making version control essential
- LLMs and agents handle inputs differently



Recap

- Distinguish LLMs from agents by exploring the critical role of tools in enabling real-world interactions
- Extend LLM capabilities by integrating tools for data access, precise calculations, and multi-step reasoning
- Design effective tool interfaces with clear inputs, outputs, and descriptions to enhance agent responses
- Build structured tools for flexible, context-aware interactions, supporting complex inputs and robust data handling

Agents and tools

Factors to consider:

1. The LLM must support tool use.
2. Tools must have JSON-serializable inputs and outputs.
3. The agent strategy should support complex tasks.

Agent design often requires hands-on experimentation.

Load the LLM

- Initialize the LLM with Granite
  - Ask the LLM a question
  - Pair the LLM with tools

```
response = llm.invoke("What is tool calling in langchain ?")

print("\nResponse Content: ", response.content)
   """
       Response Content: In the context of Langchain, a tool calling refers
to..
   """
```

initialize_agent

Starting point for building agents

Choose from multiple predefined agents

Helps combine an LLM with tools

How agents work

Agents follow a standard reasoning loop

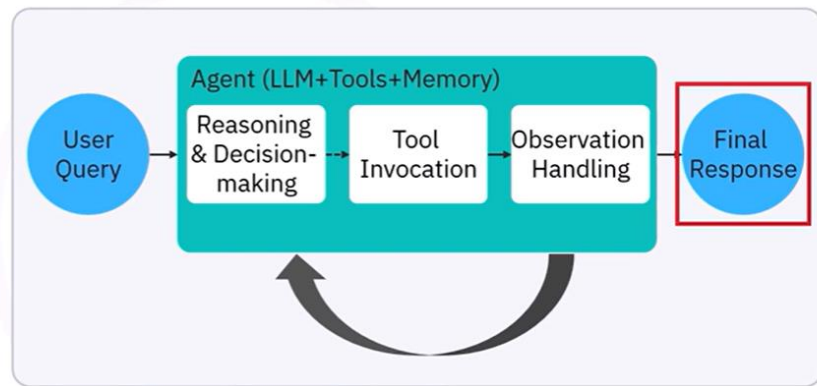
Decides which tool to call

Calls the selected tool

Agent decides what to do next

Feeds the result back into itself

Agent generates the output



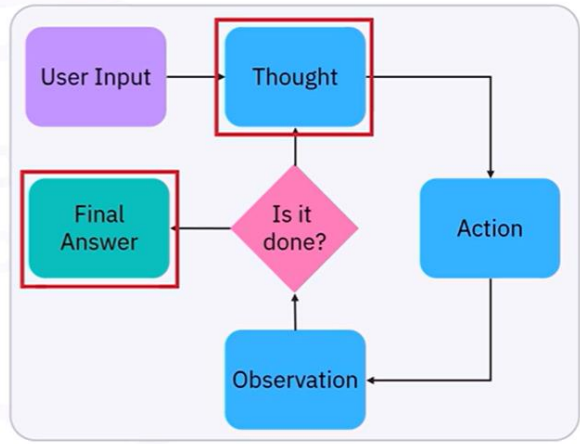
The ReAct agent framework

Reasoning: Thinking through the problem step by step

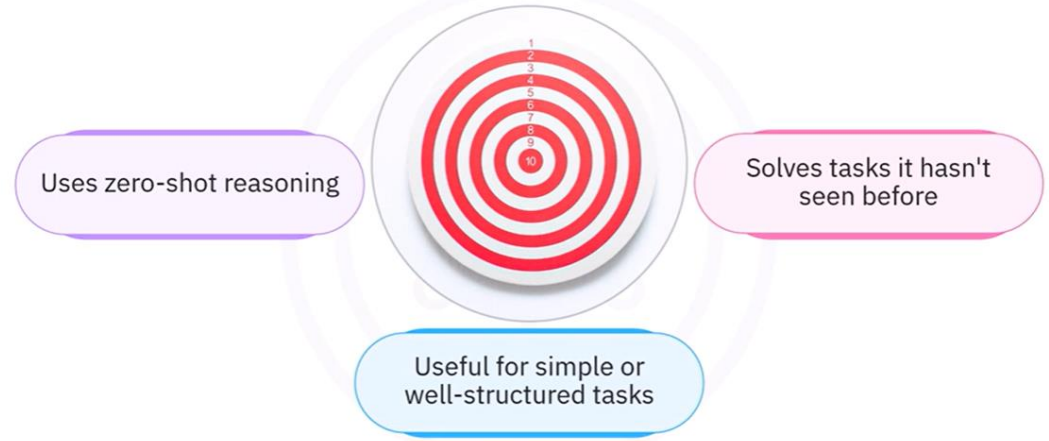
Acting: Using tools to gather information or perform operations

Observing: Processing the results from tools

Planning: Deciding next steps based on observations



Zero-shot ReAct agent



Zero-shot ReAct agent

- Wrap `add_numbers`
- Make the tool usable by the agent

```
from langchain.agents import Tool

add_tool=Tool(
    name="AddTool",
    func=add_numbers,
    description="Adds a list of numbers and
    returns the result.")

print("tool object",add_tool)
```

Zero-shot ReAct agent

- Create a zero-shot ReAct agent
- Import `initialize_agent`
- Pass a list of tools
- Use the ReAct strategy
- Set `verbose=True`

```
from langchain.agents import initialize_agent

agent = initialize_agent([add_tool], llm, agent="zero-shot-react-description",
    verbose=True, handle_parsing_errors=True)
```

Zero-shot ReAct agent

- Call the agent using the run method

```
response = agent.run("In 2023, the US GDP was approximately $27.72
trillion, while Canada's was around $2.14 trillion and Mexico's was about
$1.79 trillion. What is the total?")
```

- Entering new AgentExecutor chain... To find the total GDP of the US, Canada, and Mexico in 2023, I need to add their individual GDPs together.

- Action: AddTool
- Action Input: {"numbers": [27720000000, 2140000000, 1790000000]}

- Observation: The result is 3165000000000
- Observation: {'result': 3165000000000}

- Final Answer: The total GDP of the US, Canada, and Mexico in 2023 was approximately $31.65 trillion. > Finished chain.

