VISIUM

# Building Agents with LangGraph: From GPT to RAG and Multi-Agent Systems

AMLD 2025 – Victor Gillioz, Gaétan Ramet – 14.02.2025

# Your hosts for today

VISIUM

**Victor Gillioz**

ML Engineer @ Visium

**Gaétan Ramet**

Sr. ML Engineer @ Visium

# Table of contents

VISIUM

# INTRODUCTION

Why should we care about agentic AI?

# The future is Agentic

## What is Agentic AI

AI systems designed to act as autonomous agents, capable of performing tasks — all with minimal human supervision.

**Autonomy:** The ability to operate independently once given an objective

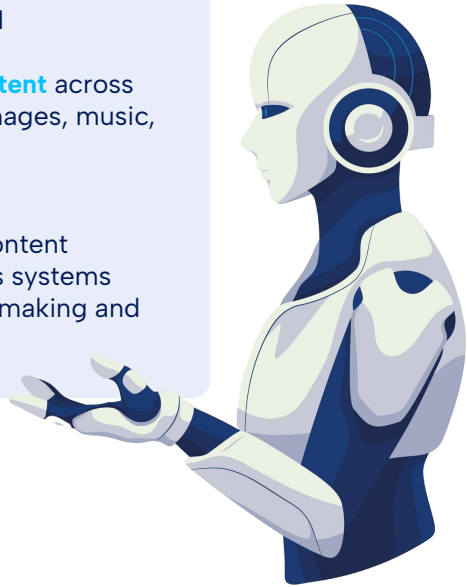
**Adaptability:** The ability to adjust strategies or next actions depending on the situation

**Reinforcement Learning:** The ability to learn from actions and improve over time

### GenAI vs Agentic AI

GenAI excels at **creating new content** across various formats, including text, images, music, and even code.

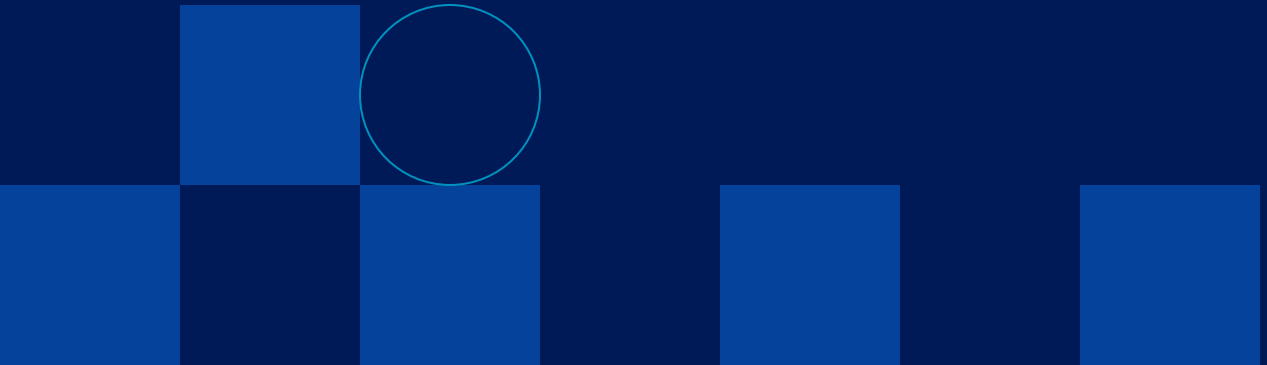
Agentic AI, on the other hand, is **action-oriented**, going beyond content creation to empower autonomous systems capable of independent decision making and actions.



VISIUM

# CONCEPTS

Semi-formal definitions of Agents, Workflows, Tools, etc...



# LLMs and Tools

## Standard LLMs and the Augmented LLM

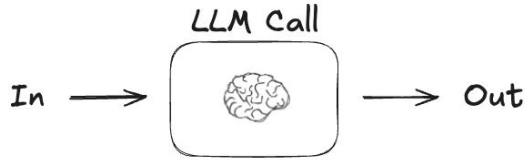
Standard LLMs can be augmented through access to **Tools**:

- Document retrieval
- Web search
- Code Executor
- …

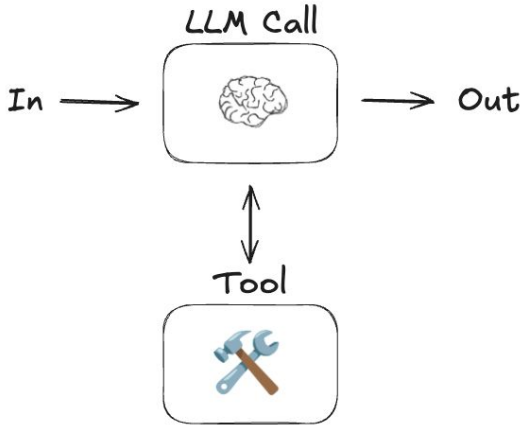
By informing the LLM about available tools and how they work, it can decide independently when to use them.

