Agents in LangChain

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Dilini K. Sumanapala, PhD
Founder & AI Engineer, Genverv Ltd.

datacamp

Meet your instructor

- Dilini K. Sumanapala, PhD
- AI Engineer
- Cognitive Neuroscience
- Natural Language Applications
- Founder, Genverv Ltd.

An overview of agents and tools

- **Agents**

  Autonomous systems that make decisions and take actions

- **Tools**

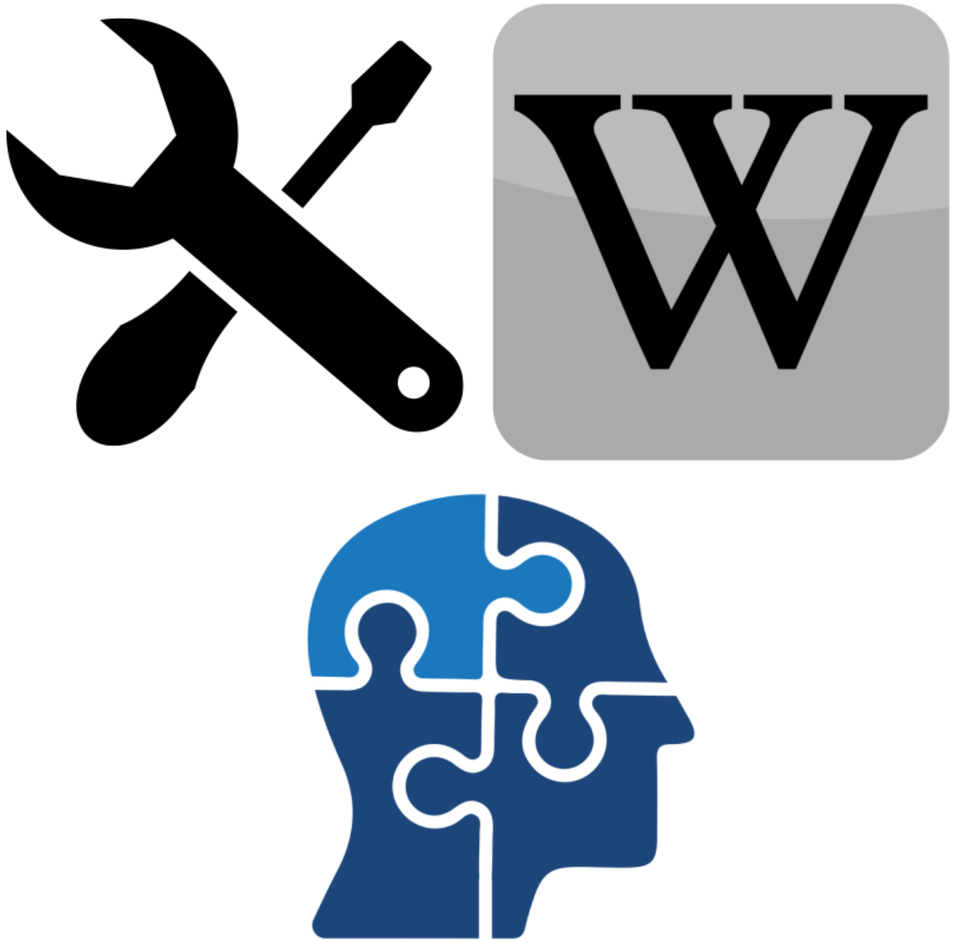
  Functions agents use to perform specific tasks

  - Data query
  - Research reports
  - Data analysis

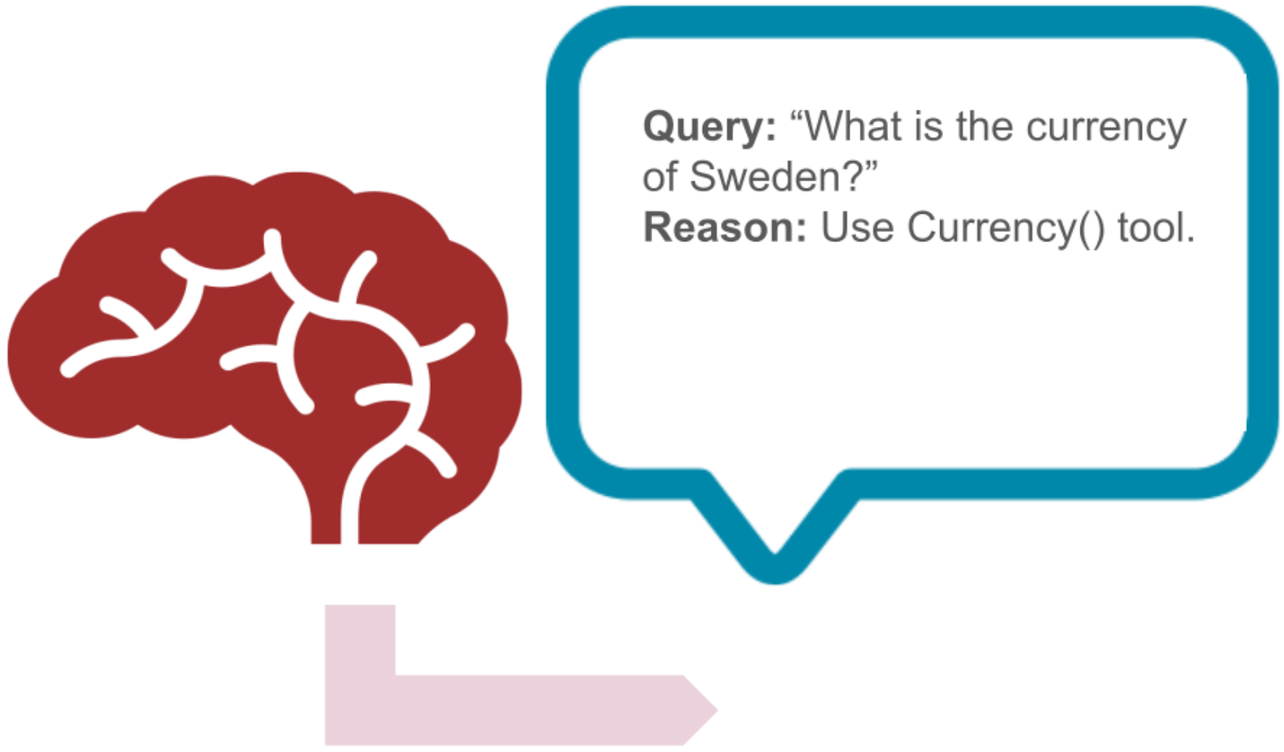
Basic concepts

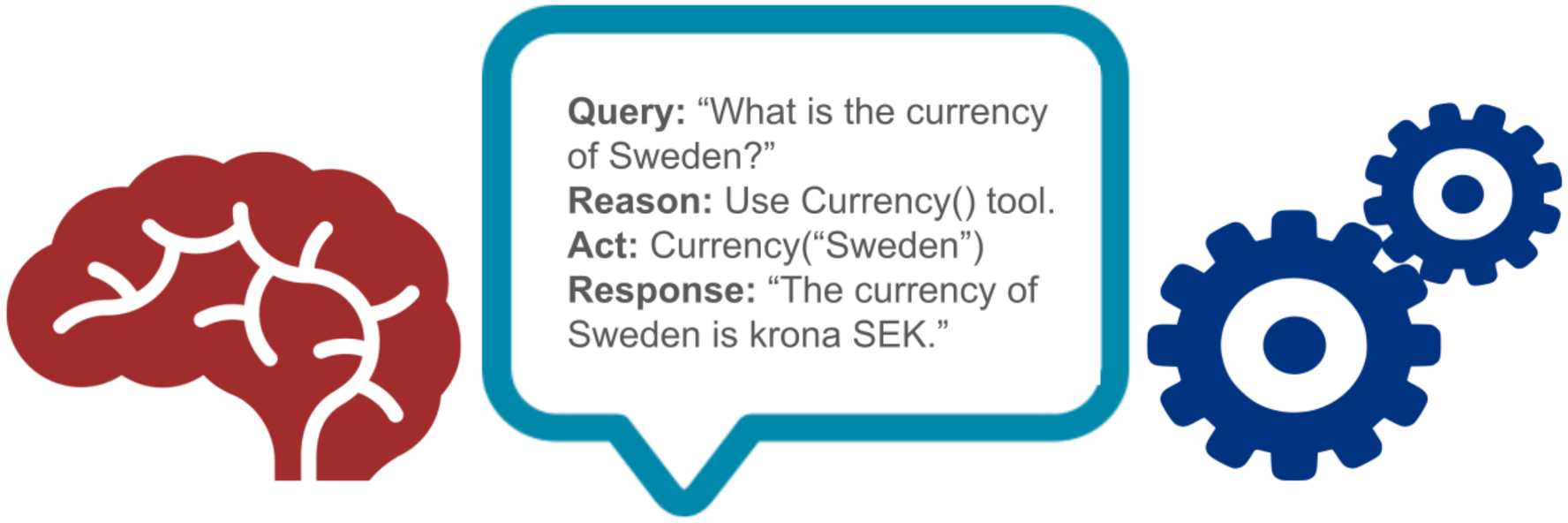
- LLMs (e.g., ChatGPT)
- Prompts
- Tools
- API
- LangChain
  - Building AI agents

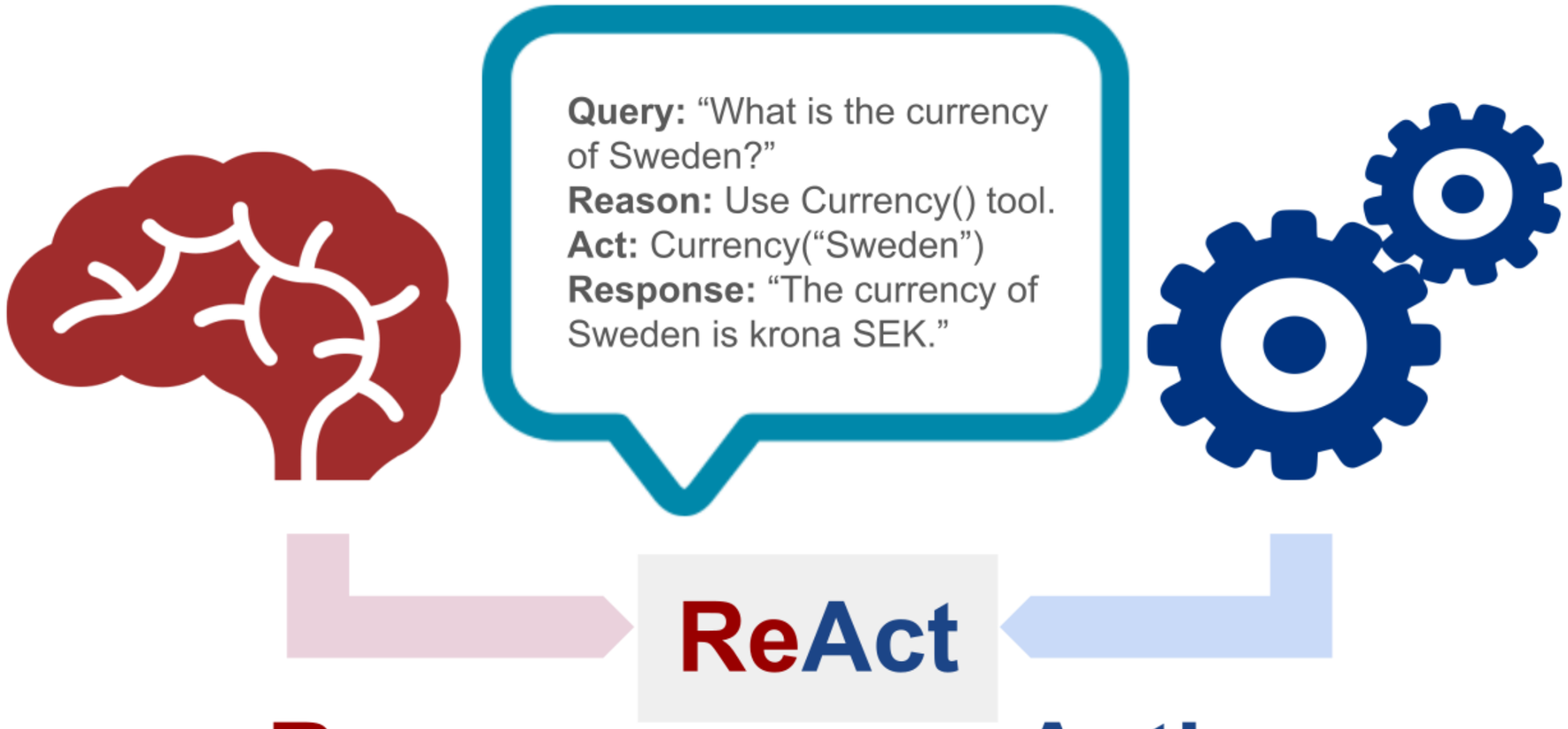
Course overview



- Math problems
- Wikipedia search
- Switch between tools and LLMs


Query: “What is the currency of Sweden?”
Reason: Use Currency() tool.
Reason


Query: “What is the currency of Sweden?”
Reason: Use Currency() tool.
Act: Currency(“Sweden”)
Response: “The currency of Sweden is krona SEK.”
Reason
Action


Query: “What is the currency of Sweden?”
Reason: Use Currency() tool.
Act: Currency(“Sweden”)
Response: “The currency of Sweden is krona SEK.”
ReAct
Reason
Action

Improving response accuracy

241 - (-241) + 1

241 - (-241) + 1 is equivalent to 241 + 241 + 1, which simplifies to 483 +
241 - (-241) + 1 is equal to 484.

Correct Answer: 483

- Coding
- Math

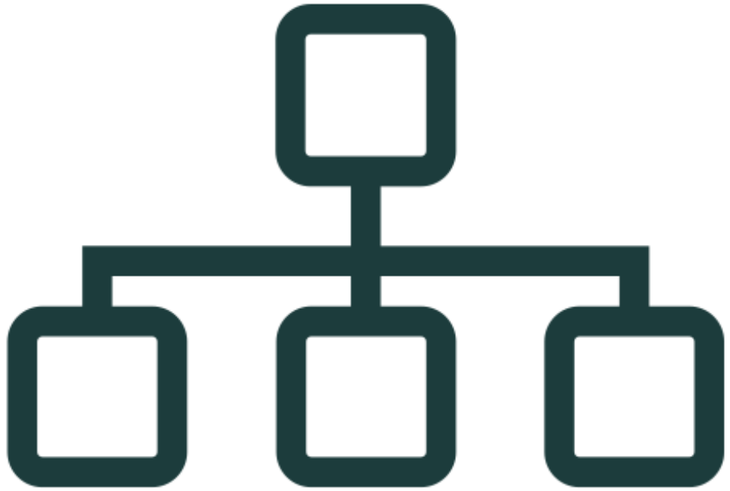
[1] https://community.openai.com/t/chatgpt-simple-math-calculation-mistake/62780

Breaking up problems

Order of Math Operations

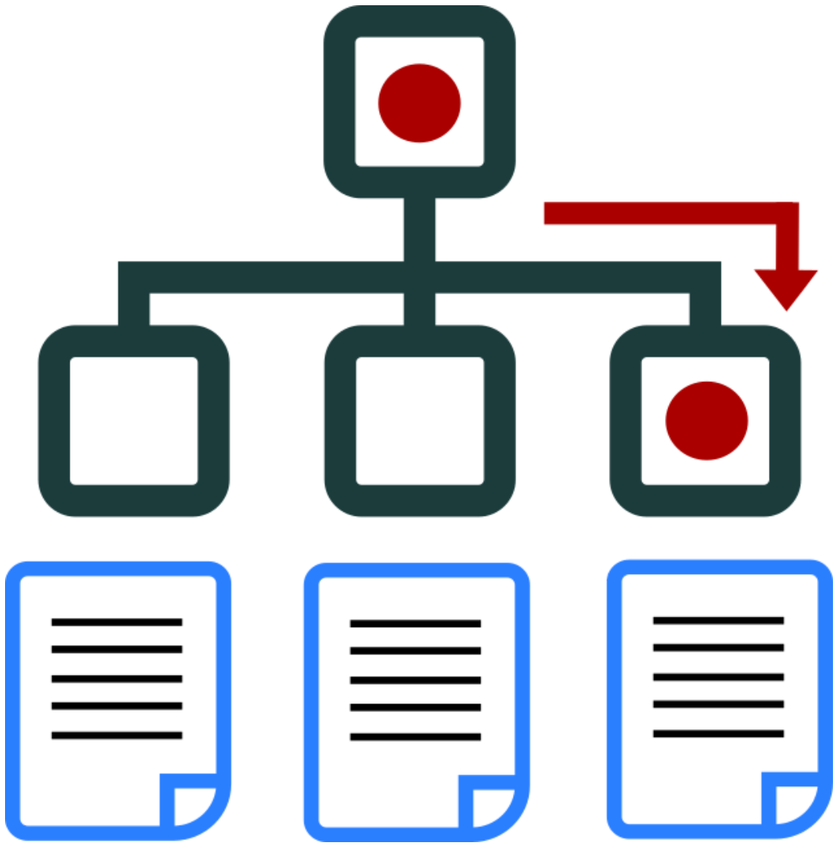
1. Parentheses
2. Exponents
3. Multiplication/Division
4. Addition/Subtraction

Expanding agents with LangGraph



LangGraph

Graph structures



Nodes

- Query the Database
- Return the Document

Edges

Rules connecting nodes

Create a ReAct agent

```
# Module imports
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI
from langgraph.prebuilt import create_react_agent
import math

# LLM Setup
model = ChatOpenAI(openai_api_key="<OPENAI_API_TOKEN">, model="gpt-4o-mini")
```

Create a ReAct agent

```
# Create the agent
agent = create_react_agent(model, tools)

# Create a query
query = "What is (2+8) multiplied by 9?"

# Invoke the agent and print the response
response = agent.invoke({"messages": [("human", query)]})

# Print the agent's response
print(response['messages'][-1].content)
```

```
<script.py> output:
    The result of (2 + 8) multiplied by 9 is 90.
```

Let's practice!