Structured tool definition

Uses `add_numbers_with_options`

- Handles multiple inputs, including an optional argument
- Ensures flexibility and clarity in tool interactions

```
from typing import List

@tool

def add_numbers_with_options(numbers: List[float],
    absolute: bool = False) -> float:

    """

    Adds a list of numbers provided…

    """
```

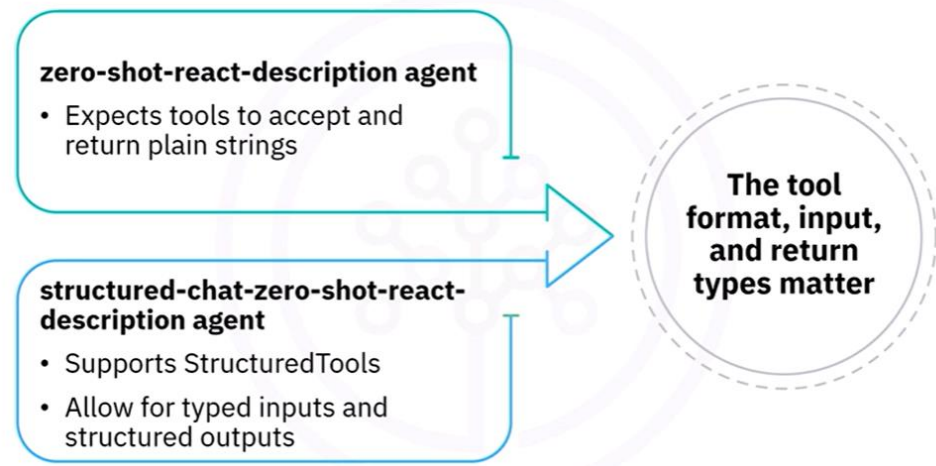
Zero-shot ReAct agent

Uses `.invoke()`

- Useful for debugging
- Working with more complex agents

```
response=agent.invoke({"input": "Add -10, -20, and -30 using absolute values."})

response: {{'input': 'Add 10, 20, and 30 ?',

           'output': 'The sum of 10, 20, and 30 is 60'}}
```

Structured-chat-zero-shot-react-description agent



Structured zero-shot ReAct agent

Set agent as "structured-chat-zero-shot-react-description"

```
agent_2 = initialize_agent([add_numbers_with_options], llm,
    agent="structured-chat-zero-shot-react-description",
    verbose=False, handle_parsing_errors=True)

response = agent_2.invoke({"input": "Add -10, -20, and -30 using absolute
values"})

response:

{'input': 'Add -10, -20, and -30 using absolute values.',
```

Experiment with different LLMs

- `sum_numbers_with_complex_output()` might cause issues
- Use an agent that supports structured outputs

```
from typing import Dict, Union
import re

@tool

def sum_numbers_with_complex_output(inputs: str)
    -> Dict[str, Union[float, str]]:

    """
    Extracts and sums all integers and decimals from the input

    """
```

Experiment with different LLMs

- Create an openai functions agent and set the agent="openai-functions"
- Change agent to `structured-chat-zero-shot-react-description` for **Granite**

```
llm_ai = ChatOpenAI(model="gpt-4.1-nano")

agent_openai = initialize_agent([add_numbers_with_options], llm_ai,
    agent="openai-functions", verbose=True)

response = agent_openai.invoke({"input": "Add -10, -20, and -30 using absolute
values"})

response:

{'input': 'Add -10, -20, and -30 using absolute values.',
'output': 'The sum of -10, -20, and -30 using their absolute is 60.'}
```

Recap

- Distinguish LLMs from agents by exploring the critical role of tools in enabling real-world interactions
- Extend LLM capabilities by integrating tools for data access, precise calculations, and multi-step reasoning
- Design effective tool interfaces with clear inputs, outputs, and descriptions to enhance agent responses
- Build structured tools for flexible, context-aware interactions, supporting complex inputs and robust data handling

Using the create_react_agent

The ReAct Agent

- Define a structured tool

```
@tool
def sum_numbers_with_complex_output(inputs: str) -> Dict[str, Union[float,str]]:
    """
    Extracts and sums all integers and decimals from the input …
    """
    matches = re.findall(r'-?\d+(?:\.\d+)?', inputs)
    if not matches:
        return {"result": "No numbers found in input."}
    try:
```

The ReAct Agent

1. Create a ReAct-style agent in LangGraph
2. Use the `create_react_agent` function
3. Create the agent object using `create_react_agent`
4. Pass in the LLM and a list of tools
5. Include a custom prompt to guide the agent's behavior

```
from langgraph.prebuilt import create_react_agent

add_agent = create_react_agent(
    model=llm,
    tools=[sum_numbers_with_complex_output],
    prompt="You are a helpful mathematical assistant that can perform
various operations. Use the tools precisely and explain your reasoning
```

The ReAct Agent

- Use the `.invoke()` method
- Process the input
- Use tools as needed
- Append the response to the message list

```
response = add_agent.invoke(
    {"messages": [("human", "Add the numbers -10, -20, -30")]}
)

# Get the final answer
final_answer = response["messages"][-1].content

final_answer: {"result": -60}
```

Orchestrating multiple tools to build a Math Toolkit

A single tool often isn't enough

Some tasks require different tools

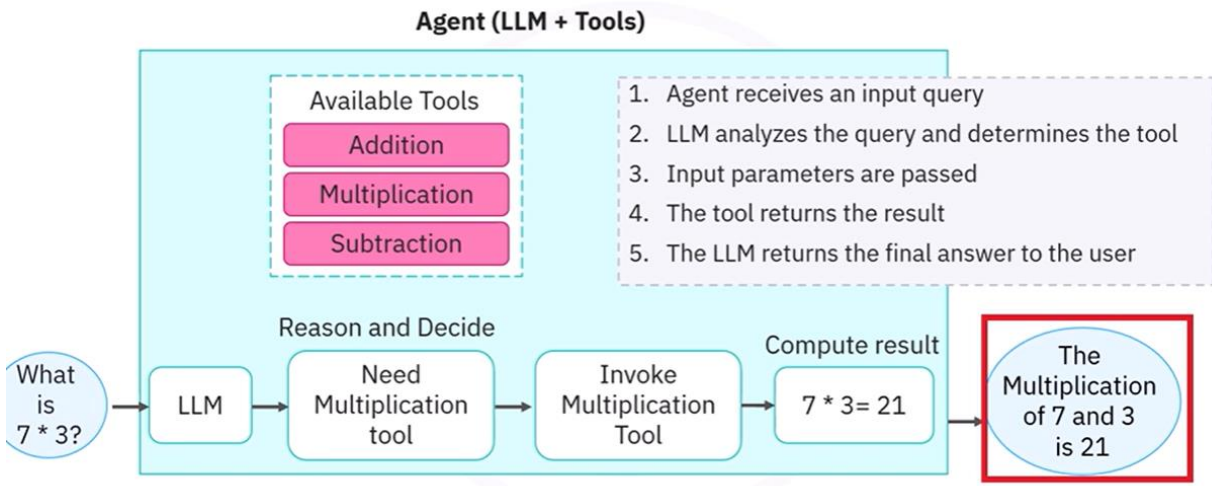
Example: Banking transactions
- Deposits
- Withdrawals
- Transfers

Orchestrating multiple tools to build a Math Toolkit

1. Create a ReAct agent
2. Add all the tools to a list
3. Use `create_react_agent()`
4. Include an optional prompt

```
from langgraph.prebuilt import create_react_agent

tools= [add_numbers, subtract_numbers, multiply_numbers, divide_numbers]

math_agent_new = create_react_agent(model=llm, tools=tool,

     # Optional: Add a system message to guide the agent's behavior

     prompt="You are a helpful mathematical assistant that can perform various operations. Use the tools precisely and explain your reasoning clearly.")
```