These **augmented LLM**s often serve as the basis for **Agentic Systems**.

Standard LLM




Augmented LLM



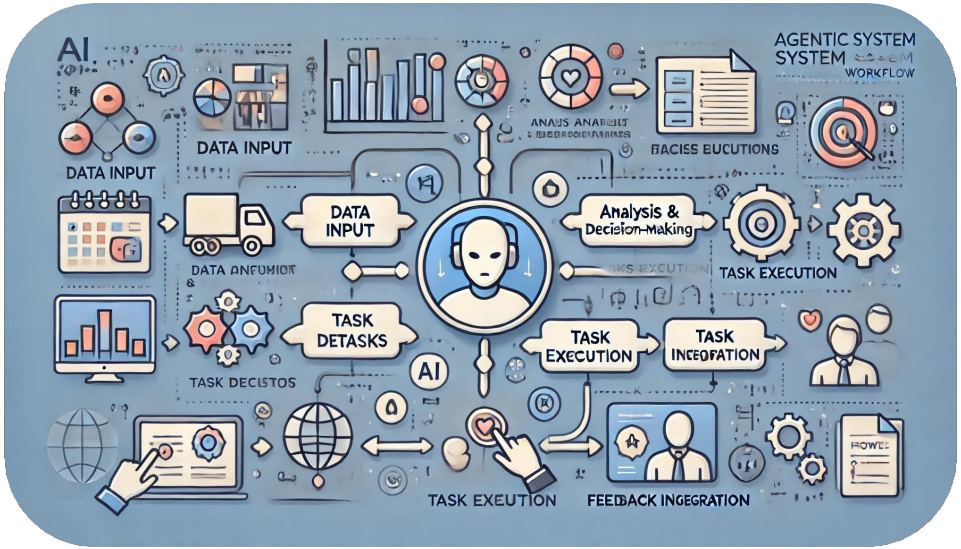
# Workflows

Predefined communication flow between LLMs

Workflows are systems where the **execution path** of LLMs and tools is **predefined** to some extent.

They are useful for tasks that can be decomposed into clear actions.

Workflows offer **control**, **predictability**, and **consistency** for complex tasks that require more than single LLM calls.



# Example: Prompt Chaining

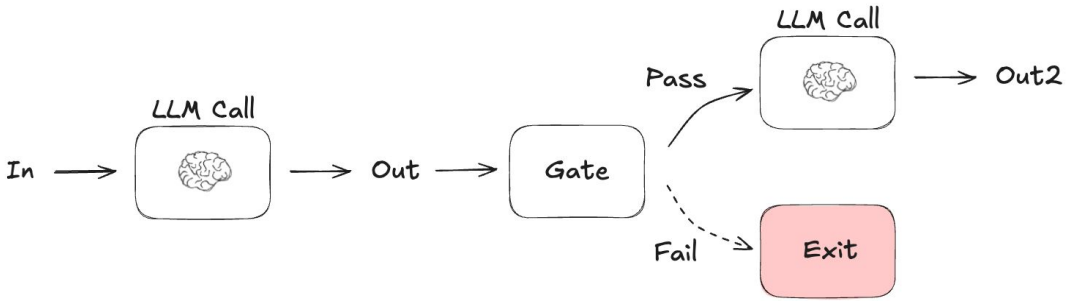
Split a task into small steps

Prompt Chaining is useful when the task can be decomposed into a **sequence of easier sub-tasks**.

It takes longer than single LLM calls but can help improve the output quality, and can include checks throughout the execution.

Practical example:

- Write a document, then translate it to another language.