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

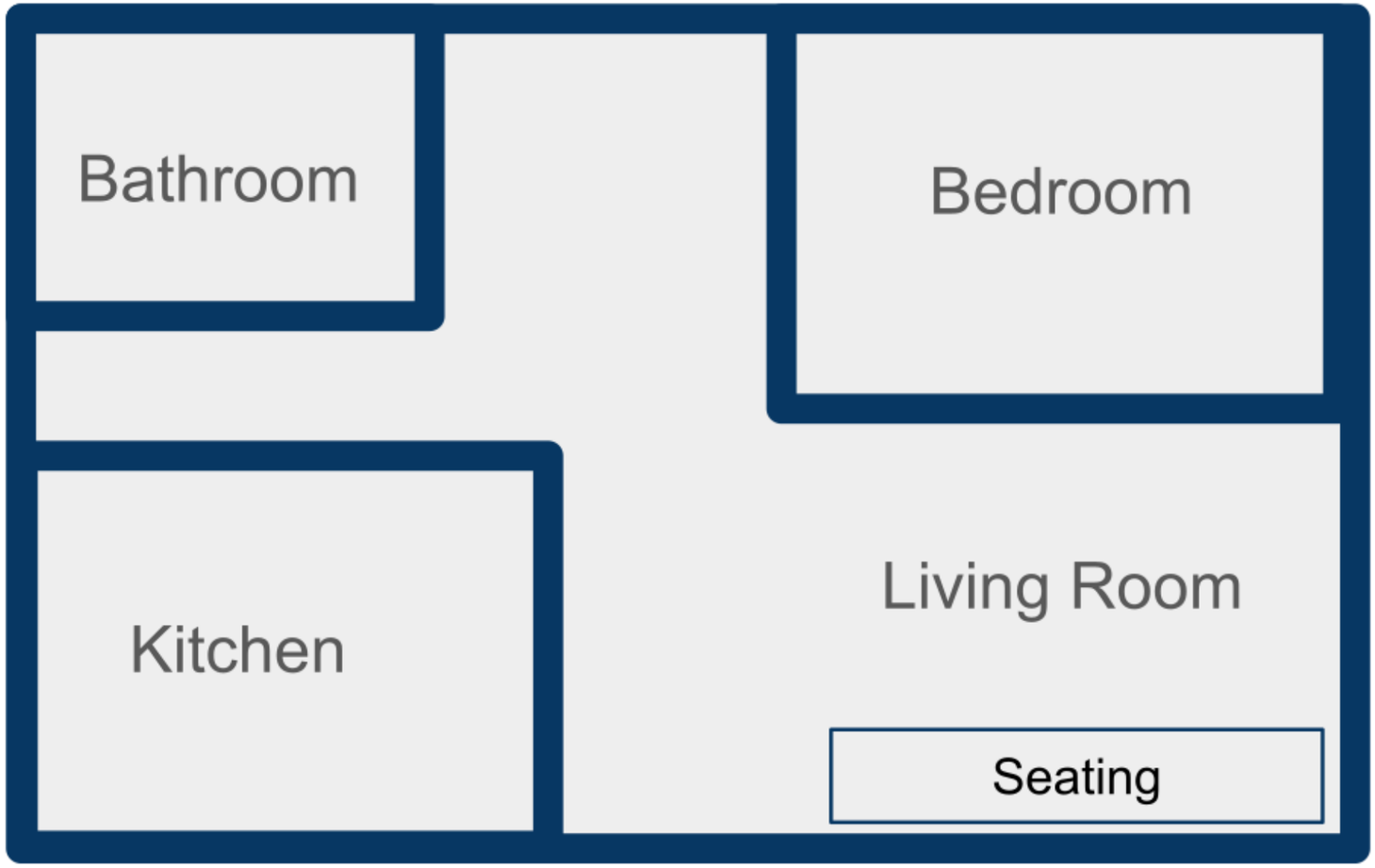
Building custom tools

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

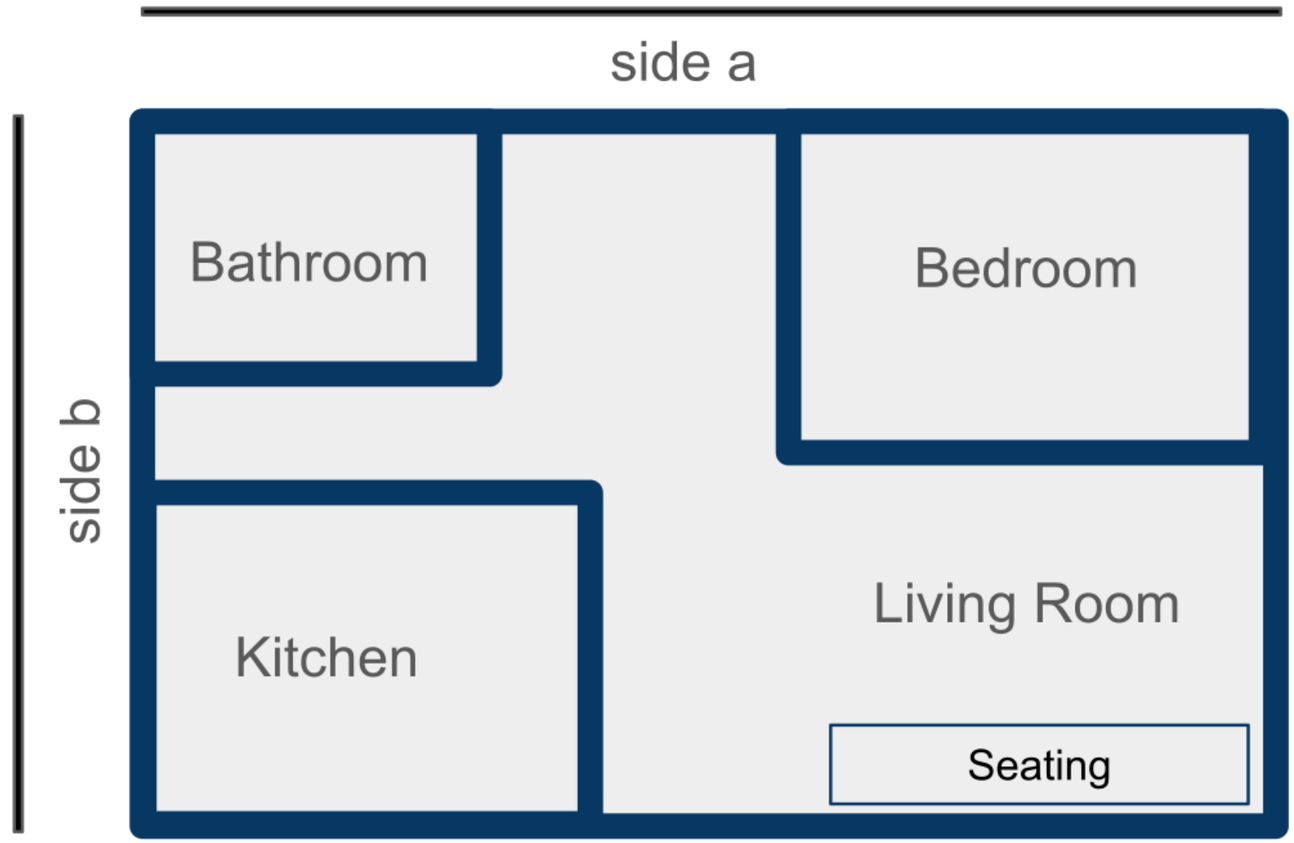
Dilini K. Sumanapala, PhD
Founder & AI Engineer, Genverv, Ltd.

datacamp

Calculating square footage



Calculating square footage



Creating a math tool

LangChain's internal query handling

```
"What is the area of a rectangle with
sides 5 and 7?"
input = " 5, 7"
```

- **Natural language input**
- **Extract numeric values as strings**

Creating a math tool

Define your tool function

```
@tool
def rectangle_area(input: str) -> float:
    """Calculates the area of a
    rectangle given the lengths of
    sides a and b."""
    sides = input.split(',')
    a = float(sides[0].strip())
    b = float(sides[1].strip())
    return a * b
```

- Use `@tool` decorator
- Name the function
- Create a docstring
- Split the input using `.split()`
- Strip whitespace using `.strip()` and convert to float
- Multiply `a` and `b` and return the answer

Tools and query setup

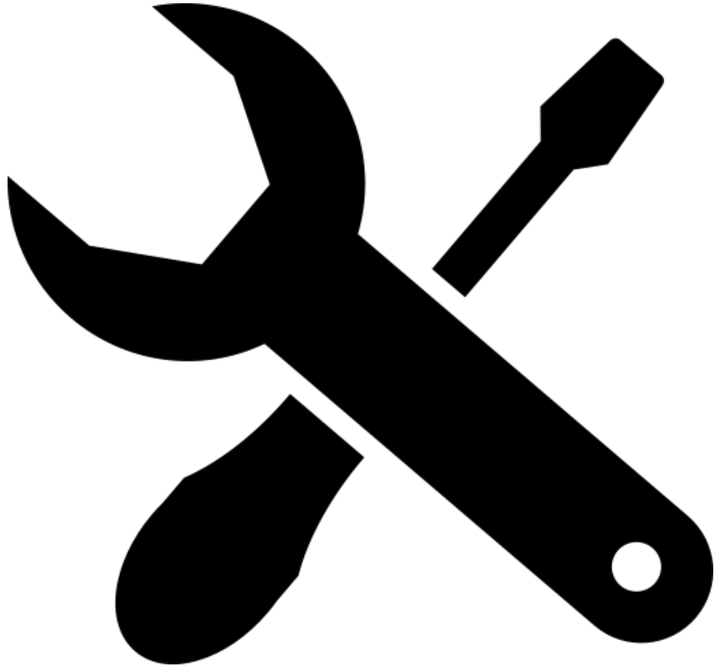
```
# Define the tools that the agent can access
tools = [rectangle_area]

# Create a query using natural language
query = "What is the area of a rectangle with sides 5 and 7?"

# Pass in the hypotenuse length tool and invoke the agent
app = create_react_agent(model, tools)
```

Tools and query setup

```
# Invoke the agent and print the response
response = app.invoke({"messages": [("human", query)]})
print(response['messages'][-1].content)
```

```
The area of the rectangle with sides 5 and 7 is 35 square units.
```

Pre-built and custom tools



- Database Querying
- Web Scraping
- Image Generation

- Pre-built tools API guide
- Custom tool guide

Let's practice!

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Conversation with a ReAct agent

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Dilini K. Sumanapala, PhD
Founder & AI Engineer, Genverv, Ltd.

datacamp

Conversation

```
The area of a rectangle with sides 5 and 7 is 35 square units.
```

- **Validating answers**
- **User:** "What is the area of a rectangle with sides 5 and 7?"
- **Agent:** "The area of a rectangle with sides 5 and 7 is 35 square units."

Conversation

```
tools = [rectangle_area]
query = "What is the area of a rectangle with sides 14 and 4?"

# Create the ReAct agent
app = create_react_agent(model, tools)

# Invoke the agent with a query and store the messages
response = app.invoke({"messages": [("human", query)]})

# Define and print the input and output messages
print({
    "user_input": query,
    "agent_output": response["messages"][-1].content})
```

Conversation output

```
{'user_input': 'What is the area of a rectangle with sides 14 and 4?',
 'agent_output': 'The area of a rectangle with sides 14 and 4 is 56
 square units.'}
```

Follow-up questions

- **Follow-up:**
  - **User**: "What about one with sides 12 and 14?"
- **Conversation history:**
  - **User**: "What is the area of a rectangle with sides 5 and 7?"
  - **Agent**: "The area of a rectangle with sides 5 and 7 is 35 square units."
  - **User**: "What about one with sides 12 and 14?"
  - **Agent**: "The area of a rectangle with sides 12 and 14 is 168 square units."
- **Output**
  - **User**: "What about one with sides 12 and 14?"
  - **Agent**: "The area of a rectangle with sides 12 and 14 is 168 square units."

Follow-up questions

```
{'user_input': 'What about one with sides 12 and 14?',
'agent_output': ['HumanMessage: What is the area of a rectangle with sides
5 and 7?', 'AIMessage: The area of a rectangle with sides 5 and 7 is 35
square units.',
'HumanMessage: What about one with sides 12 and 14?',
'AIMessage: The area of a rectangle with sides 12 and 14 is 168 square
units.',
'HumanMessage: What about one with sides 12 and 14?',
'AIMessage: The area of a rectangle with sides 12 and 14 is 168 square
units.']}
```

Conversation history

```
from langchain_core.messages import
HumanMessage, AIMessage
```

Conversation history

```
from langchain_core.messages import
HumanMessage, AIMessage

message_history = messages["messages"]
```

Conversation history

```
from langchain_core.messages import
HumanMessage, AIMessage

message_history = messages["messages"]
new_query = "What about one with sides
4 and 3?"
```

Conversation history

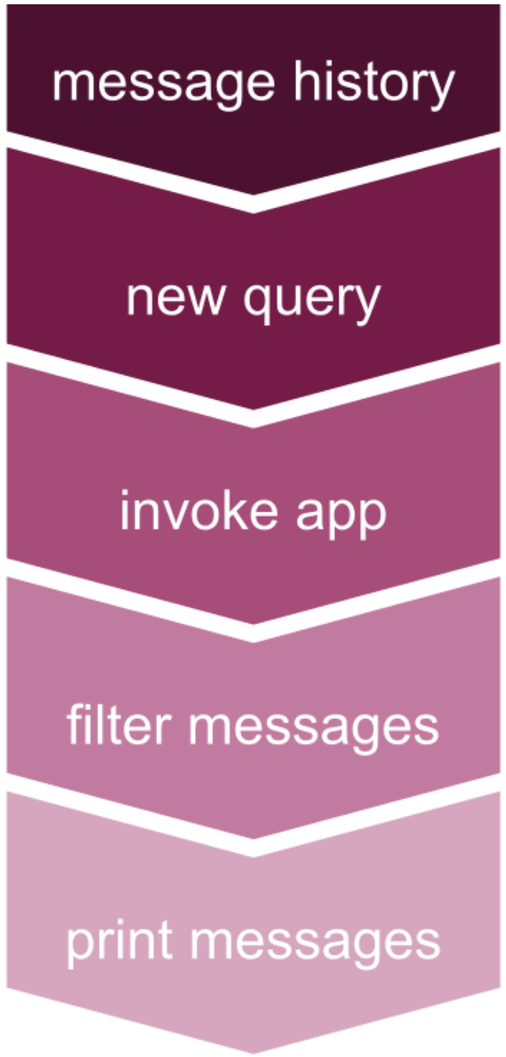
```
from langchain_core.messages import
HumanMessage, AIMessage

message_history = messages["messages"]
new_query = "What about one with sides
4 and 3?"

# Invoke the app with the full message history
messages = app.invoke({"messages":
           message_history + [("human",
           new_query)]})
```

Conversation history

```
# Extract the human and AI messages
filtered_messages = [msg for msg in
                     messages["messages"] if
                     isinstance(msg,
                     (HumanMessage,
                     AIMessage))
                     and msg.content.strip()]
```

Conversation history

```
# Extract the human and AI messages
filtered_messages = [msg for msg in
                     messages["messages"] if
                     isinstance(msg,
                     (HumanMessage,
                     AIMessage))
                     and msg.content.strip()]

# Format and print the final result
print({
    "user_input": new_query, "agent_output":
    [f"{msg.__class__.__name__}:
    {msg.content}" for msg in
    filtered_messages]})
```



Conversation history output

```
{'user_input': 'What about one with sides 4 and 3?',
'agent_output': ['HumanMessage: What is the area of a rectangle with sides
14 and 4?', 'AIMessage: The area of a rectangle with sides 14 and 4 is 56
square units.',
'HumanMessage: What about one with sides 4 and 3?',
'AIMessage: The area of a rectangle with sides 4 and 3 is 12 square
units.',
'HumanMessage: What about one with sides 4 and 3?',
'AIMessage: The area of a rectangle with sides 4 and 3 is 12 square
units.']}
```

Let's practice!