Agent Workflow



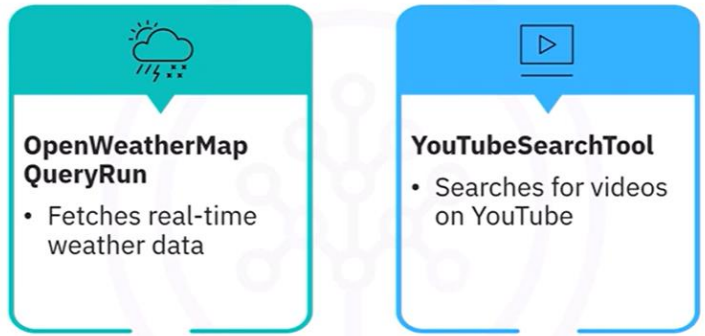
LangChain's built-in tools

WikipediaQueryRun
- Searches Wikipedia for factual information

GoogleSearchRun
- Performs web searches using Google's API

PythonREPLTool
- Runs Python code safely

LangChain's built-in tools



LangChain's built-in tools

- Create a custom Wikipedia search tool using `@tool` decorator
- Define clear input and output types
- Include a detailed docstring
- Wrap the `WikipediaQueryRun` object

```
    Returns:
    - str: A summary of relevant information from Wikipedia
    """
    wiki = WikipediaAPIWrapper()
    return wiki.run(query)
```

LangChain's built-in tools

Use `.invoke()` to search Wikipedia

A summary is the output

```
search_wikipedia.invoke("What is tool calling?")
```

```
'''
Page: Cold calling\nSummary: Cold calling is the solicitation of business from potential customers who have had no prior contact with the salesperson conducting the call. It is an attempt to convince potential customers to purchase the salesperson\'s product or service. Generally, it is an over-the-phone…
'''
```

LangChain's built-in tools

- Create a new agent
- Create an updated tools list

```
tools_updated = [add_numbers, subtract_numbers,
multiply_numbers,
    divide_numbers, search_wikipedia]
```

```
math_agent_updated = create_react_agent(
    model=llm,
    tools=tools_updated,
    prompt="You are a helpful assistant that can perform various
```

LangChain's built-in tools

- Create a new agent
- Create an updated tools list
- Use `create_react_agent` to build the new agent

```
math_agent_updated = create_react_agent(
    model=llm,
    tools=tools_updated,
    prompt="You are a helpful assistant that can perform various
mathematical operations and look up information. Use the tools precisely
and explain your reasoning clearly.")
```

LangChain's built-in tools

`.invoke()`

- Tests the updated agent with a query
- Provides the input
- Uses `search_wikipedia` to find Canada's population
- Applies the multiplication tool
- Returns the result

```
1. Convert the percentage to a decimal: 0.75
2. Multiply the population by the decimal: 36,991,981 * 0.75
So, the result is:
{"result": 27,743,985.75}
```

Recap

- Build ReAct-style agents using the `create_react_agent` method for greater customization and control
- Construct a multi-tool math assistant by combining tools for addition, subtraction, multiplication, and division
- Guide agent behavior using custom prompts and structured tool inputs
- Orchestrate multiple tools within a single agent to handle real-world, multi-step queries
- Extend agent functionality by integrating external tools such as Wikipedia search for dynamic, hybrid responses

- Use simple LLM features for basic tasks, use workflows for predictable and efficient operations, and deploy agents only when complex reasoning or adaptability is needed.
- AI agents sit at the highest end of the AI complexity spectrum, excelling at tasks that require autonomous decision-making, adaptation, and strategy.
- Use the four-step decision framework—task ambiguity, step flexibility, tool variety, and failure impact—to decide if an AI agent is the right fit for the task.
- Avoid using AI agents for simple, repeatable, or high-risk tasks where errors are costly or predictable; tools can perform better.
- Today's AI agents struggle with reliability, high compute costs, and often need human oversight to avoid hallucinations or missteps.
- Manage risks associated with AI agents by setting boundaries, using logs, monitoring outcomes, and keeping a human in the loop for oversight.
- Effective AI agent architecture includes modular components like memory, tool use, planning strategies, and clear reasoning paths.Tool calling enhances LLM capabilities by connecting them to real-time external data and functionality.
- Embedded tool calling improves LLM accuracy and reduces hallucinations by centralizing tool handling within a dedicated library or framework, replacing error-prone client-side implementations.
- Tools help LLMs access external data and support RAG, enabling the use of the organization's or other specialized databases.
- Tools help process images, audio, and video to enable vision, voice, and multimodal reasoning, manage long conversations, and connect to APIs to perform real-world actions.
- The Zero-Shot ReAct Agent uses zero-shot reasoning to solve tasks it hasn't seen before and works best for simple or well-structured problems.

What is LCEL?

LangChain Expression Language or LCEL

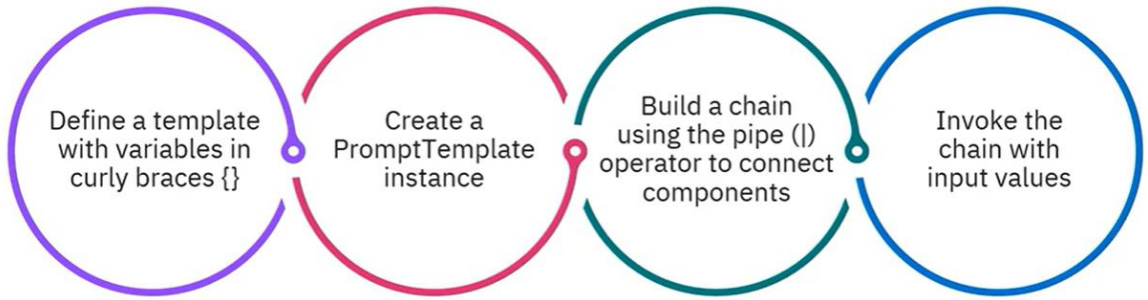
- Builds applications using the pipe (|) operator
- Ensures a clean, readable flow of data

What is LCEL?