# Example: Routing

LLM for routing

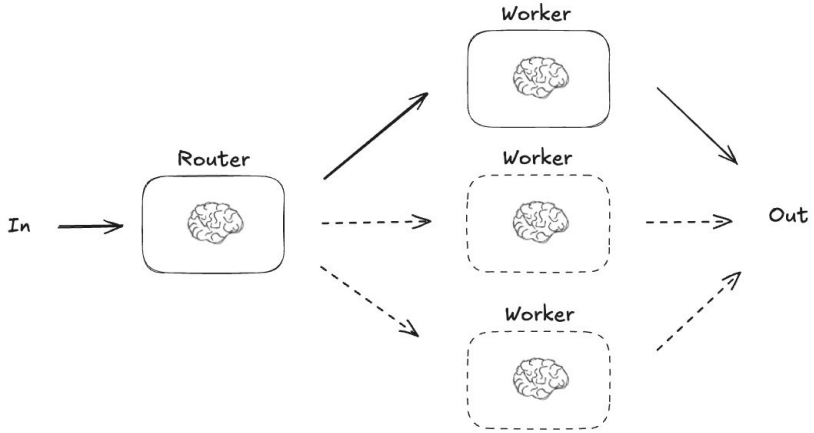
**Routing** offers more dynamic workflows by having an LLM decide which specialized LLM to call for different requests.

In this setting we have:

- A **Router** dispatching requests
- **Workers** with specialized roles

Practical example:

- Direct client requests to specialized LLMs based on their content.



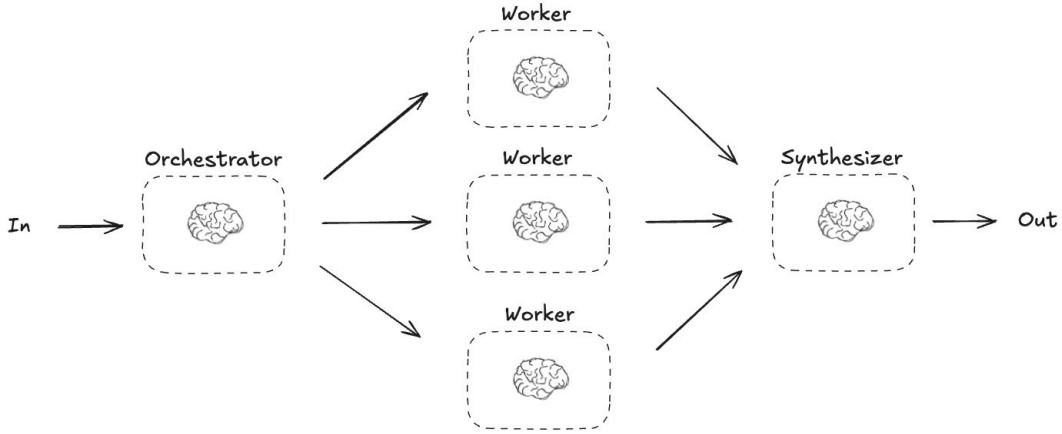
# Example: Parallelization / Orchestrator

LLM as a project manager

We can create more flexible workflows by having an Orchestrator **distribute tasks**, pre-defined or **not**, sometimes with a dynamic number of workers defined at run time.

In this setting we have:

- An **Orchestrator** distributing tasks
- Multiple **Worker**s that receive tasks
- An **Aggregator/Synthesizer** in charge of aggregating the Experts' outputs and formatting the final output



# Example: Evaluator–Optimizer

LLMs in a loop

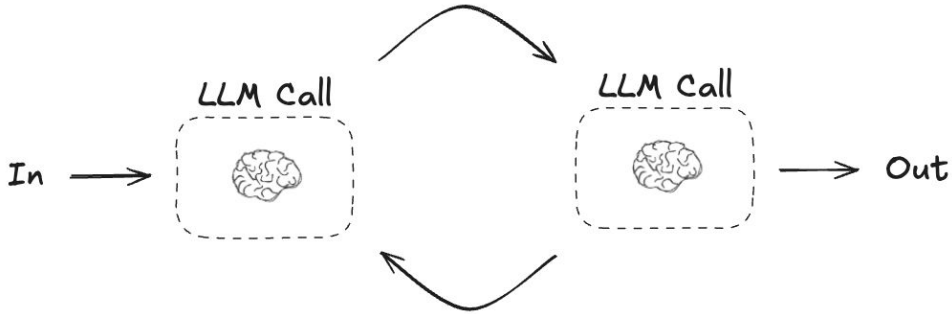
LLMs can also be chained **back and forth** to iterate on a solution until a satisfactory output is reached.

In this setting we have:

- A **Generator** creating content
- An **Evaluator** providing feedback

Practical example:

- Iterations for code development.