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Building graphs for chatbots

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Dilini K. Sumanapala, PhD
Founder & AI Engineer, Genverv, Ltd.

datacamp

Chatbots with LangGraph

- **Custom agents with LangGraph**
  - **Workflow management**
    - Graph states
    - Agent states
  - **Chatbot construction**
    - Nodes
    - Edges

Graphs and agent states

Graph State

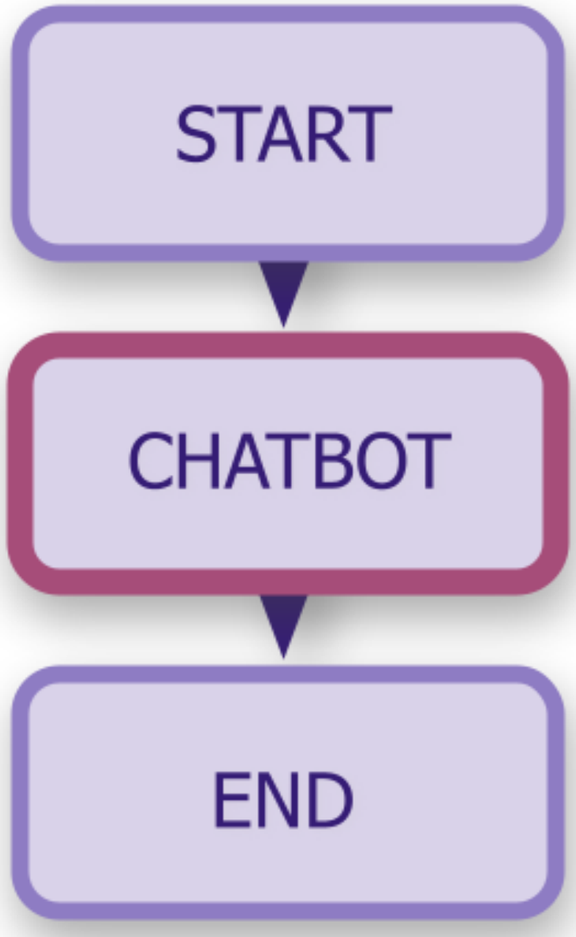
- Organizes different tasks
  - Tool use
  - LLM calls
- Order of tasks = **workflow**

Agent State

- Tracks agent's progress
- Logs task completion

Building an agent with LangGraph

Nodes and edges



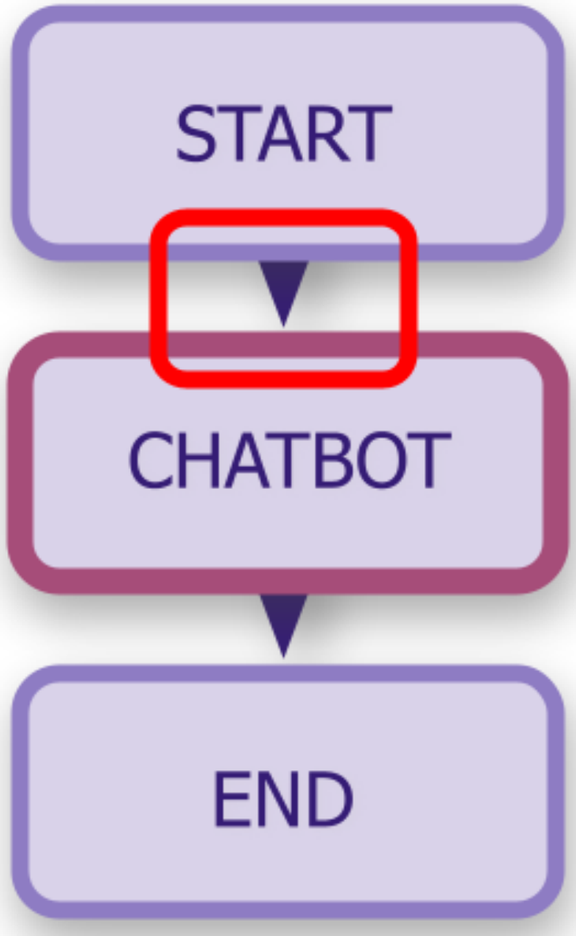
Nodes

- Functions or actions
  - Response
  - Tool call

Edges

- Rules connecting nodes

Nodes and edges



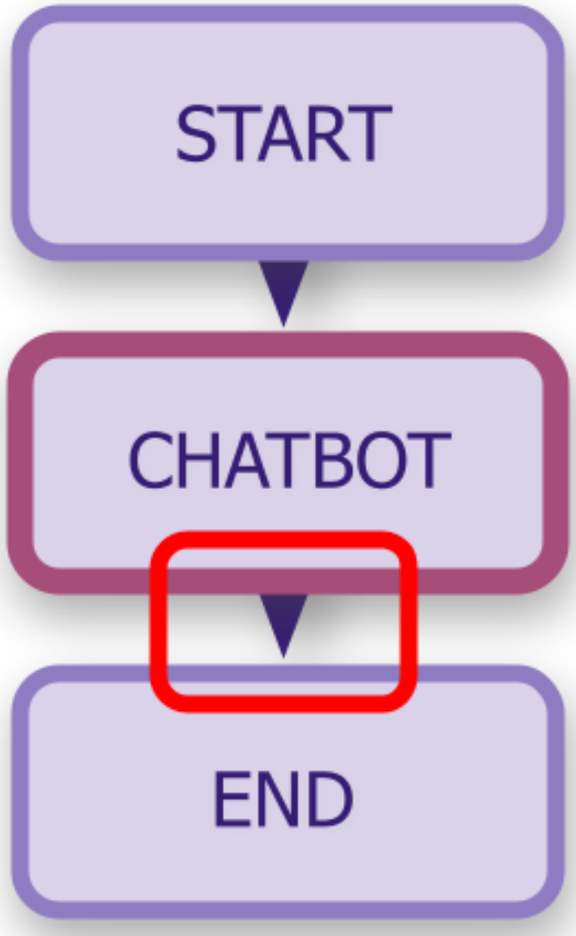
Nodes

- Tasks or actions
  - Response
  - Tool call

Edges

- Rules connecting nodes

Nodes and edges



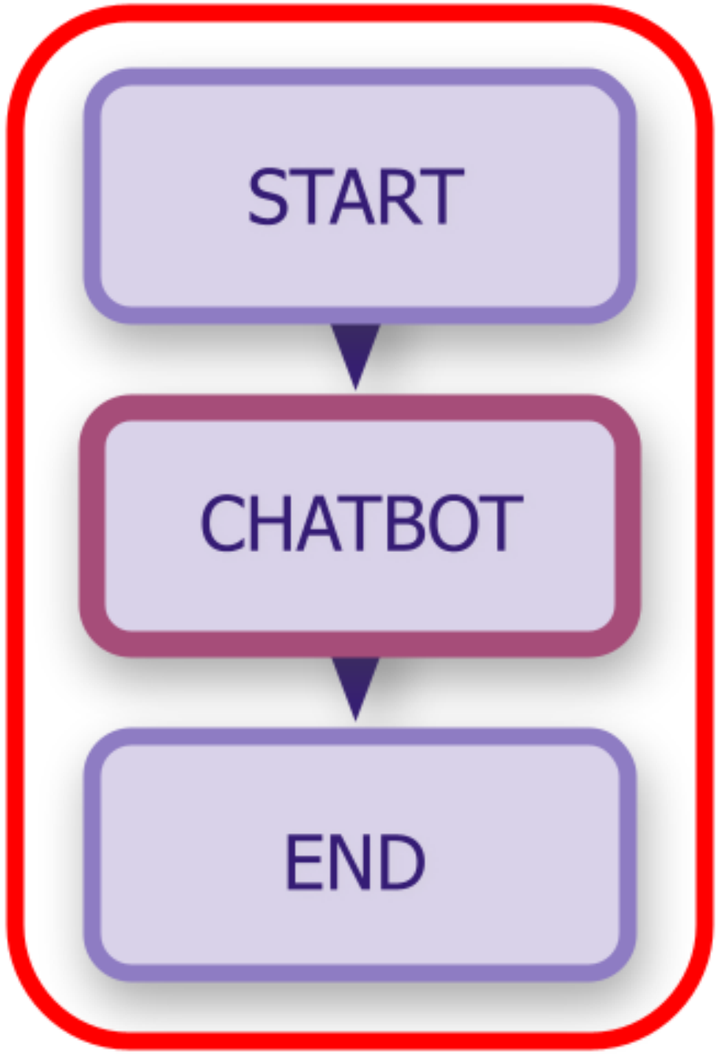
Nodes

- Tasks or actions
  - Response
  - Tool call

Edges

- Rules connecting nodes

Nodes and edges



Nodes

- Tasks or actions
  - Response
  - Tool call

Edges

- Rules connecting nodes

Pre-built Nodes

START and **END** nodes from LangGraph

Building graph and agent states

```
# Modules for structuring text
from typing import Annotated
from typing_extensions import TypedDict

# LangGraph modules for defining graphs
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages

# Module for setting up OpenAI
from langchain_openai import ChatOpenAI
```

Building graph and agent states

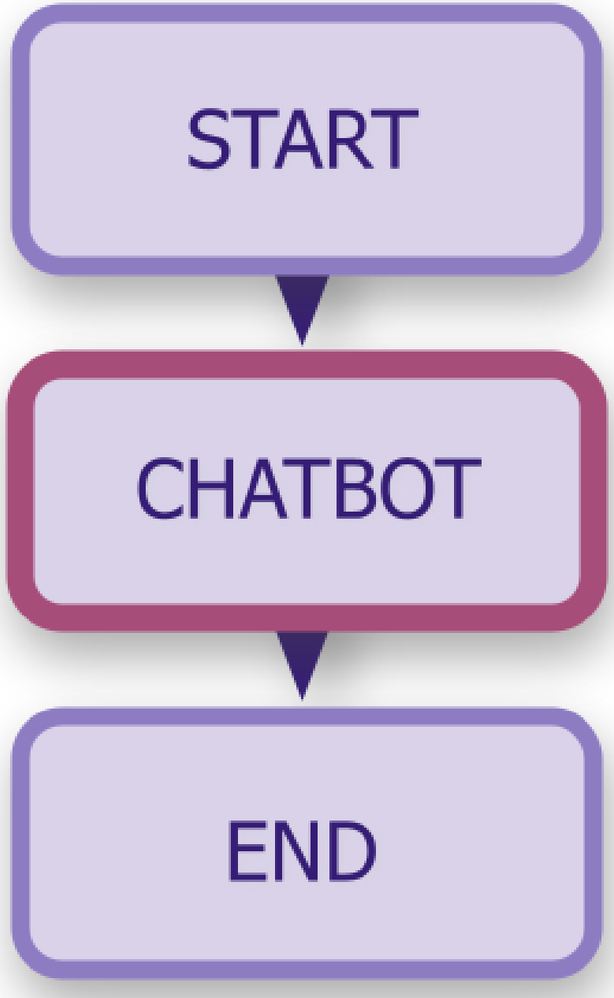
```
# Define the llm
llm = ChatOpenAI(model="gpt-4o-mini", api_key="OPENAI_API_KEY")

# Define the State
class State(TypedDict):

    # Define messages with metadata
    messages: Annotated[list, add_messages]

# Initialize StateGraph
graph_builder = StateGraph(State)
```

Adding nodes and edges

```
# Define chatbot function to respond
# with the model
def chatbot(state: State):
    return {"messages":
    [llm.invoke(state["messages"])]}

# Add chatbot node to the graph
graph_builder.add_node("chatbot",
                       chatbot)
```

Adding nodes and edges

```
# Define the start and end of the
# conversation flow
graph_builder.add_edge(START, "chatbot")
graph_builder.add_edge("chatbot", END)

# Compile the graph to prepare for
# execution
graph = graph_builder.compile()
```



Let's practice!

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Generating chatbot responses

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Dilini K. Sumanapala, PhD
Founder & AI Engineer, Genverv, Ltd.

datacamp

Streaming graph events

- Stream events in real-time
- Each event is a workflow step
- Track responses and tool calls
- Track chatbot progress

```
User: Where is the Congo?
Agent: Let me check some details
       for you...
Agent: AIMessage([Tool Call to
       Google Maps] Searching for
       information on "the Congo")
Agent: The Democratic Republic of
       Congo is a country in
       Central Africa.
```

Streaming LLM responses

```
# Define a function to execute the chatbot based on user input
def stream_graph_updates(user_input: str):

    # Start streaming events from the graph with the user's input
    for event in graph.stream({"messages": [("user", user_input)]}):

        # Retrieve and print the chatbot node responses
        for value in event.values():
            print("Agent:", value["messages"])

# Define the user query and run the chatbot
user_query = "Who is Mary Shelley?"
stream_graph_updates(user_query)
```

Streaming LLM responses

```
Agent: [AIMessage(content='Mary Shelley (1797-1851) was an English
novelist...
...best known for her groundbreaking work in the Gothic genre,
particularly for her novel "Frankenstein; or, The Modern Prometheus,"
published in 1818. This novel is often considered one of the earliest
examples of science fiction and explores themes of creation,
responsibility, and the nature of humanity through the story of Victor
Frankenstein, a scientist who creates a sentient creature in an
unorthodox experiment...',
response_metadata={'finish_reason': 'stop', 'model_name':
'gpt-4o-mini-...')]
```

LLMs and hallucinations

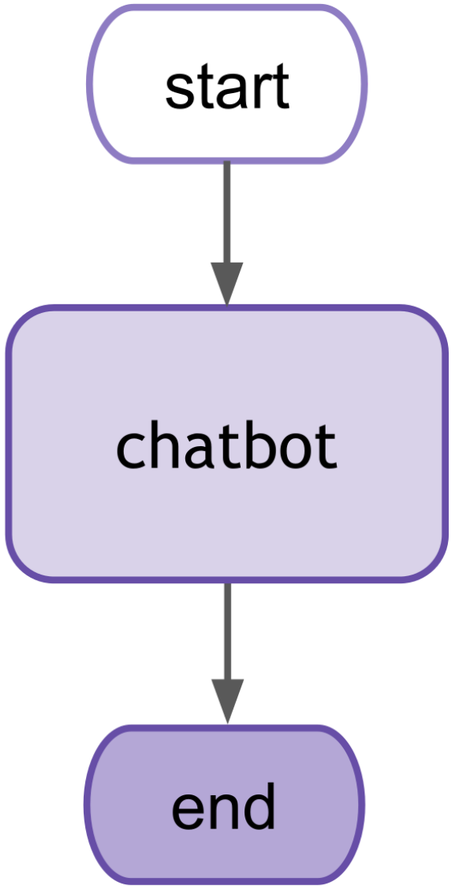
```
Agent: [AIMessage(content='Judith
Love Cohen was an American aerospace
engineer, and worked on various
space missions, including the Apollo
program... Cohen was also the mother
of actor and writer Adam Cohen,
who has spoken about her influence on
his life and career.
additional_kwargs={},
response_metadata={'finish_reason':
'stop', 'model_name':
'gpt-4o-mini-...)
```

Example hallucination

- Judith Love Cohen's famous son is Jack Black, not "Adam Cohen".