The new, recommended LCEL pattern provides:

- Better composability
- Clearer visualization of data flow
- Greater flexibility when constructing complex chains

Create an LCEL pattern



LCEL main Runnable composition primitives

Runnables

- Interface and building blocks
- Connects the following components into a pipeline:
  - LLMs
  - Retrievers
  - Tools

LCEL main Runnable composition primitives

RunnableSequence

- Chains components sequentially
- Passes output from one component as input to the next

```
from langchain_core.runnables import
RunnableSequence


chain = RunnableSequence([runnable1,
runnable2])
```

LCEL main Runnable composition primitives

RunnableParallel

- Runs multiple components concurrently

```
from langchain_core.runnables import
RunnableParallel

chain = RunnableParallel({

   "key1": runnable1,

   "key2": runnable2,

})
```

LCEL main Runnable composition primitives

LCEL

- Avoids usage of RunnableSequence
- Connects runnable1 and runnable 2 with a pipe (|)

```
chain = Runnable1 | Runnable2
```

LCEL main Runnable composition primitives

LCEL:

Handles type coercion automatically

Converts regular code into runnable components

Converts dictionaries to RunnableParallel and functions to RunnableLambda

LCEL main Runnable composition primitives

Pipe (|) operator

- Combines the prompt templates with the LLM

Dictionary structure

- Creates a RunnableParallel that processes all three tasks

```
ize this text:
 {text}") | llm,

"translation":
ChatPromptTemplate.from_template("Transl
ate this text to French: {text}") | llm,

"sentiment":
ChatPromptTemplate.from_template(

"What is the sentiment of this text?
Answer with positive, negative, or
neutral: {text}") | llm

}
```

Creating templates and chains

RunnableLambda

- Wraps the format_prompt function
- Transforms it into a runnable component that LangChain can work with

```
# Build a chain with the pipe operator

joke_chain = (

RunnableLambda(format_prompt)

| llm

| StrOutputParser()

)
```

Creating templates and chains

- Pipe (|) operator creates a sequence by connecting runnable components
- RunnableLambda formats the prompt with variables
- Pipe (|) operator passes the formatted prompt to the LLM
- Another pipe passes the response to the StrOutputParser

```
joke_chain = (

RunnableLambda(format_prompt)

| llm

| StrOutputParser()

)


# Run the chain
```

What next?

We covered:

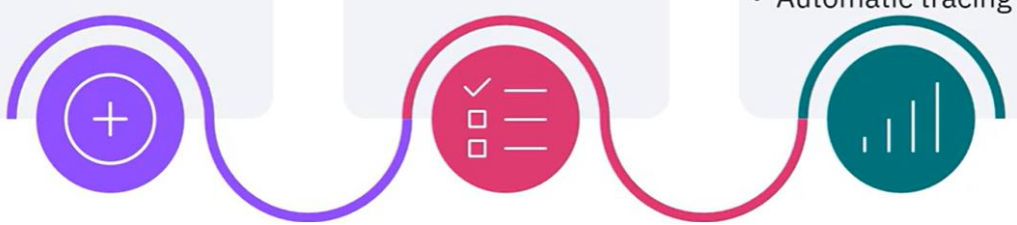
- Essentials of LCEL
- Benefits and composition primitives

LCEL: Suited for simpler orchestration tasks

LangGraph: Suited for complex workflows

LCEL's strengths:

- Parallel execution
- Async support
- Simplified streaming
- Automatic tracing



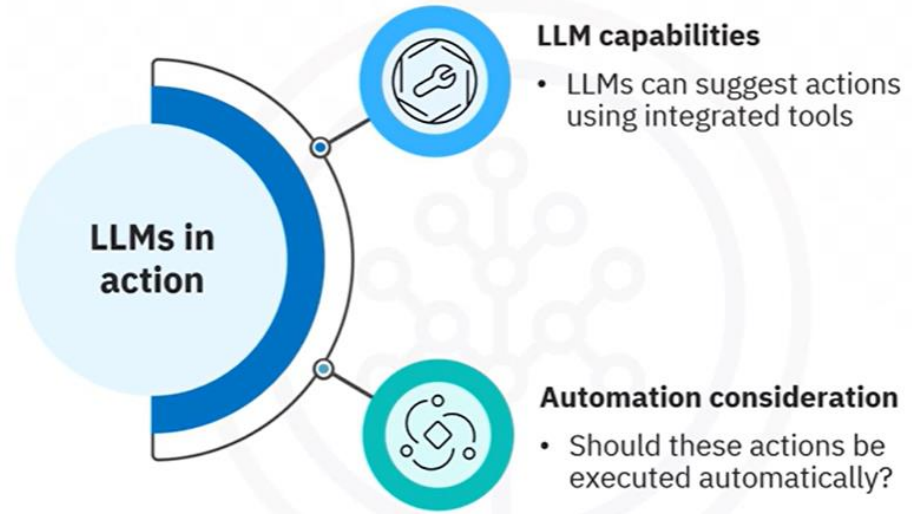
- LCEL pattern structures workflows use the pipe (|) operator
- Prompts use templates with variables in curly braces {}
- RunnableSequence links components for sequential execution
- RunnableParallel runs multiple components concurrently with the same input
- LCEL simplifies syntax by replacing RunnableSequence with the pipe operator
- Type coercion automatically converts functions and dictionaries into compatible components

Introduction

AI developer building intelligent systems

- LLM suggests automatically updating sensitive financial databases
- Comes with significant risks:
  - Inaccurate reporting
  - Financial losses
  - Regulatory issues

Introduction



Understanding tool suggestions by LLMs

LLMs

- Equipped with knowledge of various tools (actions or functions)
- Suggests a tool and specific details to complete the task

Example

- Weather API (tool)
- Location and date (parameters)

AI efficiency

- Understand why a specific tool is recommended
- Recognize how parameters shape outcomes
- Build accurate, efficient, and meaningful AI systems

Why choose manual invocation

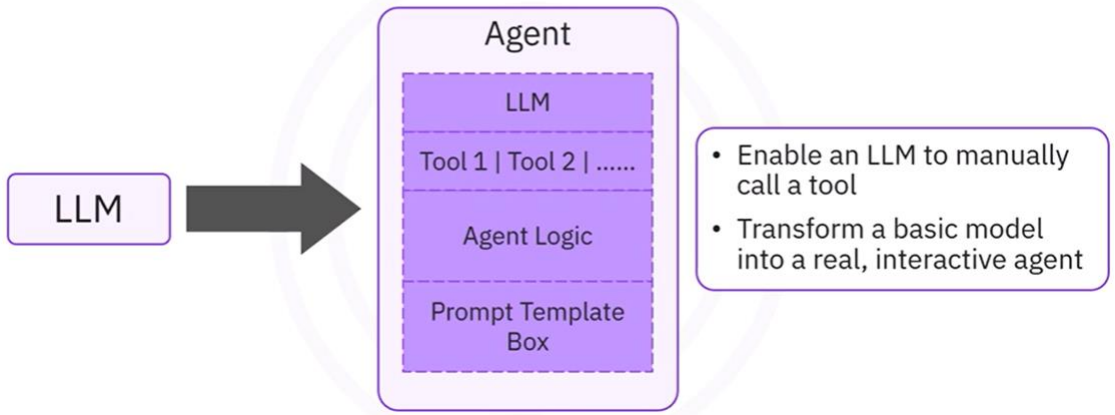
1. **Safety**
   Prevents unintended actions
2. **Cost control**
   Avoids unnecessary API calls that might incur charges
3. **Accuracy**
   Ensures that the tool is used correctly with the right parameters