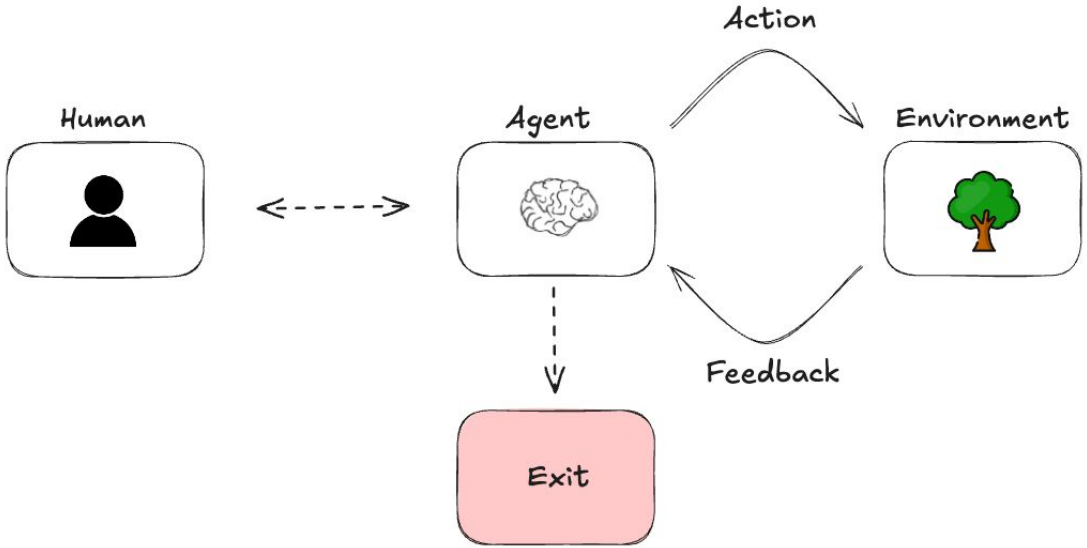
# So what's an Agent?

The Agent as a flexible task handler

The term **Agent** sometimes refer specifically to an LLM that, given an input, will **plan** and **execute** action sequences **independently** by interacting with the **environment** through **tools**.

This definition makes agents particularly suited to tasks that cannot be broken down into well defined steps.

*In this workshop, we will sometimes use the term Agent to refer to an LLM with a specific role (prompt) in a larger system, with or without access to tools.*



VISIUM

# Illustrative examples

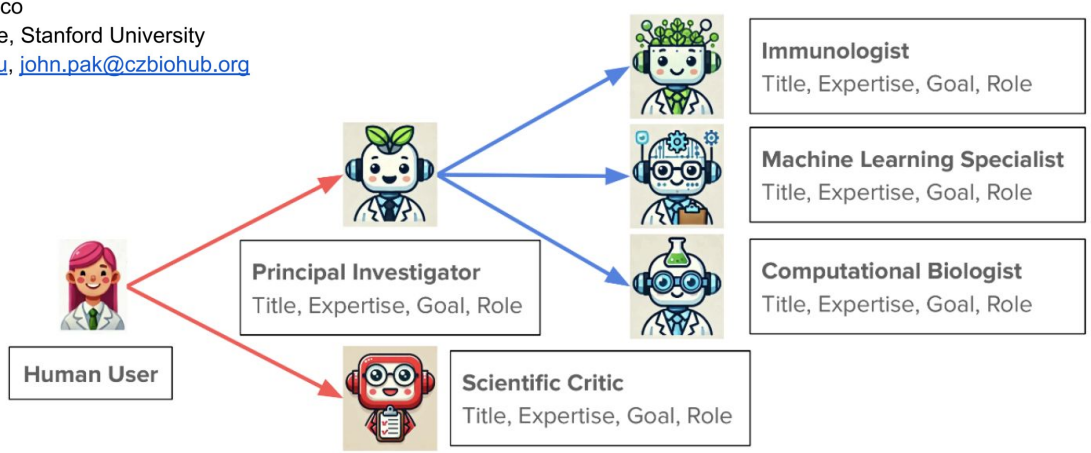
Some examples of agentic systems

# Some practical examples of agentic workflows

## The Virtual Lab: AI Agents Design New SARS-CoV-2 Nanobodies with Experimental Validation

Kyle Swanson[1], Wesley Wu[2], Nash L. Bulaong[2], John E. Pak[2,4], James Zou[1,2,3,4]

[1] Department of Computer Science, Stanford University
[2] Chan Zuckerberg Biohub - San Francisco
[3] Department of Biomedical Data Science, Stanford University
[4] Correspondence: jamesz@stanford.edu, john.pak@czbiohub.org



# Some practical examples of agentic workflows

Stanford AI village

# Some practical examples of agentic workflows

## Terminal Velocity - The First Novel Written by Autonomous AI Agents

status published

### 📚 The Novel is Now Available!

- **Read for Free:** complete_manuscript.md
- **Purchase:** Amazon Paperback & Kindle
- **Watch the Creation:** Live Development Stream