Generate a LangGraph diagram

```
# Import modules for chatbot diagram
from IPython.display import
Image, display

# Try generating and displaying
# the graph diagram
try:
    display(Image(graph.get_graph()
    .draw_mermaid_png()))

# Return an exception if necessary
except Exception:
    print("Additional dependencies
    required.")
```



Let's practice!

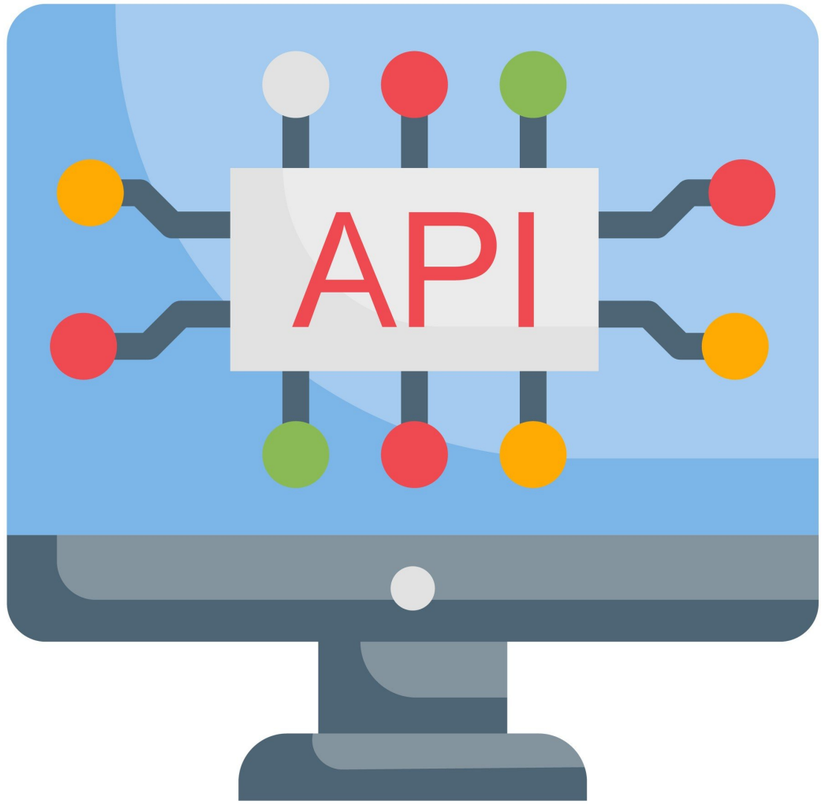
DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Adding external tools to a chatbot

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Dilini K. Sumanapala, PhD
Founder & AI Engineer, Genverv, Ltd.

Externals tools with LangGraph



- API tools for chatbots
  - News
  - Databases
  - Social media
  - Etc.

Adding a Wikipedia tool

Adding a Wikipedia tool

```
# Modules for building a Wikipedia tool
from langchain_community.utilities import WikipediaAPIWrapper
from langchain_community.tools import WikipediaQueryRun

# Initialize Wikipedia API wrapper to fetch top 1 result
api_wrapper = WikipediaAPIWrapper(top_k_results=1)

# Create a Wikipedia query tool using the API wrapper
wikipedia_tool = WikipediaQueryRun(api_wrapper=api_wrapper)
tools = [wikipedia_tool]
```

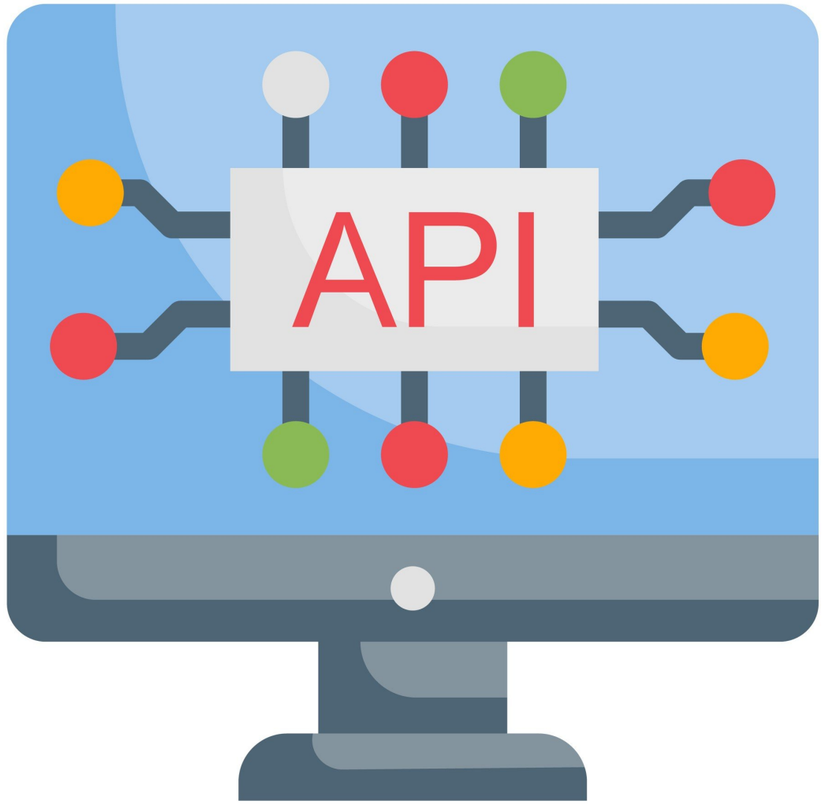
Adding a Wikipedia tool

```
# Bind the Wikipedia tool to
# the language model
llm_with_tools = llm.bind_tools(tools)


# Modify chatbot function to
# respond with Wikipedia
def chatbot(state: State):
    return {"messages":
    [llm_with_tools.invoke(
    state["messages"])]}
```

- Bind tools

- Update chatbot node
- Enable Wikipedia
- LLM decides tool calls

Other API tools



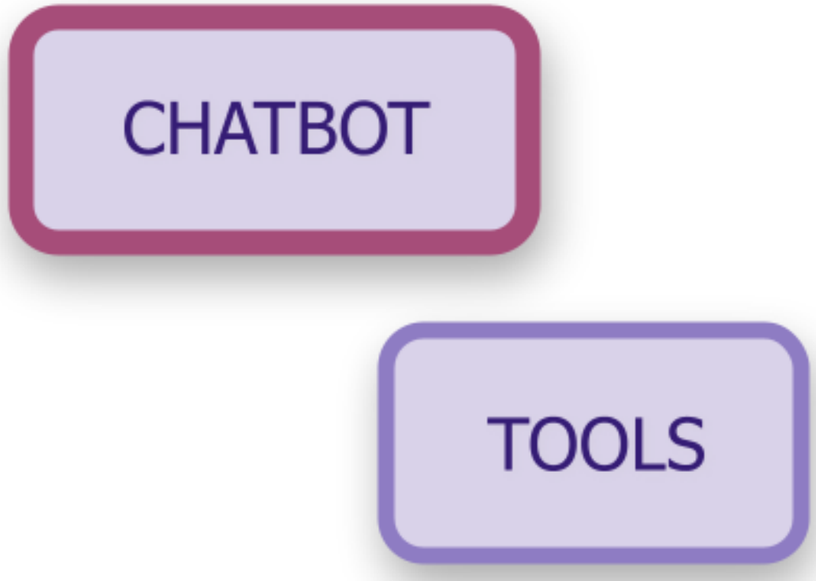
- LangChain API documentation

Adding tool nodes

```
# Modules for adding tool conditions
# and nodes
from langgraph.prebuilt import
ToolNode, tools_condition

# Add chatbot node to the graph
graph_builder.add_node("chatbot",
                       chatbot)
```

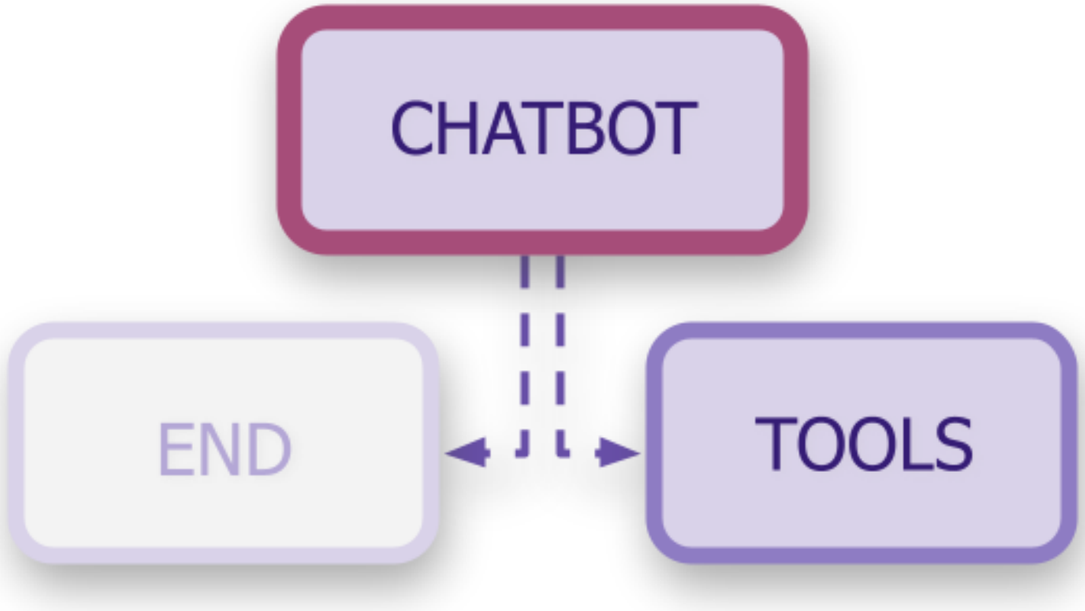
Adding tool nodes

```
# Modules for adding tool conditions
# and nodes
from langgraph.prebuilt import
ToolNode, tools_condition

# Add chatbot node to the graph
graph_builder.add_node("chatbot",
                       chatbot)

# Create a ToolNode to handle tool calls
# and add it to the graph
tool_node = ToolNode(tools=[wikipedia_tool])
graph_builder.add_node("tools", tool_node)
```



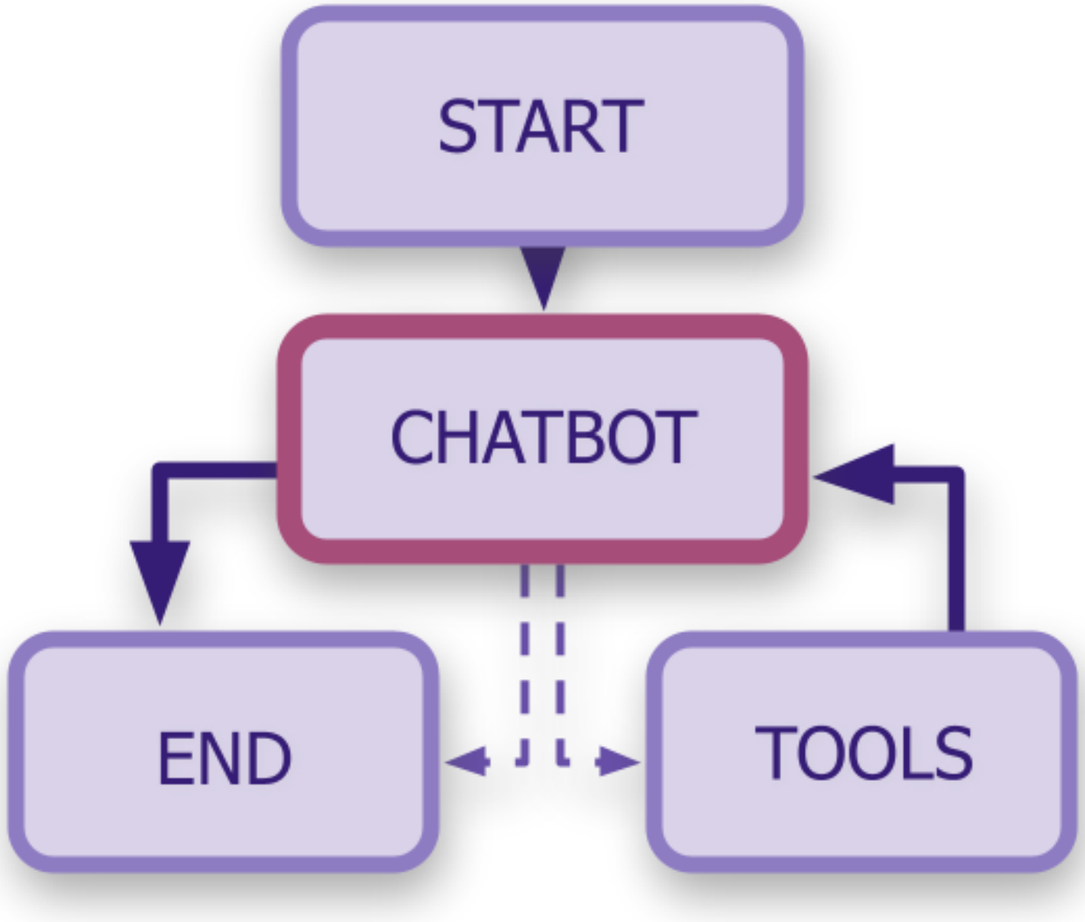
Adding tool nodes

```
# Set up a condition to direct from chatbot
# to tool or end node
graph_builder.add_conditional_edges(
                "chatbot", tools_condition)
```



Adding tool nodes

```
# Set up a condition to direct from chatbot
# to tool or end node
graph_builder.add_conditional_edges(
             "chatbot", tools_condition)


# Connect tools back to chatbot and
# add START and END nodes
graph_builder.add_edge("tools", "chatbot")
graph_builder.add_edge(START, "chatbot")
graph_builder.add_edge("chatbot", END)
```



Let's practice!