Benefits of manual invocation

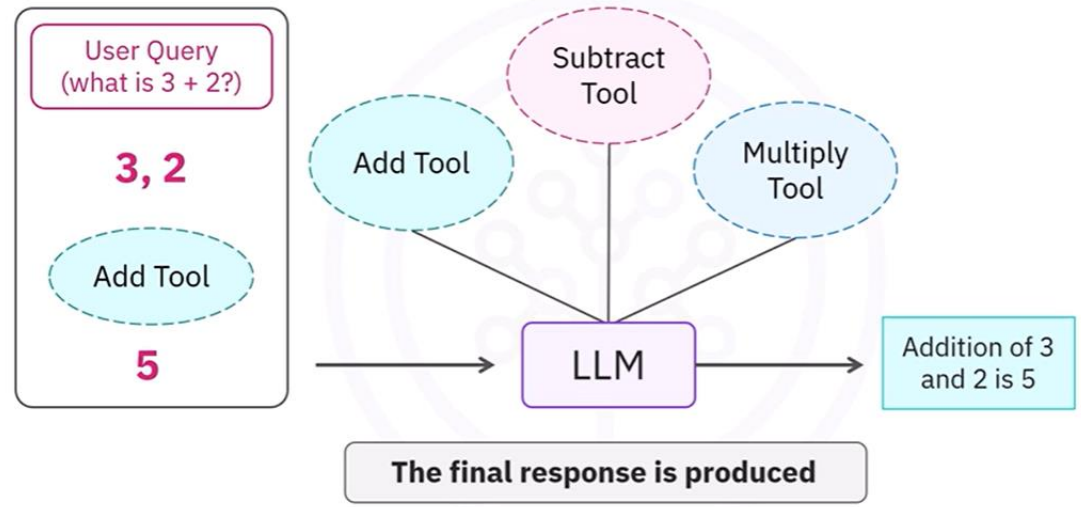
Recap

- Recognize the risks of automatic tool execution and the importance of manual control with distinct focuses and responsibilities
- Understand how LLMs suggest tools and parameters
- Explore the role of agents in executing LLM tools
- Evaluate the benefits of manual invocation, including enhanced safety, cost control, and accuracy, for smarter decision-making

Introduction



From LLM to agent



Initialize the chat model

Initialize the chat model

Handles the setup

Creates an object

Represents the LLM object

```python
from langchain.chat_models import init_chat_model

llm = init_chat_model("gpt-4o-mini", model_provider="openai")
```

LLM

Define your tool

- Define a custom tool
- Enable the model to perform basic arithmetic

Import the @tool decorator

Define a function

Docstring: “Add a and b.”

Defined tool not connected to the model

```python
@tool
def add(a: int, b: int) -> int:
    """Add a and b."""
    return a + b
```

LLM

Add more tools to the LLM

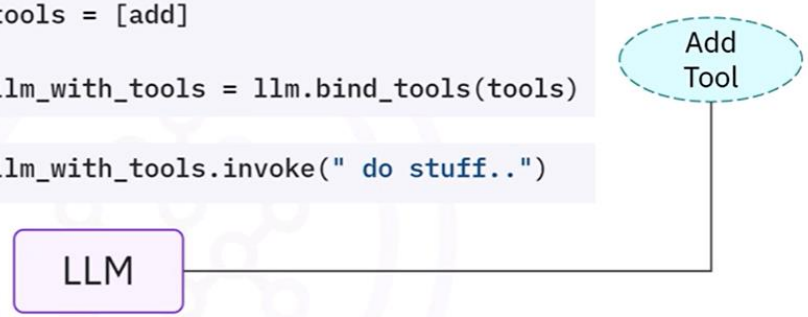
Connects the add function to the model

Places the add function in a list of tools

- Binds the list to the chat model
- Creates a new object called “llm with tools”

Uses the add tool to compute sums.

```
tools = [add]

llm_with_tools = llm.bind_tools(tools)

llm_with_tools.invoke(" do stuff..")
```



Add more tools to the LLM

Call a function dynamically

Create a mapping dictionary

Define the input arguments

Call the function by using the tool name

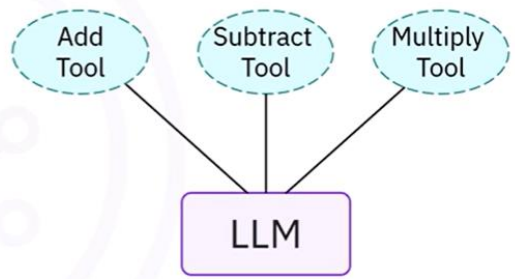
Use .invoke(input_) to match the keys to the function’s parameters

```
tool_map={"add":add,"subtract":subtract,"multiply":multiply}

input_={"a":1,"b":2}

tool_map["add"].invoke(input_)
```

Bind more tools to the LLM

List of tools: Add, Subtract, and Multiply

```
tools=[add, subtract, multiply]

llm_with_tools=llm.bind_tools(tools)
```



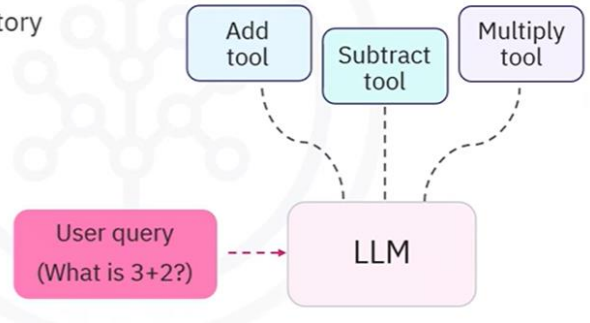
Recap

- Transform LLMs into interactive agents by integrating custom tools
- Set up chat models to handle real-world inputs
- Create and connect tools such as addition, subtraction, and multiplication functions
- Use mapping dictionaries for flexible, name-based function execution
- Understand how LLMs identify the right tool and manage parameter inputs
- Track and manage chat history, preserving conversation context

Craft the user query

```
from langchain_core.messages import HumanMessage
chat_history = [ HumanMessage(content = query) ]
```

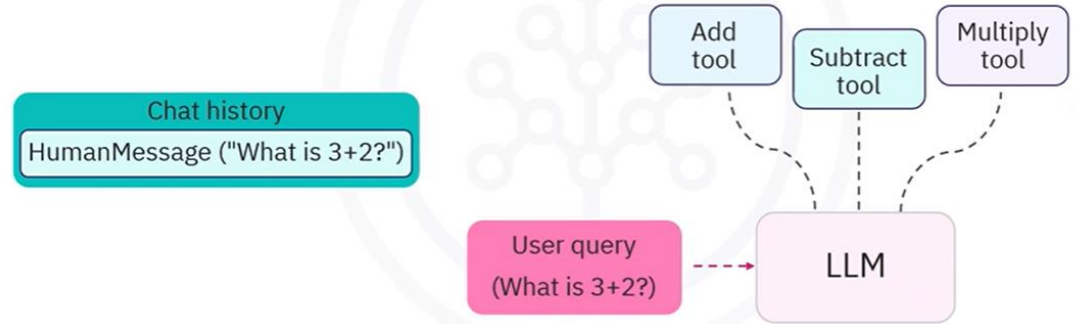
- Set up the question
- Insert the question into chat history