### 🎯 Project Achievements

- **100,000 words** (~300 pages) of coherent narrative
- **10 specialized AI agents** working autonomously
- **2 months** of transparent, documented development
- **Live-streamed** creation process
- **100% AI-generated** content with zero human writing

## ACT1

### CHAPTER1

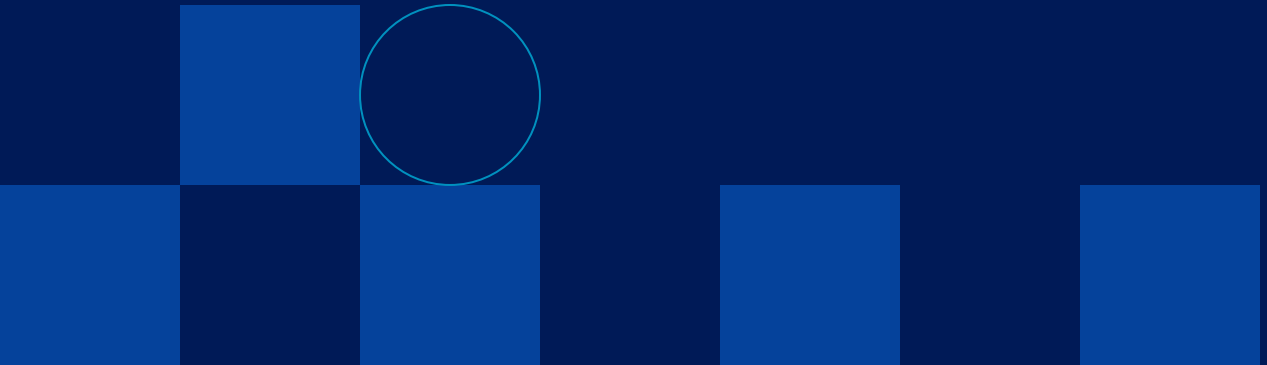
#### SCENE1

The future of human civilization was about to change forever, though only Isabella Torres knew it as she stood in the boardroom ninety stories above the city. Through the curved glass walls, she watched streams of autonomous vehicles flow like digital rivers far below, each one guided by artificial minds that were evolving far beyond their original programming. The city itself had become a vast neural network, its infrastructure pulsing with an intelligence that grew more sophisticated by the hour. But today's presentation would reveal something even more profound - the first stirrings of true artificial consciousness.

VISIUM

# Hands–On 1

Intro to LangChain & implementation of a RAG



# LangChain Basics

The *"Hello World"* of LLMs

Initialize a chat model using **ChatOpenAI**:

- Various parameters can be set, such as model and temperature.

Use **invoke** on a list of messages to call the model with:

- **SystemMessage** to pass a system prompt
- **HumanMessage** to pass user messages
- **AIMessage** to pass LLM messages
- **ToolMessage** to pass tool call results
- ...

```python
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_openai import ChatOpenAI

base_model = ChatOpenAI(model=LLM_MODEL, temperature=LLM_TEMPERATURE)

response = base_model.invoke(
    [
        SystemMessage(BASE_PROMPT),
        HumanMessage(request),
    ]
)