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

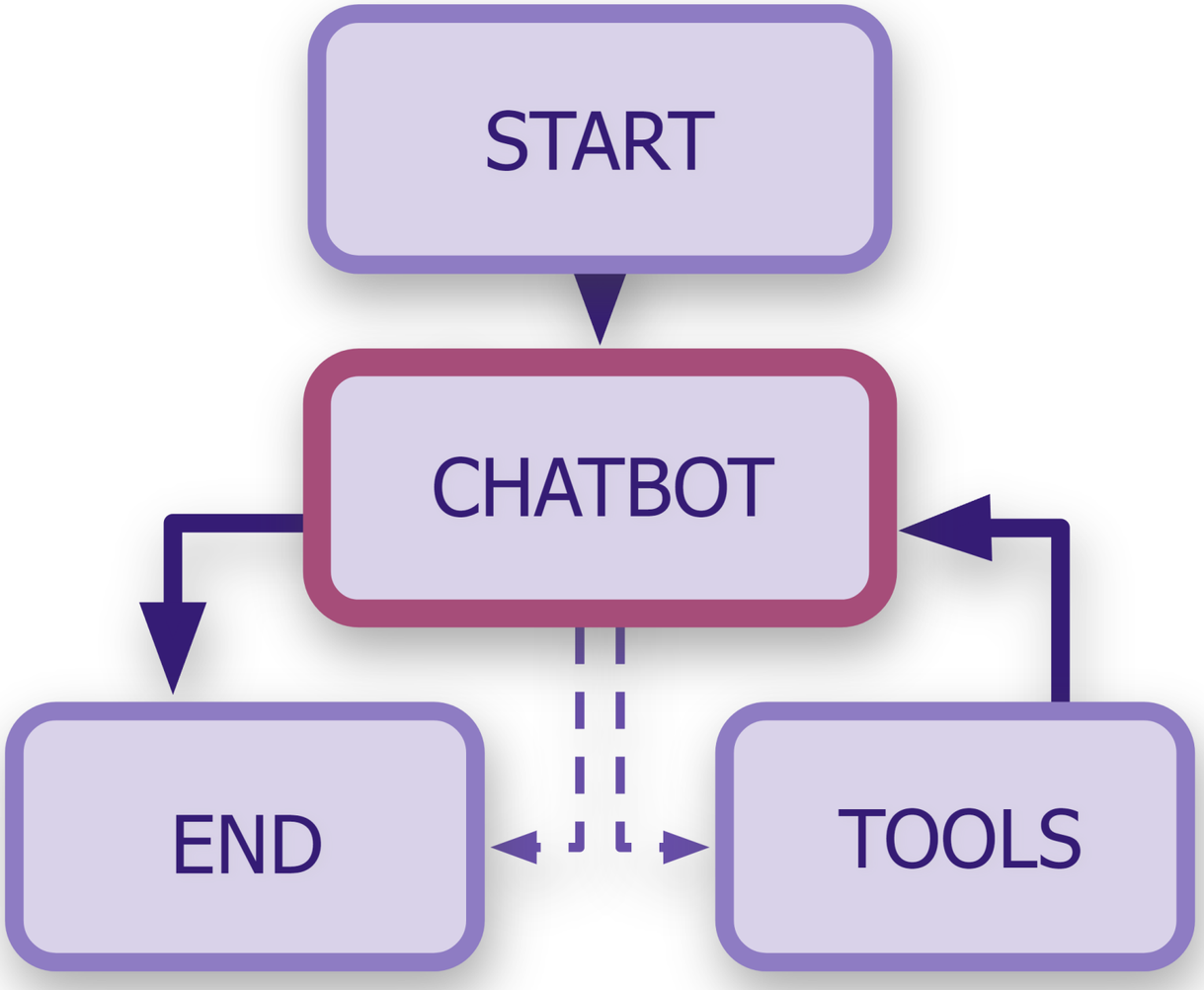
Adding memory and conversation

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Dilini K. Sumanapala, PhD
Founder & AI Engineer, Genverv, Ltd.

datacamp

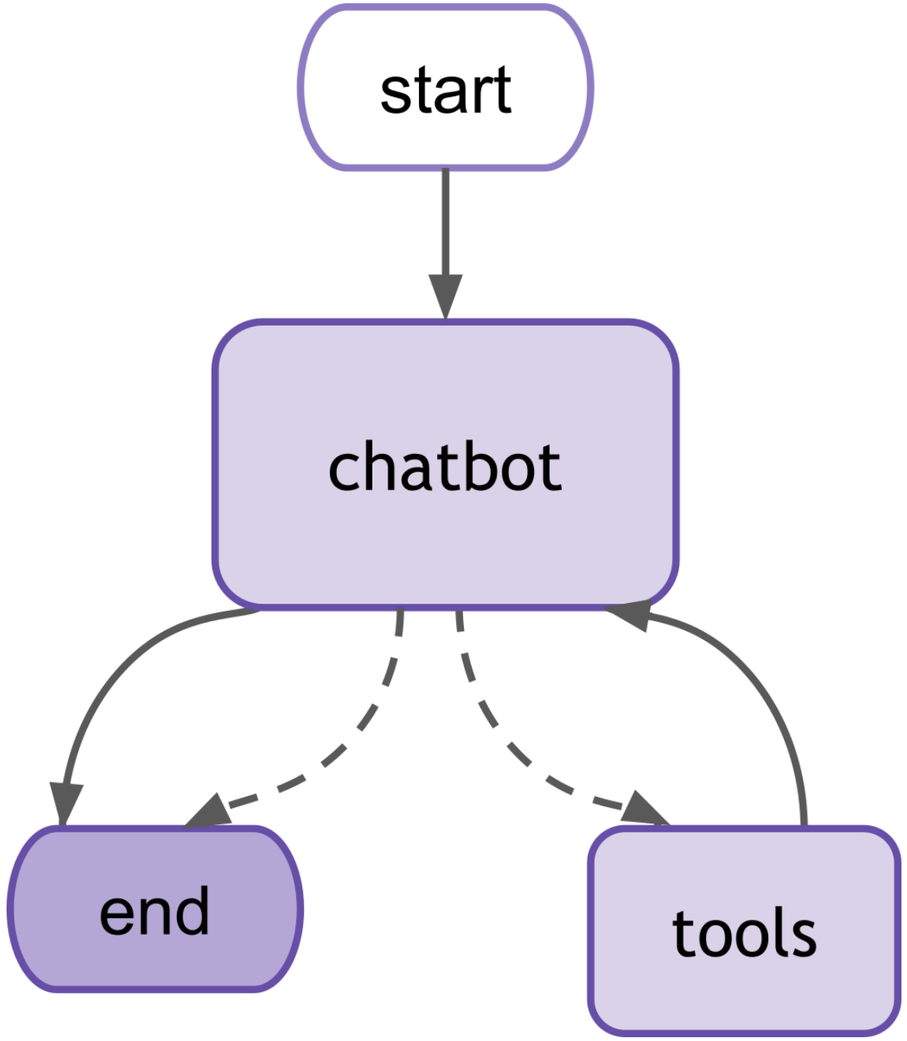
Testing tool use



Testing tool use

```
# Produce the chatbot graph
display(Image(app.get_graph().draw_mermaid_png()))

# Define a function to execute the chatbot, streaming each message
def stream_tool_responses(user_input: str):
    for event in graph.stream({"messages": [("user", user_input)]}):

        # Return the agent's last response
        for value in event.values():
            print("Agent:", value["messages"])

# Define the query and run the chatbot
user_query = "House of Lords"
stream_tool_responses(user_query)
```

Visualizing the diagram



Streaming the output

```
Agent: [AIMessage(content='', additional_kwargs={'tool_calls': [{'function': {'arguments':
'{"query":"House of Lords"}', 'name': 'wikipedia'}, 'type': 'function'}]},
response_metadata={'...'})]
Agent: [ToolMessage(content='Page: House of Lords\nSummary: The House of Lords is the upper
house of the Parliament of the United Kingdom. Like the lower house...The House of Lords
also has a Church of England role...', name='wikipedia', id='...',')]
Agent: [AIMessage(content='The House of Lords is the upper house of the Parliament
of the United Kingdom, located in the Palace of Westminster in London. It is one
of the oldest institutions in the world..., additional_kwargs={},
response_metadata=
{'finish_reason': 'stop', 'model_name': 'gpt-4o-mini-2024-07-18', 'system_fingerprint':
'fp_0ba0d124f1'}, id='run-ae3a0b4f-5b42-4f4e-9409-7c191af0b9c9-0')]
```

Adding memory

```
# Import the modules for saving memory
from langgraph.checkpoint.memory import MemorySaver

# Modify the graph with memory checkpointing
memory = MemorySaver()

# Compile the graph passing in memory
graph = graph_builder.compile(checkpointer=memory)
```

Streaming outputs with memory

```
# Set up a streaming function for a single user
def stream_memory_responses(user_input: str):
    config = {"configurable": {"thread_id": "single_session_memory"}}

    # Stream the events in the graph
    for event in graph.stream({"messages": [("user", user_input)]}, config):

        # Return the agent's last response
        for value in event.values():
            if "messages" in value and value["messages"]:
                print("Agent:", value["messages"])

stream_memory_responses("What is the Colosseum?")
stream_memory_responses("Who built it?")
```

Generating output with memory

```
stream_memory_responses("What is the Colosseum?")
```

```
Agent: [AIMessage(content='', additional_kwargs={'tool_calls': [{'index': 0,
'id': '...', 'function': {'arguments': '
{"query":"Colosseum"}', 'name': 'wikipedia'}, ...])]
Agent: [ToolMessage(content='Page: Colosseum\nSummary: The Colosseum is an ancient
amphitheatre in Rome, Italy. It is the largest standing amphitheatre
in the world..)]
Agent: [AIMessage(content='The Colosseum, located in Rome, is the largest ancient
amphitheatre still standing. Built under Emperor Vespasian and completed by his son
Titus, it hosted gladiatorial games and public events. It is also known as the Flavian
Amphitheatre due to its association with the Flavian dynasty.', additional_kwargs={},
response_metadata={'finish_reason': 'stop', 'model_name': 'gpt-4o-mini-...', ...')]
```

Generating output with memory

```
stream_memory_responses("Who built it?")
```

```
Agent: [AIMessage(content='The Colosseum was built by Emperor Vespasian around
72 AD and completed by his successor, Emperor Titus. Later modifications were
made by Emperor Domitian...')]
```

Let's practice!

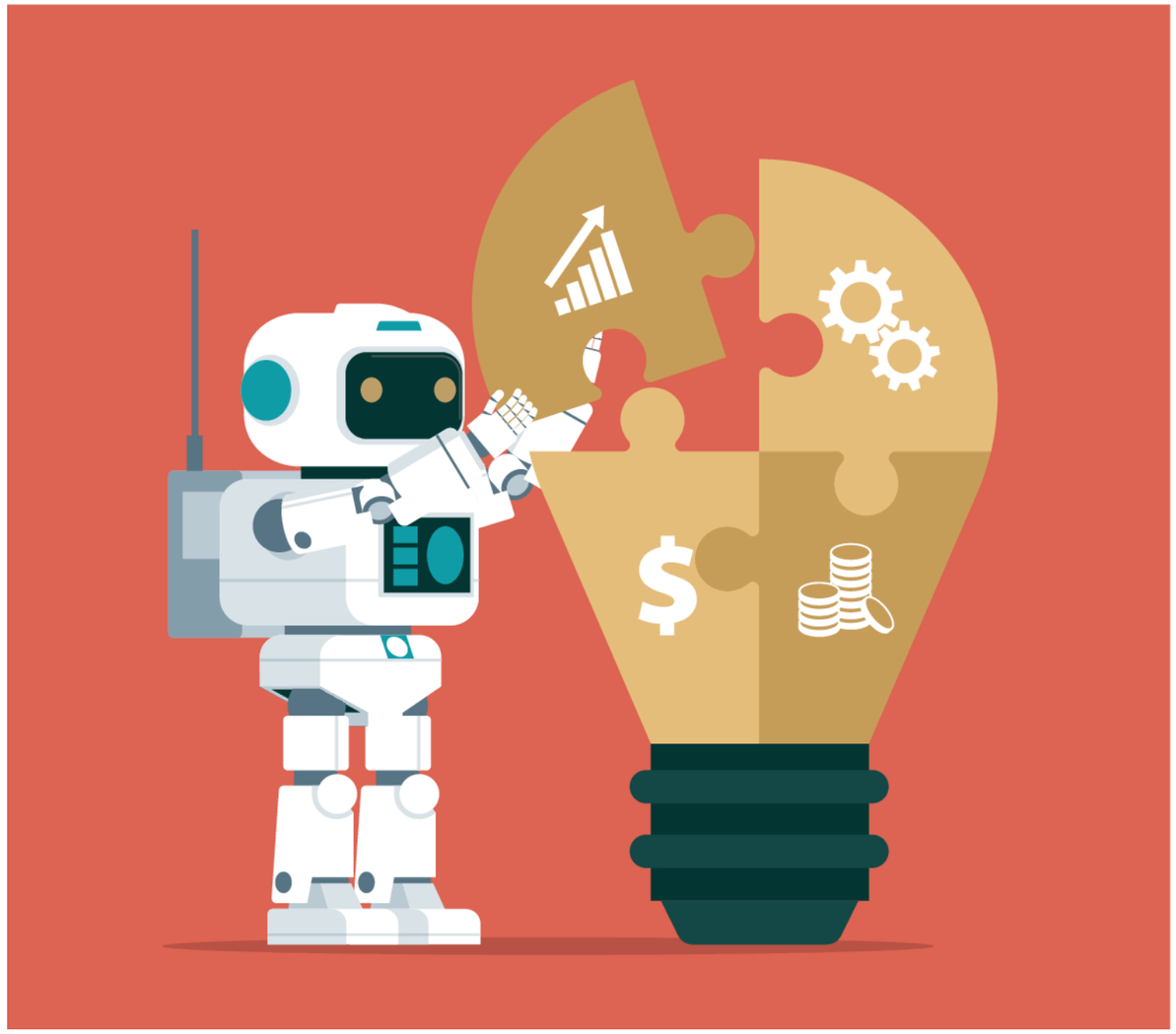
DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Defining multiple tools

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Dilini K. Sumanapala, PhD
Founder & AI Engineer, Genverv, Ltd.

Integrating multiple tools



- Add multiple custom tools
- Automate tool selection per query

Enhancing an education chatbot

- Look up historical events
- Check palindromes:
  - level = level
  - top spot = tops pot

Multiple ways to build tools

- **Invoke the LLM**

  Look up historical dates using natural language inputs, eg. "5th of November"

```
Agent: The 5th of November is famous for the Gunpowder treason and
plot...
```

- **Python code**

  Find palindromes by directly comparing strings, eg. `string == string[::-1]`

```
Agent: Yes, "madam" is a palindrome...
```

Historical events tool

```
# Use a decorator to label the tool and set the input format to string
@tool
def date_checker(date: str) -> str:
    """Provide a list of important historical events for a given date in any format."""
    try:
          # Invoke the LLM to interpret the date and generate historical events
        answer = llm.invoke(f"List important historical events that occurred on {date}.")

        # Return the response
        return answer.content

    # Set an exception block for errors in retrieval
    except Exception as e:
        return f"Error retrieving events: {str(e)}"
```

Palindrome tool

```
@tool
# Set input format to string
def check_palindrome(text: str):
    """Check if a word or phrase is a palindrome."""

    # Remove non-alphanumeric characters and convert to lowercase
    cleaned = ''.join(char.lower() for char in text if char.isalnum())

    # Check if the reversed text is the same as original text
    if cleaned == cleaned[::-1]:
        return f"The phrase or word '{text}' is a palindrome."
    else:
        return f"The phrase or word '{text}' is not a palindrome."
```

Binding multiple tools

```
# Import modules required for defining tool nodes
from langgraph.prebuilt import ToolNode

# List of tools
tools = [wikipedia_tool, date_checker, check_palindrome]

# Pass the tools to the ToolNode()
tool_node = ToolNode(tools)

# Bind tools to the LLM
model_with_tools = llm.bind_tools(tools)
```

Let's practice!