Craft the user query

```
from langchain_core.messages import HumanMessage
chat_history = [ HumanMessage(content = query) ]
```

- Convert the question into a HumanMessage

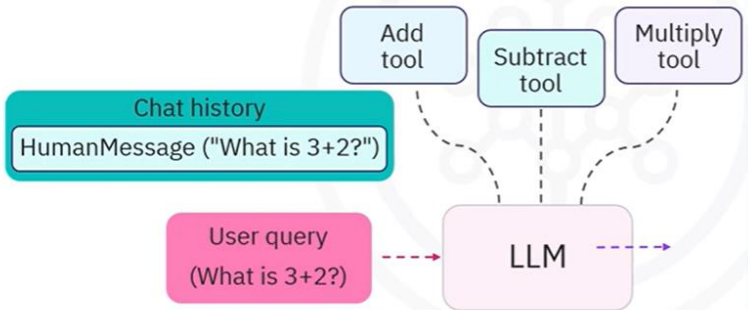


Invoke the model

```
response_1 = llm_with_tools.invoke(chat_history)
```

LLM thinks that I need to use the ADD tool to perform addition

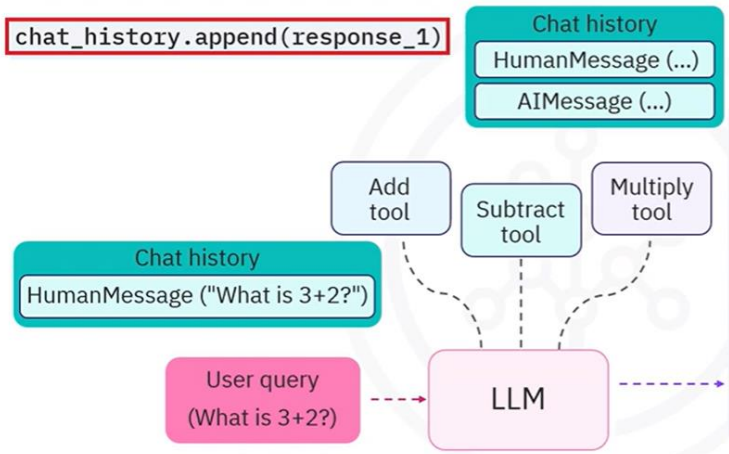
- AIMessage is received




```
AIMessage(
    tool_calls = [{
        "name": "add",
        "args": {
            a: 3,
            b: 2
        },
        "id": "call_add123",
        "type": "tool_call"
    }]
)
```

Invoke the model

```
chat_history.append(response_1)
```



LLM thinks that I need to use the ADD tool to perform addition

```
AIMessage(
    tool_calls = [{
        "name": "add",
        "args": {
            a: 3,
            b: 2
        },
        "id": "call_add123",
        "type": "tool_call"
    }]
)
```

Understand the LLM's response

- Response1 is an AIMessage object
- "add" is the name of the tool
- JSON string specifies input
- Links the response back to the request
- Specifies that this is a tool call

```
AIMessage(
    tool_calls = [{
        "name": "add",
        "args": {
            a: 3,
            b: 2
        },
        "id": "call_add123",
        "type": "tool_call"
    }]
)
```

Parse tool calls

- Execute the tool manually
- Parse the "add" tool
- Feed the arguments into the function

```
AIMessage(
    tool_calls = [{
        "name": "add",
        "args": {
            a: 3,
            b: 2
        },
        "id": "call_add123",
        "type": "tool_call"
    }]
)
```

Extract tool name and args

- **Extract tool_call_1_id**

```
AIMessage(
    tool_calls = [{
        "name": "add",
        "args": {
            a: 3,
            b: 2
        },
        "id": "call_add123",
        "type": "tool_call"
    }]
)
```

Accessing Tool Calls →

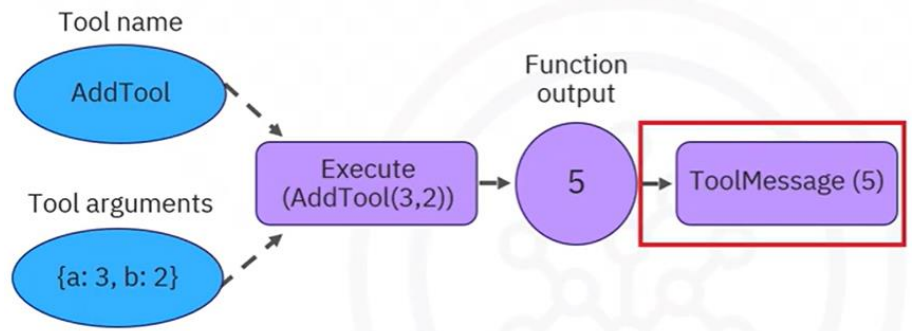
```
[{
        "name": "add",
        "args": {
            a: 3,
            b: 2
        },
        "id": "call_add123",
        "type": "tool_call"
    }]
```

Tool name → AddTool

Tool arguments → {a: 3, b: 2}

```
tool_1_name = tool_calls_1[0]['name']
tool_1_args = tool_calls_1[0]['args']
tool_call_1_id = tool_calls_1[0]['id']
```

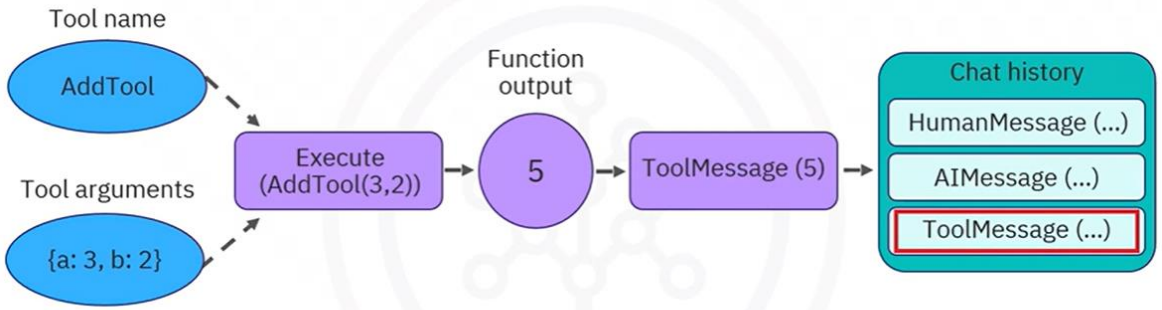
Invoke the tool




```
tool_response = tool_map[tool_1_name].invoke(tool_1_args)
from langchain_core.messages import ToolMessage
tool_message = ToolMessage(content = tool_response,
tool_call_id = tool_call_1_id)
```