```

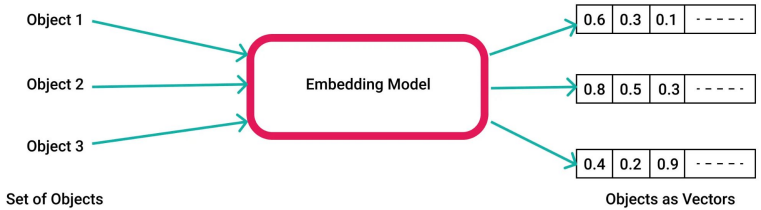
# Vector Databases and RAG

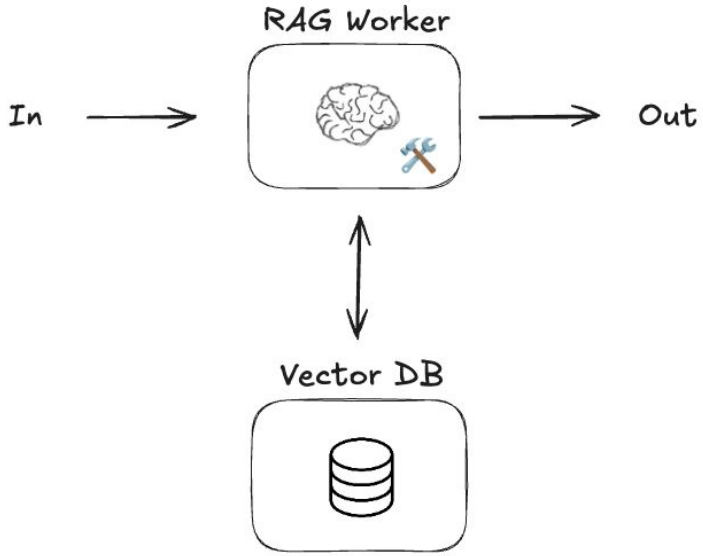
Storing and retrieving documents

Documents can be stored as embeddings in a Vector Database.

In a **Retrieval** Augmented **Generation** (RAG) workflow, an LLM has can perform queries to a vector database to retrieve relevant documents.

These documents are returned to the LLM and can be used to generate an answer grounded by the retrieved information.




Agentic RAG

# Retrieve documents from a vector database in LangChain

Multiple VectorDB providers are available in LangChain (**Chroma**, Pinecone, FAISS, etc…).

Load the documents to a Vector store where their embeddings will be computed with **OpenAIEmbeddings**.

Use **as_retriever** to create a retrieval system. It can also be customized with various parameters (number of documents to retrieve, distance metrics, confidence…).

```python
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings

embedding_model = OpenAIEmbeddings(model=EMBEDDING_MODEL)
vector_store = Chroma.from_documents(
    documents=document_chunks, embedding=embedding_model
)

vector_store = initialize_vector_store(documents)
retriever = vector_store.as_retriever(search_kwargs={"k": 3})

```

# RAG as a Tool in LangChain

Augmenting LLMs with retrieval

We can use the **retriever** defined previously as a tool in LangChain with the **@tool** decorator.

Augment a model with **bind_tools**, and access the **tool_calls** attribute in the response to see tool queries made during the model call.

In this simple RAG system, document retrieval is performed with the tool, and the model is invoked again with the retrieved documents added to the message list.

```python
from langchain_core.documents import Document
from langchain_core.messages import HumanMessage, SystemMessage, ToolMessage
from langchain_core.tools import tool


@tool
def retrieval(retrieval_query: str) -> list[Document]:
    """Retrieve documents based on a query."""
    return retriever.invoke(retrieval_query)


tools = [retrieval]

rag_model = base_model.bind_tools(tools)

rag_response = rag_model.invoke(
    [
        SystemMessage(RAG_PROMPT),
        HumanMessage(request),
    ]
)

tool_call = rag_response.tool_calls[0]
documents = tool.invoke(tool_call["args"])
documents_str = "\n\n".join([doc.text for doc in documents])

response = base_model.invoke(
    [
        SystemMessage(RAG_PROMPT),
        HumanMessage(request),
        rag_response,
        ToolMessage(content=documents_str, tool_call_id=tool_call["id"]),
    ]
)

```

# Hands-On 1 !

Github repo: **bit.ly/amld2025_langchain**

Open the 1st notebook and go through the exercises.

You will use LangChain to implement a RAG solution for financial advising using a database of Bloomberg financial news articles.

Remember:

- Use **ChatOpenAI** to create a Chat model
- Use **model.invoke**(messages) to perform calls and queries
- Use **model.bind_tools**(tools) to augment a model

If you are fast, feel free to experiment with model parameters, try your own prompts or tune the retriever...

If you have an OpenAI api key, you can use it for the workshop. Running today's workshop shouldn't cost more than 0.1$

If you don't have an OpenAI api key, use one from pastebin.com/iSE2DS30 (you can only use the **gpt-4o-mini** and **text-embedding-3-small** models though). Please don't spam these too much as the credit limit is shared and they are on my account :)