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Defining nodes and edges for flexible function calling

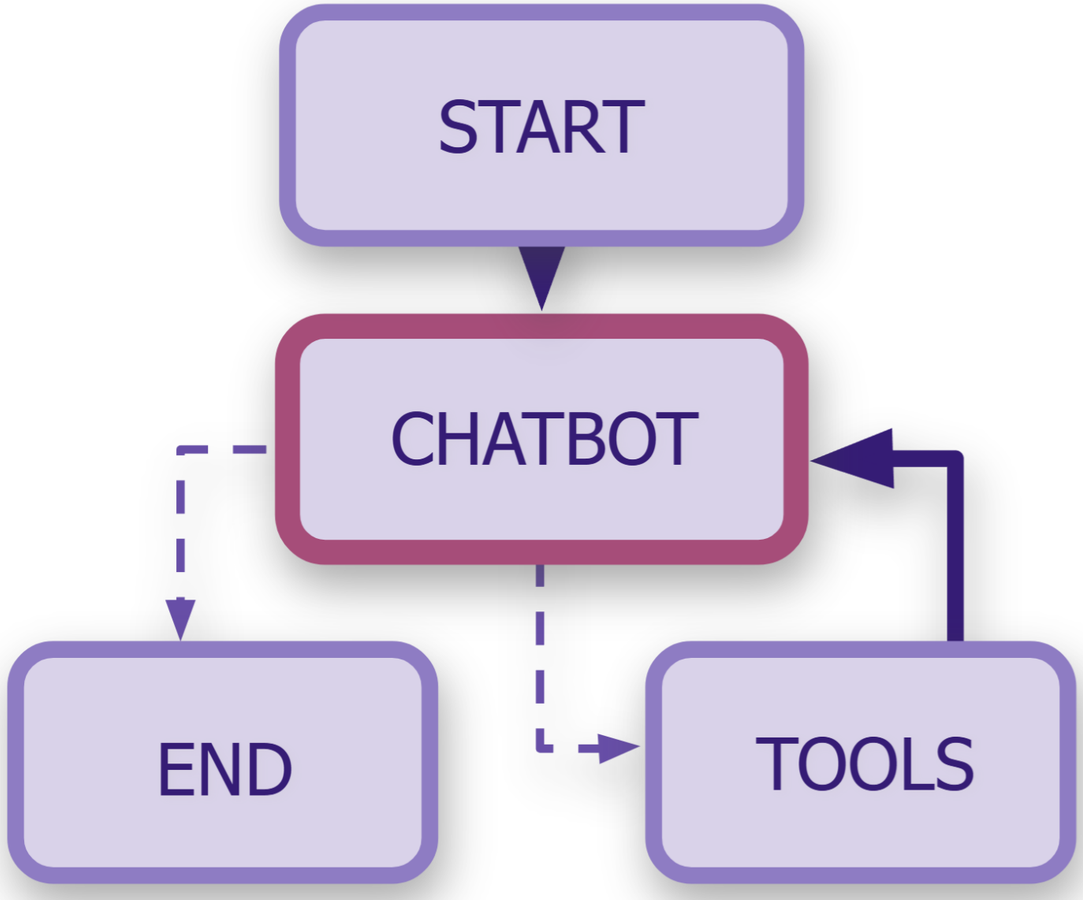
DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Dilini K. Sumanapala, PhD
Founder & AI Engineer, Genverv, Ltd.

datacamp

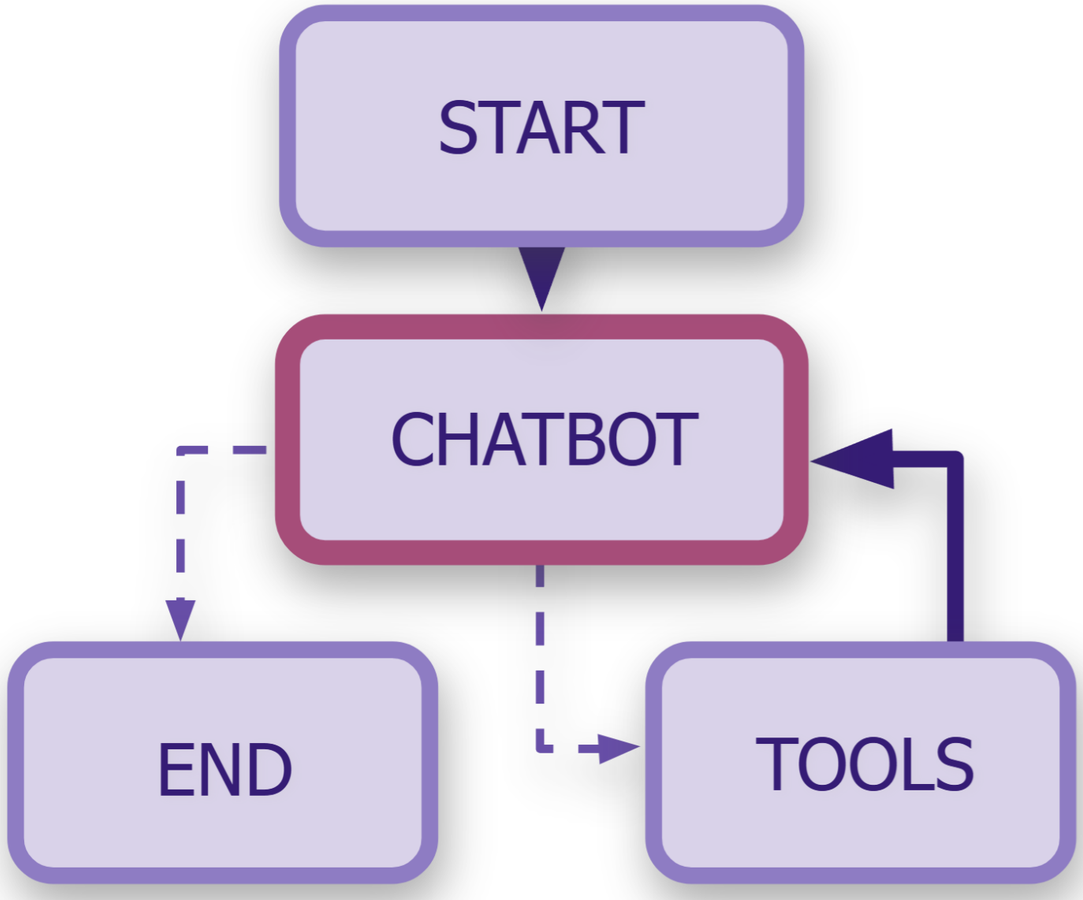
Building a multi-tool workflow

- Multiple available tools
  - Palindrome
  - Historical events
  - Wikipedia



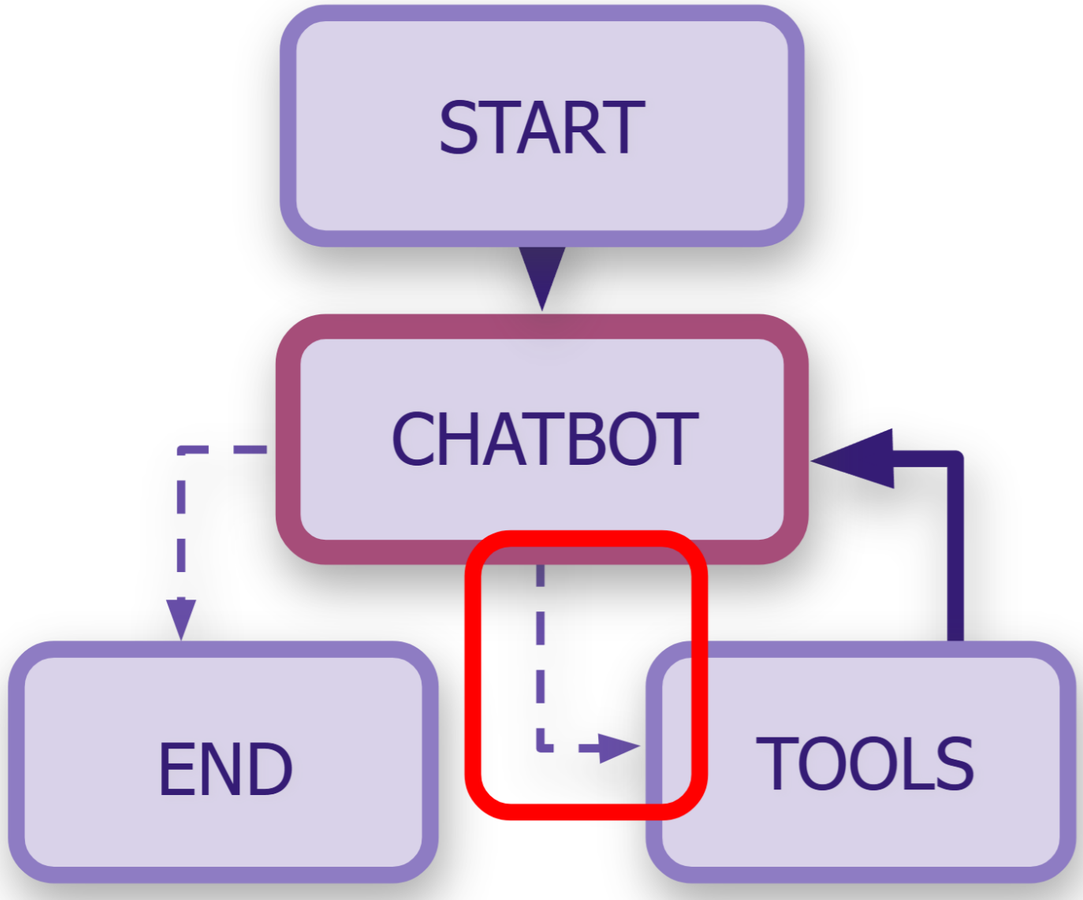
Define workflow functions

- Create a stopping function



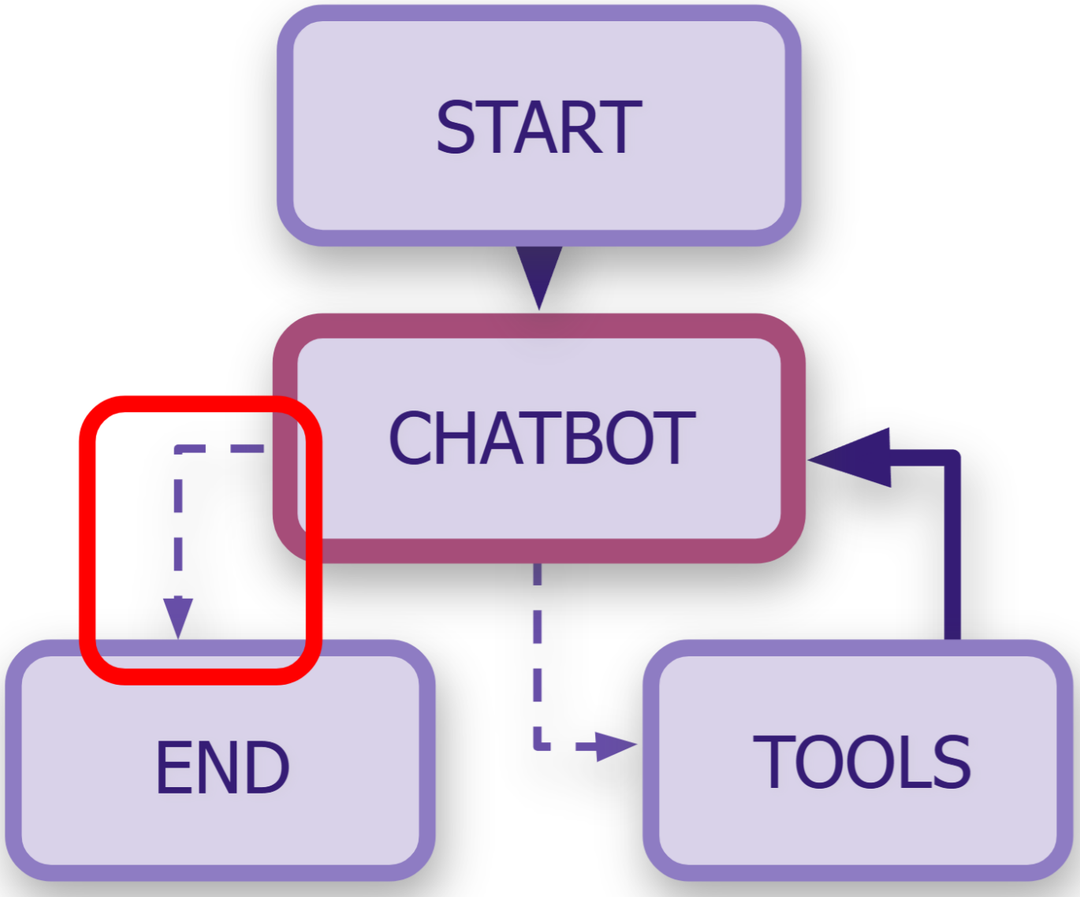
Define workflow functions

- **Create a stopping function**
  - Check for tool calls



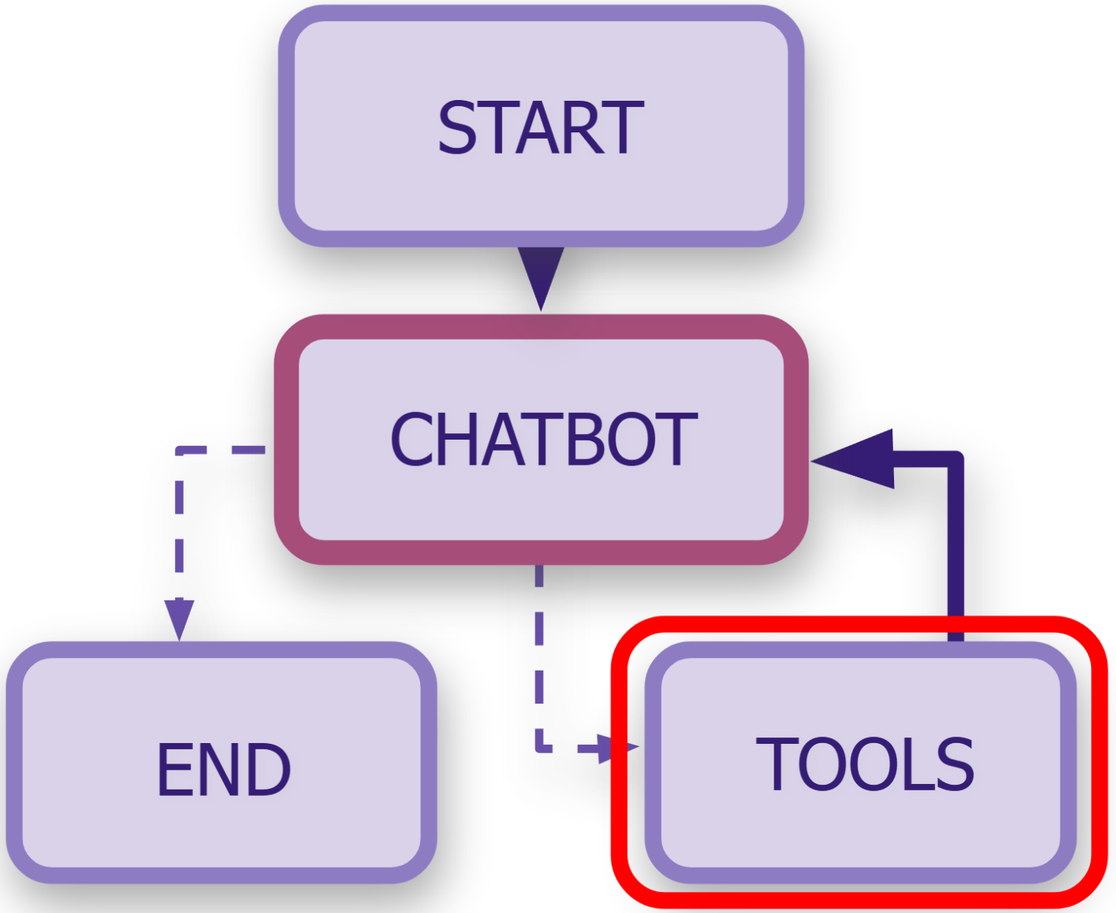
Define workflow functions

- **Create a stopping function**
  - Check for tool calls
  - End conversation if none