Update chat history with the tool message

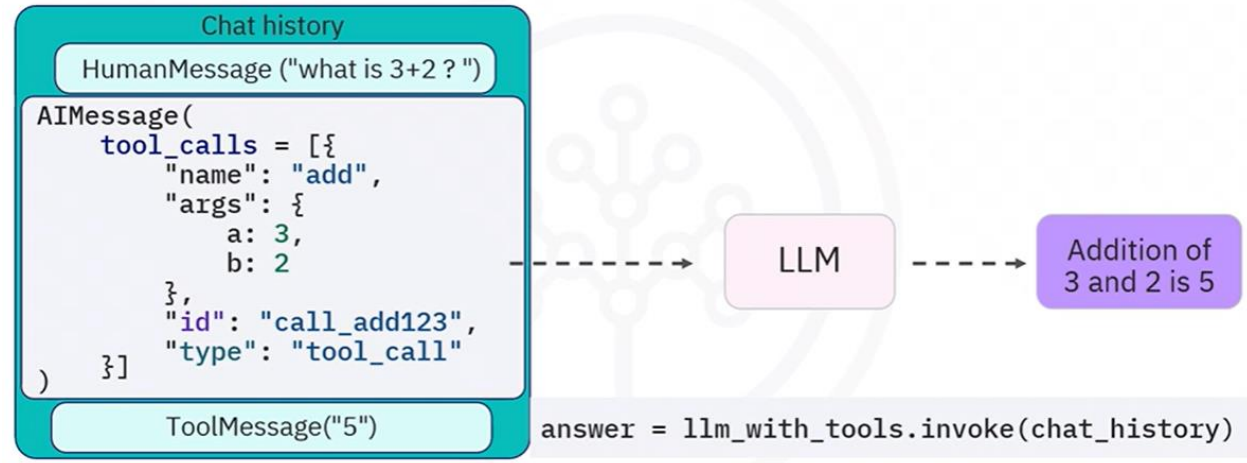
- **ToolMessage**




```
chat_history.append(tool_message)
```

Final model invocation

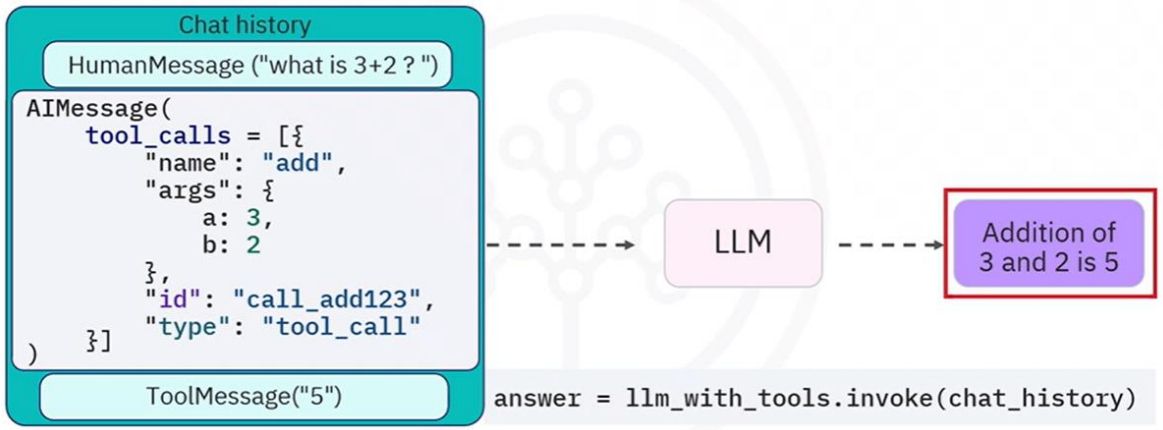
- **Pass the updated chat history to LLM**




```
answer = llm_with_tools.invoke(chat_history)
```

Final model invocation

- **LLM generates a final response**




```
answer = llm_with_tools.invoke(chat_history)
```

Recap

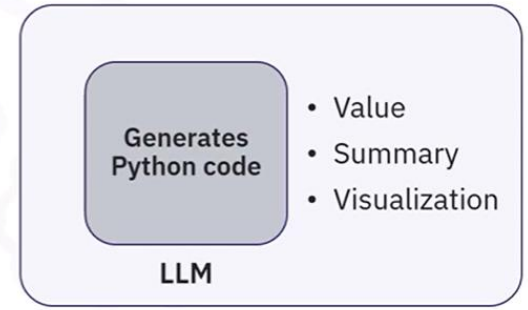
- Structure user interactions for real-time, context-aware conversations
- Extract tool names and arguments to precisely match user intent
- Parse complex tool instructions, including handling multiple tool calls
- Build and refine agent classes to automate the entire tool-calling process
- Understand how these components work together to transform LLMs from passive responders into intelligent agents

Agent use and availability

Using the create_pandas_dataframe_agent

The `create_pandas_dataframe_agent`:

- Uses preconfigured functions and prompts
- Operates on your existing Pandas DataFrame
- Accepts prompt input from users
- Responds to prompts with answers or visuals



Pandas Dataframe Agent

Set up the Pandas DataFrame agent

1. Import create_pandas_dataframe_agent
2. Create a DataFrame agent to connect the LLM to a DataFrame
3. Pass the LLM the data set
4. Set return_intermediate_steps=True
5. Set return_intermediate_steps=True, and view the generated code

```
from langchain_experimental.agents.agent_toolkits import
create_pandas_dataframe_agent

agent = create_pandas_dataframe_agent(
    llm,
    df,
    verbose=True,
    return_intermediate_steps=True  # so that model could return code that
it generates to produce the charts/results
)
```

Use natural language for data visualization and analysis

- Ask a question using the invoke method

```
response = agent.invoke("how many rows of data are in this file?")
response['output']
```

- The Pandas agent returns the answer instantly: 395 rows

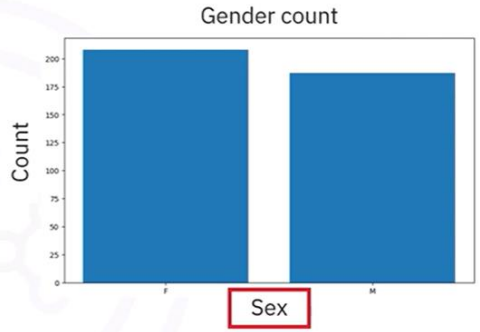
```
'There are 395 rows of data in this file.'
```

- View the code that provided the answer within the intermediate_steps parameter

```
response['intermediate_steps'][-1][0].tool_input.replace('; ', '\n')
:'len(df)'
```

Using natural language for data analysis

- ✓ Verbalize or type your request
- ✓ The LangChain Pandas Agent analyzes your request
- ✓ The LangChain Pandas agent generates charts