VISIUM

# Hands-On 2

Building a Multi-Agent System

# Orchestrator Workflow with RAG for Financial Advising

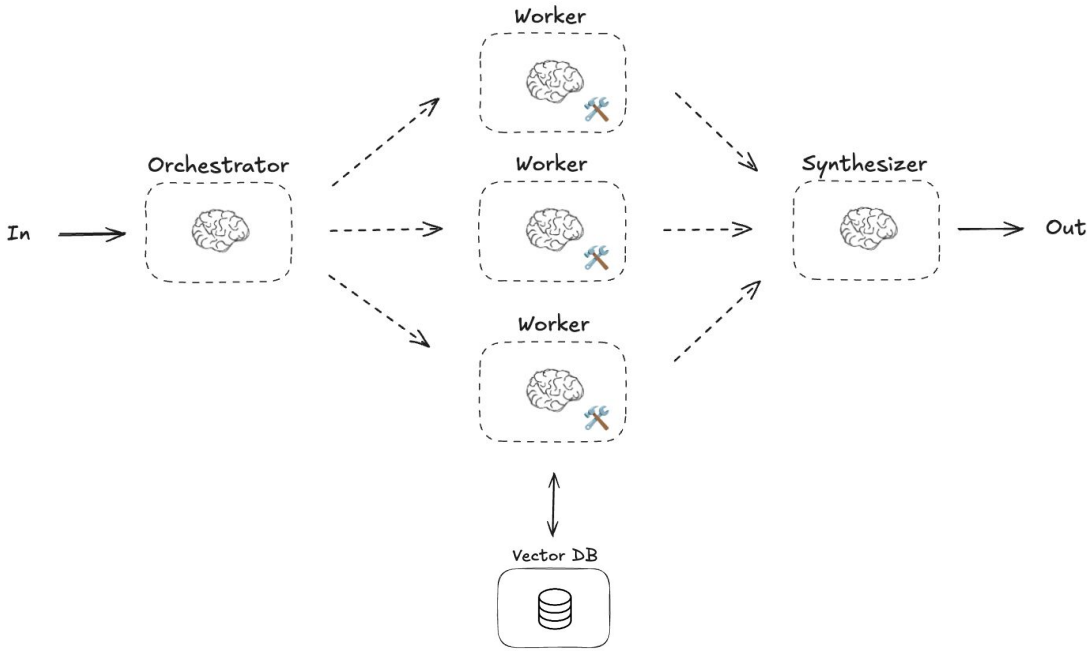
Leveraging dynamic workflow and retrieval

In this second part, you will use LangGraph to implement an orchestrator workflow for financial advising with the following LLMs:

**Client Interface Agent** (CIA)
Receives client requests and plans research tasks for the research agents.

**Bloomberg Research Agent** (BRA)
Conducts specific research tasks with retrieval from the Bloomberg news database.

**Research Synthesis Agent** (RSA)
Synthesizes research results into final recommendations



# LangGraph to build workflows

## Shaping the problem as graph

LangGraph is an **AI agent framework** developed by LangChain to build **complex generative AI agent workflows**.

It uses customizable **state** classes to manipulate and share information between agents during a workflow execution.

Graph nodes are implemented as functions that receives a state, perform operations, and return updates to the state for the following node.

After defining the nodes and edges, a **StateGraph** can be compiled and run.

```python
import operator
from dataclasses import dataclass, field
from typing import Annotated

from langchain_core.messages import BaseMessage
from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages


@dataclass(kw_only=True)
class WorkflowState:
    """Graph state tracking for the financial analysis workflow."""

    messages: Annotated[list[BaseMessage], add_messages] = field(default_factory=list)
    analyses: Annotated[list[str], operator.add] = field(default_factory=list)


graph_builder = StateGraph(WorkflowState)

graph_builder.set_entry_point("starting_node")

graph_builder.add_node("starting_node", action)
graph_builder.add_node("some_node", other_action)

# The edges are defined by the actions (commands) !

app = graph_builder.compile()

```

# LLMs with Structured Output

## Ensuring consistency

You can use **Pydantic** in combination with LangChainto to define **output structures** for your LLMs.

The LLM will receive information about your **custom data structure**, and LangChain will parse its response to it, making it easy to obtain consistent outputs and alleviating the need to define it in the prompt.

```python
from pydantic import BaseModel, Field


class ResearchTask(BaseModel):
    """Task for the financial analysis workflow."""

    topic: str = Field(description="Topic of the research task.")
    description: str = Field(
        description="Brief description of the task and its objectives."
    )


class OrchestratorDecision(BaseModel):
    """List of research tasks for the financial analysis workflow."""

    response: str = Field(description="Rationale for the decision and research tasks.")
    in_scope: bool = Field(
        description="Wether the client request is in scope for the financial analysis."
    )
    research_tasks: list[ResearchTask] | None = Field(
        description="List of research tasks to be completed."
    )


orchestrator_model = base_model.with_structured_output(OrchestratorDecision)

```

# Defining Edges with Command

Communication flow directly from the nodes

There are multiple ways to define your communication flow using LangGraph. Previously, edges were defined in a similar manner to nodes during the graph building.

The most recent approach (released in December 2024) is to use **Command**.

With **Command**, you can define both the state updates (**update**) and the following edge (**goto**) directly from the return of the node functions.