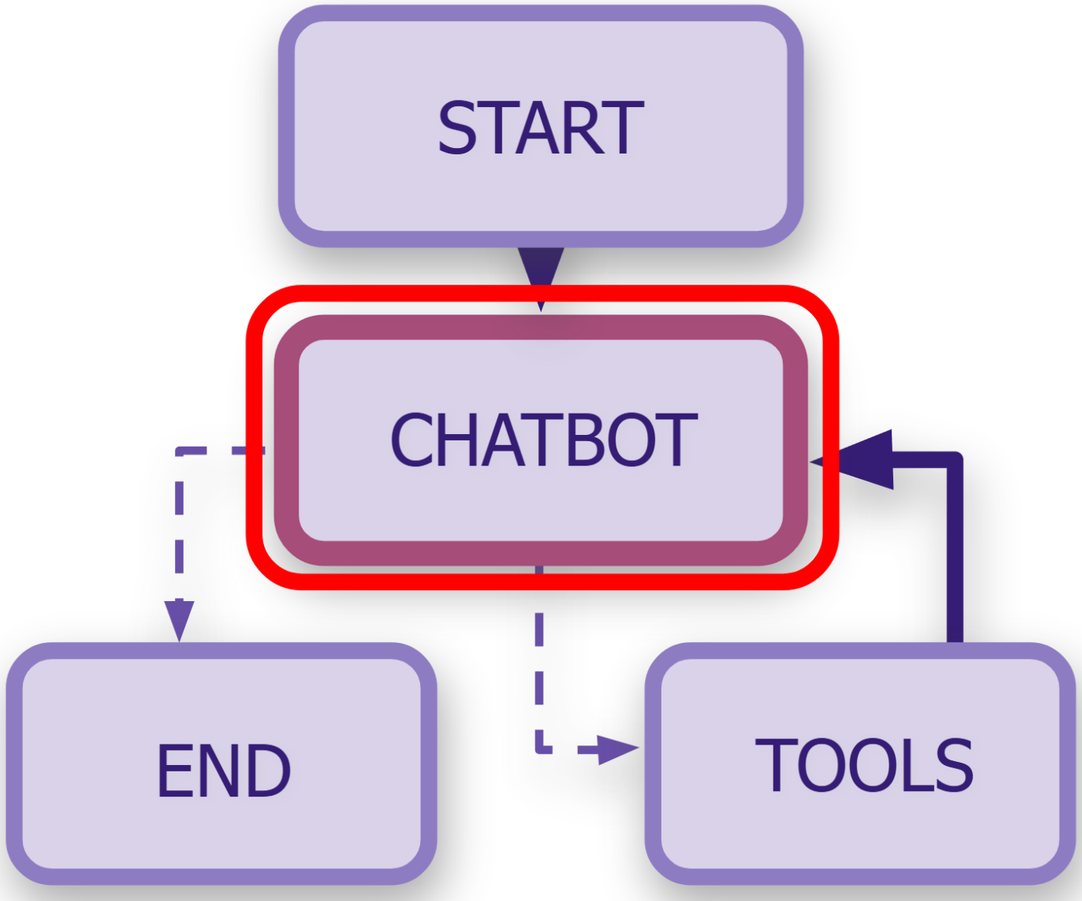
Define workflow functions

- **Create a stopping function**
  - Check for tool calls
  - End conversation if none
- **Create a dynamic tool caller**
  - Return a tool response if tool call present



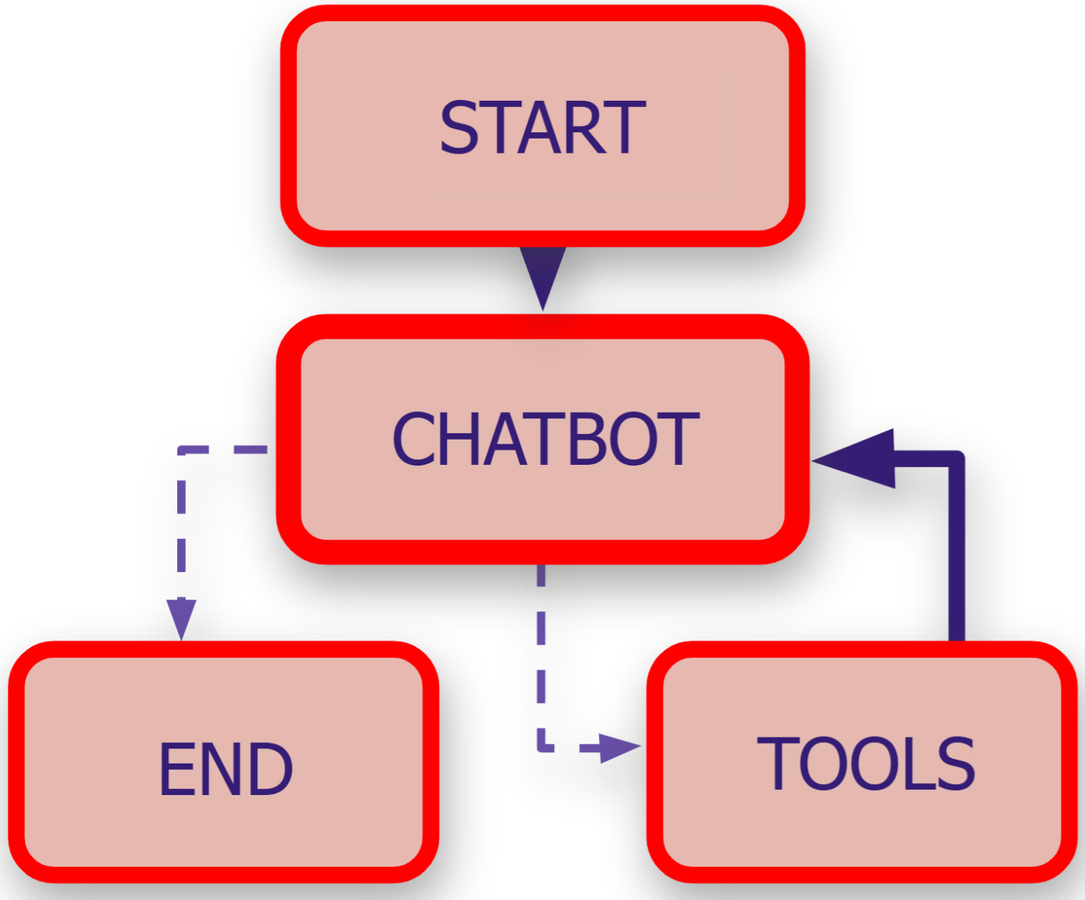
Define workflow functions

- **Create a stopping function**
  - Check for tool calls
  - End conversation if none
- **Create a dynamic tool caller**
  - Return a tool response if tool call present
  - Invoke the LLM with **just** the chatbot node if no tool calls



Define workflow functions

- **Create a stopping function**
  - Check for tool calls
  - End conversation if none
- **Create a dynamic tool caller**
  - Return a tool response if tool call present
  - Invoke the LLM with **just** the chatbot node if no tool calls
- **Compile the full graph**



Create a stop condition function

```
from langgraph.graph import MessagesState, START, END

# Use MessagesState to define the state of the stopping function
def should_continue(state: MessagesState):

    # Get the last message from the state
    last_message = state["messages"][-1]

    # Check if the last message includes tool calls
    if last_message.tool_calls:
        return "tools"

    # End the conversation if no tool calls are present
    return END
```

Create a dynamic tool caller

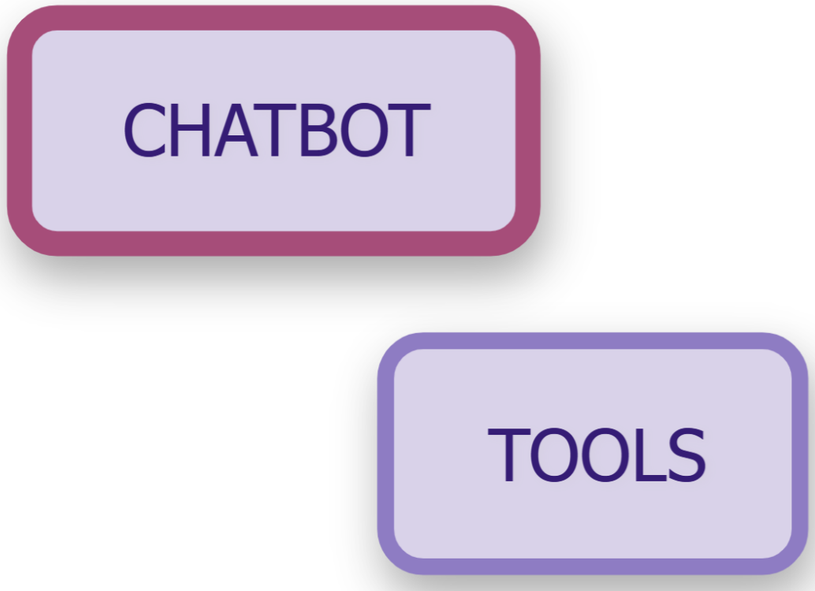
```
# Extract the last message from the history
def call_model(state: MessagesState):
    last_message = state["messages"][-1]

    # If the last message has tool calls, return the tool's response
    if isinstance(last_message, AIMessage) and last_message.tool_calls:

        # Return the messages from the tool call
        return {"messages": [AIMessage(content=last_message.tool_calls[0]["response"])]}

    # Otherwise, proceed with a regular LLM response
    return {"messages": [model_with_tools.invoke(state["messages"])]}
```

Create the graph

```
workflow = StateGraph(MessagesState)
```

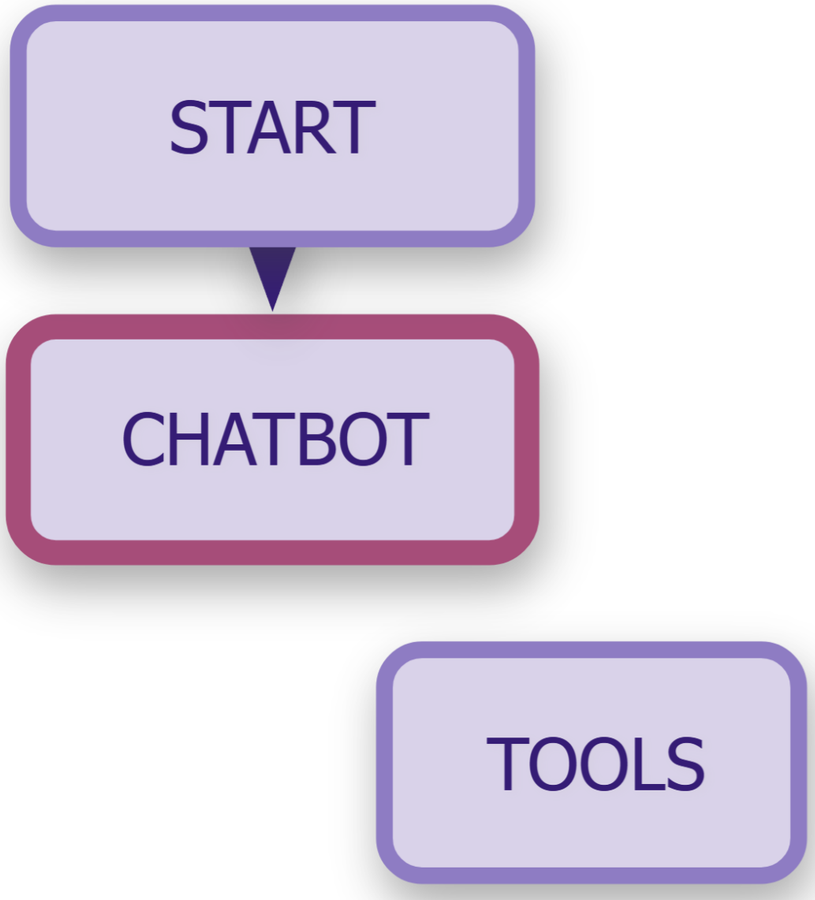
Create the graph

```
workflow = StateGraph(MessagesState)

# Add nodes for chatbot and tools
workflow.add_node("chatbot", call_model)
workflow.add_node("tools", tool_node)
```



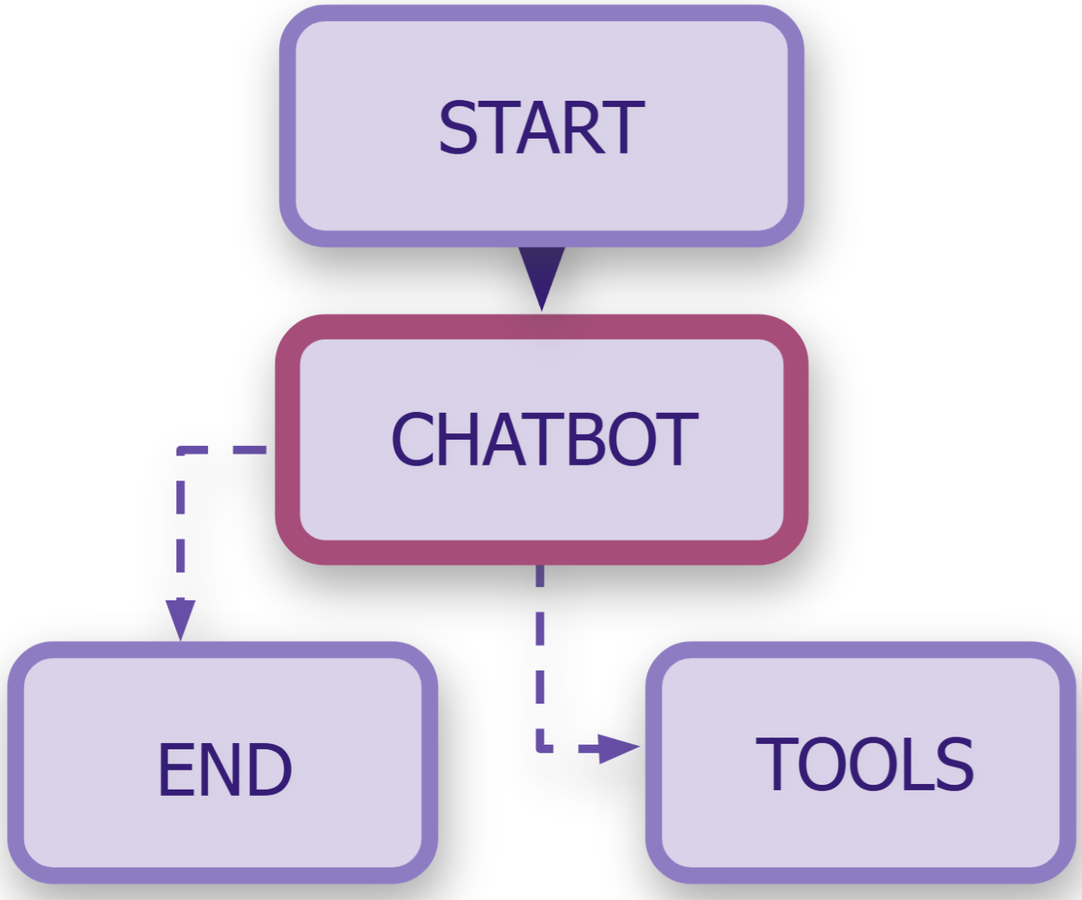
Create the graph

```
workflow = StateGraph(MessagesState)

# Add nodes for chatbot and tools
workflow.add_node("chatbot", call_model)
workflow.add_node("tools", tool_node)

# Connect the START node to the chatbot
workflow.add_edge(START, "chatbot")
```