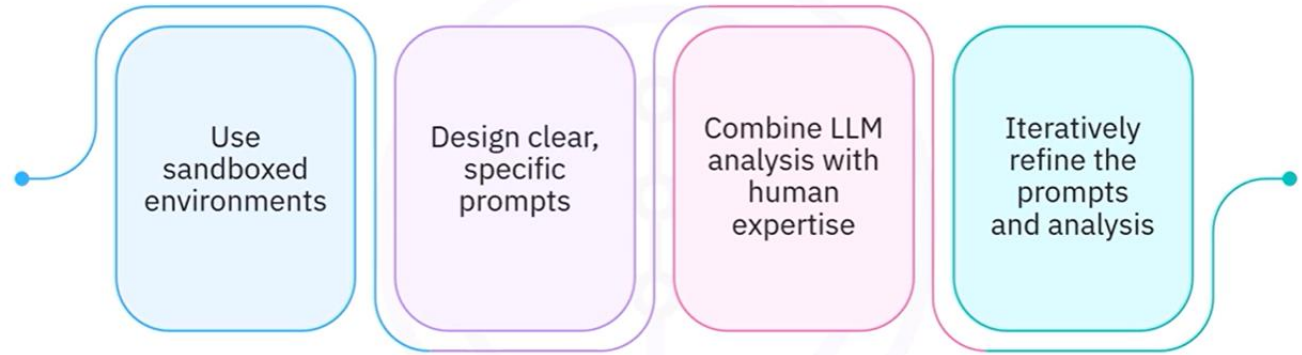
Say "Plot the gender count with bars."

```
response = agent.invoke("Plot the gender count with bars.")
```

Recap

- Using the LangChain Pandas agent is ideal for exploration and rapid prototyping but is ***not recommended*** for production environments unless comprehensive safeguards are in place.
- The LangChain Pandas agent is different because it uses preconfigured functions and prompts, operates on your existing Pandas DataFrame, accepts prompt inputs from users, and responds to prompts with answers or visuals.
- You can set up the LangChain Pandas agent to work with an external LLM such as IBM Watsonx.ai model.

Best Practices for Effective Use



Recap

- To set up the LangChain Pandas agent with an IBM Watsonx.ai model, first, initialize your model with WatsonxLLM, then connect it to a Pandas DataFrame using the `create_pandas_dataframe_agent` function.
- You can analyze and visualize data by asking natural language questions to the Pandas DataFrame agent, which instantly generates code and returns clear data insights.
- The agent-generated Python code directly interacts with your DataFrame, filtering, aggregating, and visualizing data based on your natural language prompts.
- Always use sandboxed environments, design clear prompts, validate LLM analysis with human expertise, and iteratively refine your queries for safe and effective AI-driven data analysis.

An AI-powered SQL agent's purpose

- Bridge the gap between natural language and SQL
- Enhance data accessibility by using AI

Provides more users with the ability to access and interpret data

AI-powered SQL agent capabilities

Database operations

- Reads and understands database schemas
- Retrieves schemas only from relevant tables

Query management

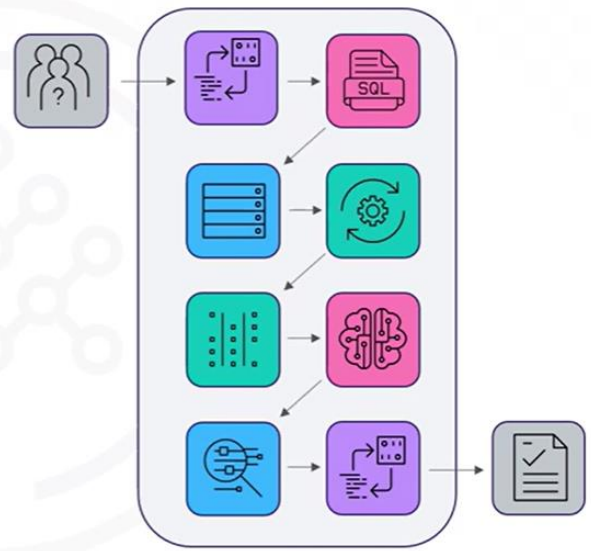
- Supports multi-step querying
- If a query fails, the agent
  - Captures errors
  - Analyzes tracebacks
  - Automatically retries the task using a corrected query

AI-powered SQL agent limitations & considerations

- AI interpretations might not always be accurate
- Complex queries might require manual adjustments
- Continuous testing and validation are essential

How AI-powered SQL agents retrieve information

1. The user asks a question.
2. The agent receives the question.
3. The LLM interprets the question and generates an SQL query.
4. The database connector sends the SQL query to a database.
5. The database processes the query.
6. The database returns raw data.
7. The database connector passes data to LLM.
8. The LLM processes raw data.
9. The LLM transforms data into a readable response.
10. The agent displays the readable response.



Load the LLM using IBM watsonx.ai

2 Initialize the watsonx.ai model by providing:

- **model_id:** Identifier for the specific model (Example: **ibm/granite-3-2-8b-instruct**)
- **credentials**: Dictionary containing authentication details
- **params**: Generation parameters like **max_new_tokens** and temperature

```
model = Model(
    model_id="ibm/granite-3-2-8b-instruct",
    credentials={"apikey": "your_api_key", "url": "https://us-
south.ml.cloud.ibm.com"},
    params={"max_new_tokens": 256, "temperature": 0.5},
    project_id="your_project_id"
```

Connect IBM watsonxLLM to LangChain

3 Wrap the model with watsonxLLM for LangChain integration

```
granite_llm = WatsonxLLM(model=model)
```

You can now:

- Build interactive chatbots
- Connect it to SQL agents for database queries
- Integrate the watsonx.ai model with vector stores for semantic search

Create the SQL agent

1. Import the agent creation function
2. Initialize the agent with the LLM and database connection
3. (Optional) Enable verbose logging
4. Set the agent type to zero-shot reasoning

```
# Create the SQL agent
agent_executor = create_sql_agent(
    llm=granite_llm,            # The LLM model we defined earlier
    db=db,                      # The database connection instance
    verbose=True,               # Set to True for detailed logs
    agent_type=AgentType.ZERO_SHOT_REACT_DESCRIPTION)  # Type of agent
```

Recap

- Begin the implementation process by creating a Python virtual environment. This is done by running the **virtualenv** command.
- Install the necessary libraries, which include **ibm-watsonx-ai**, **LangChain**, and **mysql-connector-python**. These libraries enable the AI model's integration, LangChain functionalities, and database connectivity.
- After the virtual environment is set up, you need to launch the MySQL server and access the MySQL CLI (Command Line Interface).
- To allow LangChain to interact with the database, you must build a database connector.
- LangChain requires an agent to interact with the database using natural language. Import the **create_sql_agent** function from LangChain's community toolkit.
- Initialize the agent with the LLM instance, the database connection, and the agent type. Enable the verbose parameter to view detailed logs of the query translation and execution.
- To run a query, you use a natural language query along with the SQL agent.