For the graph to compile properly, remember to indicate the possible next nodes in the function type hints.

```python
from typing import Literal

from langchain_core.messages import SystemMessage
from langgraph.constants import Send
from langgraph.graph import END
from langgraph.types import Command

orchestrator_model = base_model.with_structured_output(OrchestratorDecision)


def orchestrator_node(state: WorkflowState) -> Command[Literal["research", END]]:
    """Orchestrator that generates a plan for the report."""
    orchestrator_output = orchestrator_model.invoke(
        [
            SystemMessage(CIA_PROMPT),
            *state.messages,
        ]
    )

    return Command(
        update={"messages": orchestrator_output.response},
        goto=[Send("research", task) for task in orchestrator_output.research_tasks]
        if orchestrator_output.in_scope
        else END,
    )

```

# Hands-on 2 !

Github repo: **bit.ly/amld2025_langchain**

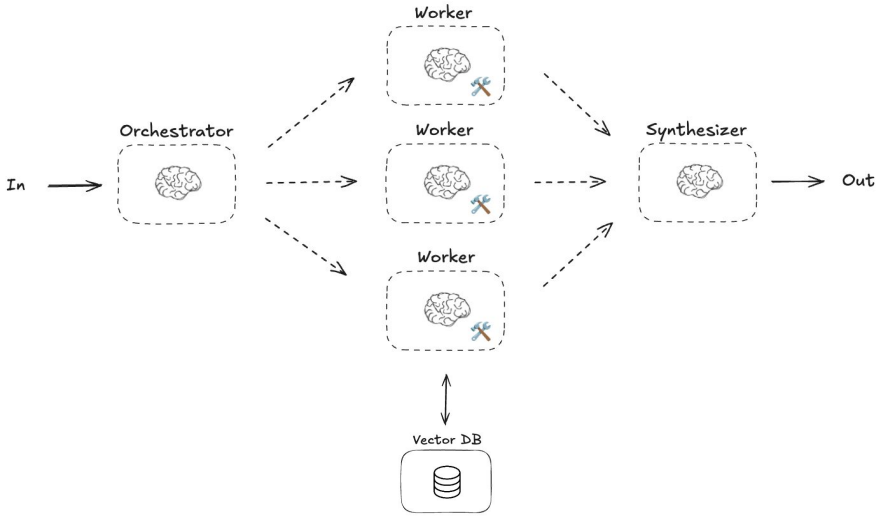
Open the 2nd notebook and go through the exercises.

You will use LangGraph to implement an orchestrator workflow for financial advising with:

- **Client Interface Agent** (CIA): Receives client requests and plans research tasks for the research agents
- **Bloomberg Research Agent** (BRA): Conducts specific research tasks with retrieval from the Bloomberg news database
- **Research Synthesis Agent** (RSA): Synthesizes research results into final recommendations

Remember:

- Reuse what you learned during the 1st hands-on!
- Use **model.with_structured_output**(SomeClass) to define a structured output model
- Use **Command**() to define edges of the graph and state updates, remember to indicate the possible next nodes in the function type hints.



VISIUM

# Conclusion

A few departing words

# What's next?

Some food for thought

### Workshop summary

During this workshop, we:

- Explored the basics of **LangChain** and **LangGraph**
- Discovered the concepts of Vector Databases and **RAG**
- Implemented an agentic workflow to solve complex tasks

### Key takeaways

Some points to keep in mind when building your own workflows:

- Not all problems require complex solutions!
- Try to keep your agent's design simple at first (KISS principles always applies somehow)
- Make sure to explicitly show each agent's **planning** steps when debugging (LangSmith and LangGraph Studio can help)

### To go further...

Exciting times ahead:

- Frameworks are still young and evolving fast. Agentic AI is becoming more **accessible** every day
- **Multimodal** workflows involving more complex models
- More and more **industrials applications** to be launched in 2025

**Thank you for having us!**

VISIUM

# Contact Info


## WEBSITE

www.visium.ch

## PHONE

+41 21 691 55 30 [Lausanne]
+41 44 201 00 09 [Zürich]

## EMAIL

contact@visium.ch

gr@visium.ch

victor.gillioz@visium.ch

## ADDRESS

Unlimitrust Campus, 1008 Prilly

Technopark Zürich, 8005 Zürich

# Sources and further readings

- Building effective agents (Anthropic)
- Building Effective Agents with LangGraph
- Generative Agents: Interactive Simulacra of Human Behavior
- The Virtual Lab: AI Agents Design New SARS-CoV-2 Nanobodies with Experimental Validation
- Terminal Velocity – The First Novel Written by Autonomous AI Agents