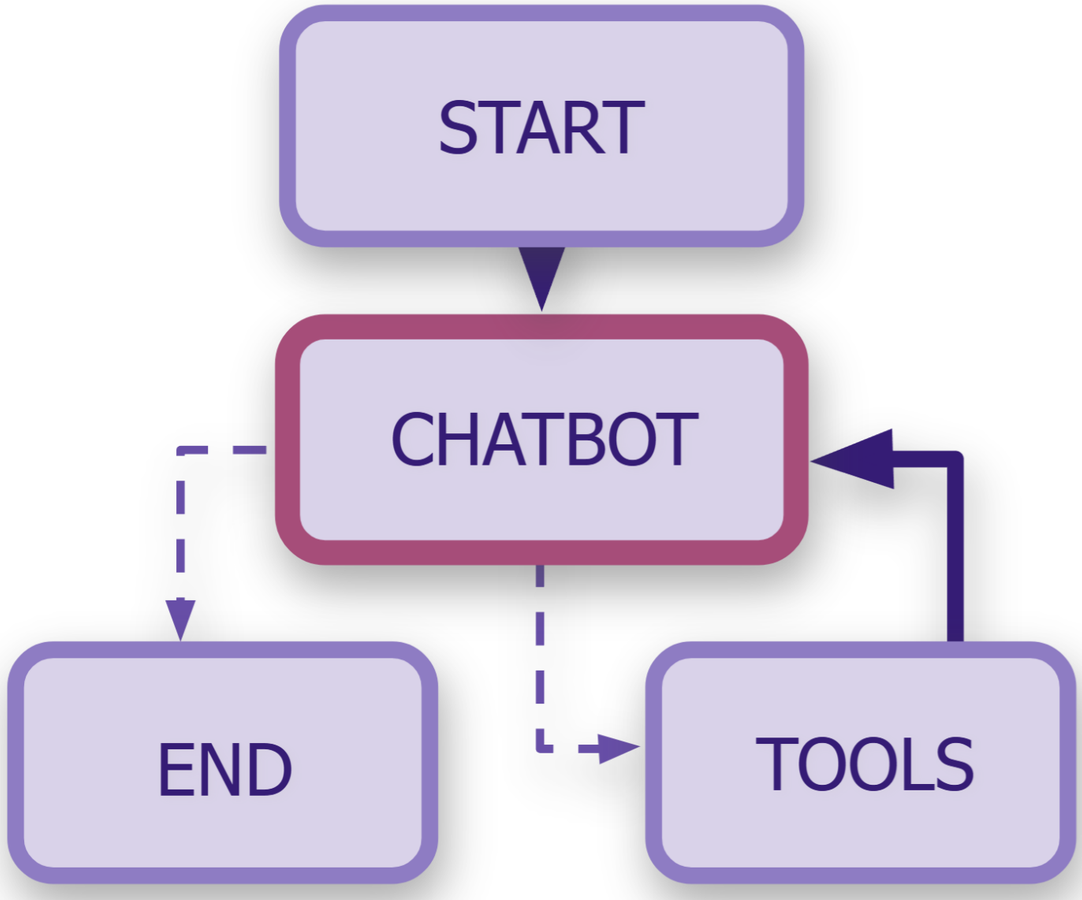
Create the graph

```
workflow = StateGraph(MessagesState)

# Add nodes for chatbot and tools
workflow.add_node("chatbot", call_model)
workflow.add_node("tools", tool_node)

# Connect the START node to the chatbot
workflow.add_edge(START, "chatbot")

# Define conditions, then loop back to chatbot
workflow.add_conditional_edges("chatbot",
         should_continue, ["tools", END])
```



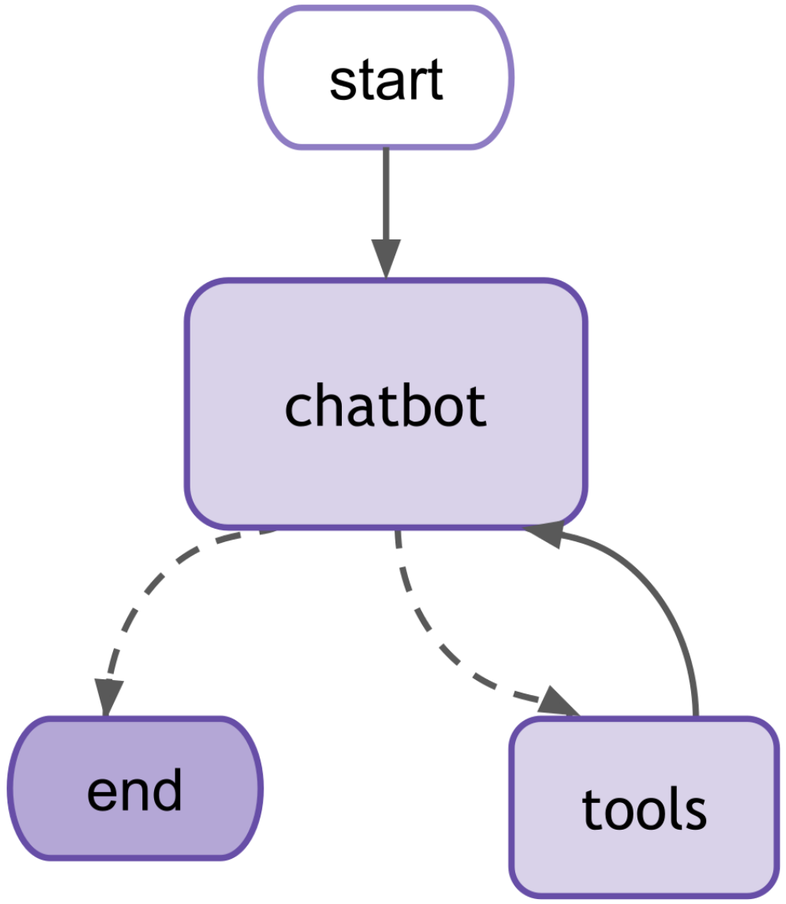
Create the graph

```
workflow = StateGraph(MessagesState)

# Add nodes for chatbot and tools
workflow.add_node("chatbot", call_model)
workflow.add_node("tools", tool_node)

# Connect the START node to the chatbot
workflow.add_edge(START, "chatbot")

# Define conditions, then loop back to chatbot
workflow.add_conditional_edges("chatbot",
         should_continue, ["tools", END])
workflow.add_edge("tools", "chatbot")
```



Adding memory

```
# Set up memory and compile the workflow
memory = MemorySaver()
app = workflow.compile(
                checkpointer=memory)

display(Image(app.get_graph()
        .draw_mermaid_png()))
```



Let's practice!

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

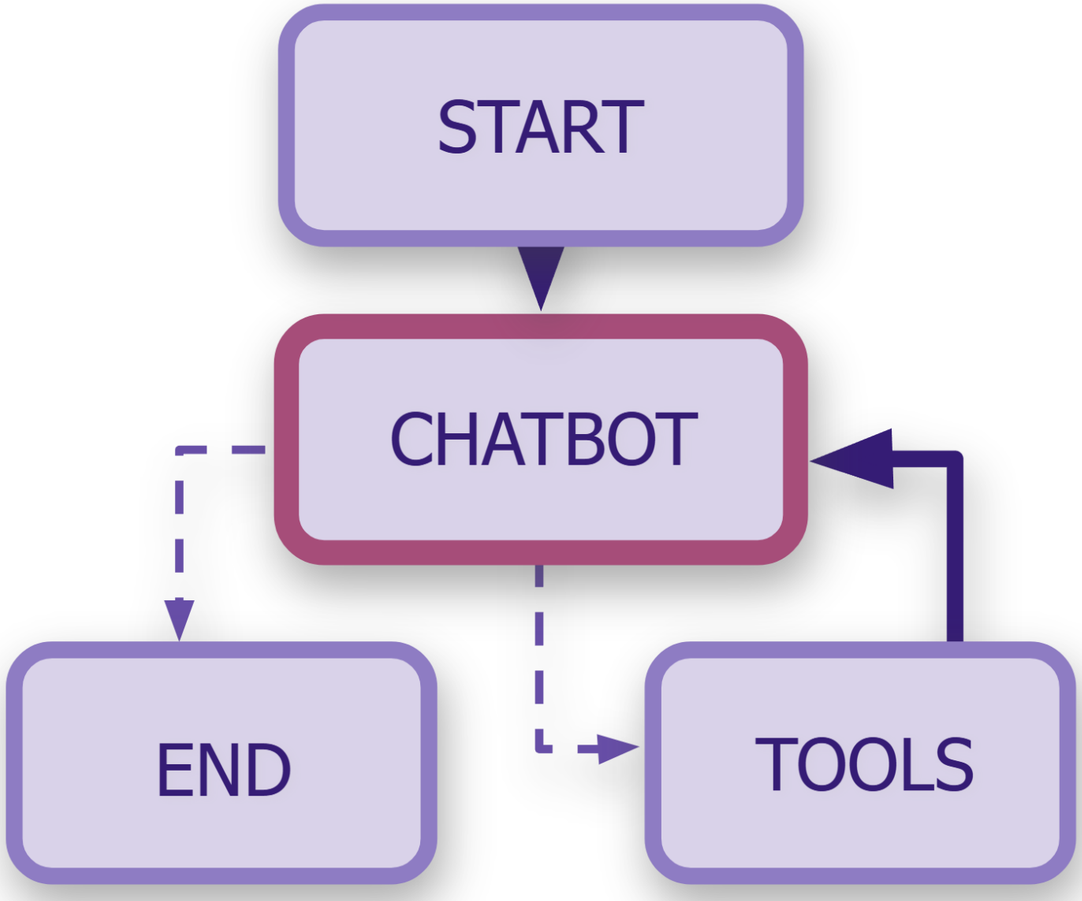
Organize chatbot outputs with memory

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Dilini K. Sumanapala, PhD
Founder & AI Engineer, Genverv, Ltd.

datacamp

Streaming multiple tool outputs



- **Print outputs with multiple tools**

```
- Agent: [Tool 1: answer]
- Agent: [Tool 2: answer]
```

- **Print user query and enable memory**

```
- User: [Query]
- Agent: [Tool 1: answer]

- User: [Follow-up query]
- Agent: [Tool 1: follow-up answer]
```

Streaming multiple tool outputs

```
from langchain_core.messages import AIMessage, HumanMessage

config = {"configurable": {"thread_id": "1"}}

# Create input message with the user's query
def multi_tool_output(query):
    inputs = {"messages": [HumanMessage(content=query)]}

    # Stream messages and metadata from the chatbot application
    for msg, metadata in app.stream(inputs, config, stream_mode="messages"):

        # Check if the message has content and is not from a human
        if msg.content and not isinstance(msg, HumanMessage):
            print(msg.content, end="", flush=True)
    print("\n")
```

Test with multiple tools

Check dynamic tool assignment

```
multi_tool_output("Is `Stella won no wallets` a palindrome?")
multi_tool_output("What happened on April 12th, 1955?")
```

```
The phrase or word 'Stella won no wallets' is a palindrome.
Yes, the phrase "Stella won no wallets" is a palindrome.

April 12th, 1955, is notable for several reasons, particularly in the context of science and
technology. On this date, the first successful polio vaccine developed by Dr. Jonas Salk
was announced to the public. This vaccine was a significant breakthrough in the fight against
poliomyelitis, a disease that had caused widespread fear and numerous outbreaks, particularly
affecting children. The announcement marked a turning point in public health and led to
widespread vaccination campaigns that ultimately contributed to the near-eradication of polio...
```

Follow-up questions with multiple tools

```
# Print the user query first for every interaction
def user_agent_multiturn(queries):
    for query in queries:
        print(f"User: {query}")

        # Stream through messages corresponding to queries, excluding metadata
        print("Agent: " + "".join(msg.content for msg, metadata in app.stream(
                {"messages": [HumanMessage(content=query)]}, config, stream_mode="messages")

            # Filter out the human messages to print agent messages
            if msg.content and not isinstance(msg, HumanMessage)) + "\n")

queries = ["What happened on the 12 April 1961?", "What about 10 December 1948?",
"Is `Mr. Owl ate my metal worm?` a palindrome?", "What about 'palladium stadium?'"]
user_agent_multiturn(queries)
```

Full conversation output

- Historical events

```
User: What happened on the 12 April 1961?
Agent: On April 12, 1961, Yuri Gagarin, a Soviet cosmonaut, became the first human to travel
into space aboard the Vostok 1 spacecraft...

User: What about 22 November 1963?
Agent: December 10, 1948... marks the Universal Declaration of Human Rights (UDHR).
```

- Palindrome check

```
User: Is `Mr. Owl ate my metal worm?` a palindrome?
Agent: The phrase or word 'Mr. Owl ate my metal worm?' is a palindrome...

User: What about 'palladium stadium'?
Agent: No, the phrase `palladium stadium` is not a palindrome...
```

Let's practice!

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Congratulations!

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN

Dilini K. Sumanapala, PhD
Founder & AI Engineer, Genverv, Ltd.

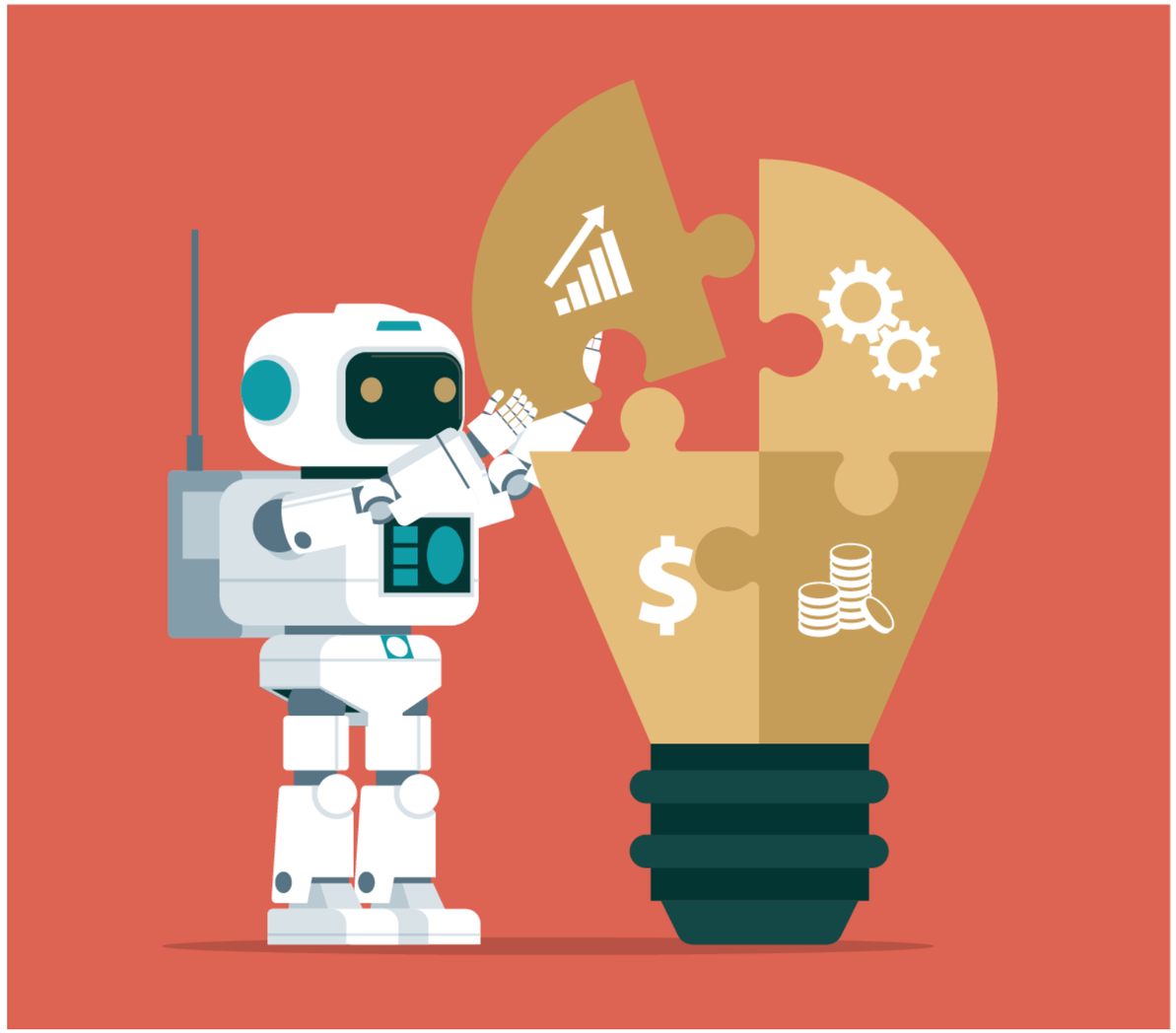
Essentials of LangChain agents

- Created ReAct agents
- Enhanced agents with tools
- Enabled follow-up questions
- Reasoning and LLMs
- Natural language responses

Building chatbots with LangGraph

- **Integrate external APIs**
  - eg. Wikipedia
- **Define**
  - Graph and agent states
  - Nodes
  - Edges
- **Memory**
- **Conversation**

Building dynamic chatbots



Advanced features

- Switching between tools
- Dynamic LLM calls
- Create multiple tools
- Flexible workflows
- Multiple tool memory
- Multi-turn conversation
- Large workloads, eg.
  - School curriculum

The LangChain ecosystem

- AI workflows
  - Scalable
  - Flexible
- Building foundational agents
- Integrating tools
- Advanced memory

The LangChain ecosystem

- **Real world settings**
  - **LangSmith**
    debugging and evaluation
  - **LangGraph**
    agent customization
  - **LangGraph Platform**
    agent deployment
- **Documentation**
  - LangChain products
  - LangGraph and LangGraph Platform

Thank you!

DESIGNING AGENTIC SYSTEMS WITH LANGCHAIN