



An Introduction to AI Agents and Building Multi-Agent Systems with LangGraph

Lauzhack 2025 – Alejandro Bonell, Lucas Eckes, Younes Moussaif – 04.06.2025

Table of contents

01 **Introduction**

Why should we care about agentic AI?

02 **Concepts**

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

03 **Illustrative examples**

Some examples of agentic systems

04 **LangGraph**

Why LangGraph is the open-source
reference for building multi-agent systems?

05 **Financial Analyst Multi-Agent System**

Live coding session of a financial
multi-agent system using Yahoo Finance
and LangGraph

06 **Conclusion**

A few departing words



INTRODUCTION

Why should we care about agentic AI?



Going beyond LLMs

From Understanding to Action

LLMs have revolutionized AI with their language capabilities, but they're just the foundation for what comes next.



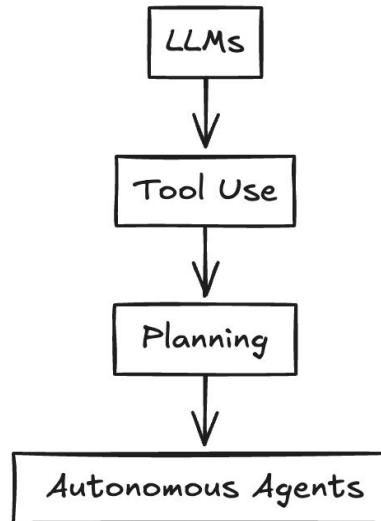
Tool Integration: Connecting LLMs to external tools, APIs, and resources to interact with the digital world



Workflows: Multi-step processes with conversation history, contextual decision-making, and interactions that bridge understanding and execution



From Text to Action: Moving beyond generating content to executing tasks and making real-world decisions



Now that language models can understand and reason effectively, the next frontier is empowering them to take meaningful action.

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

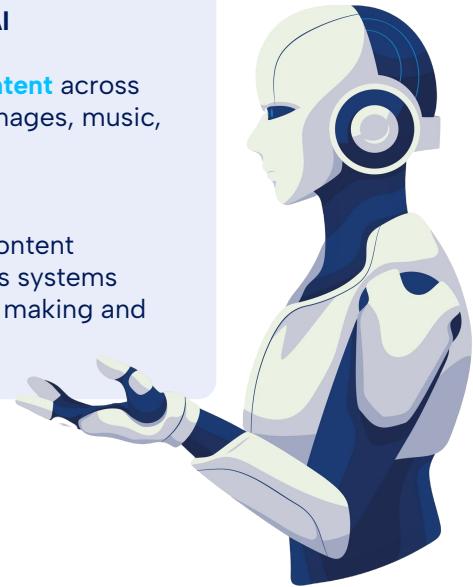


Task Execution: The ability to complete complex workflows and objectives

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.





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
- ...

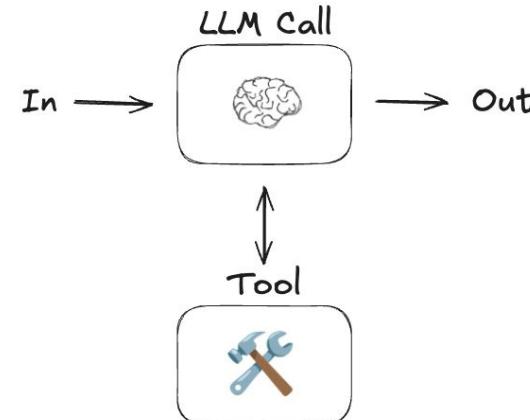
By informing the LLM about available tools and how they work, it can decide independently when to use them.

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

Standard LLM



Augmented LLM



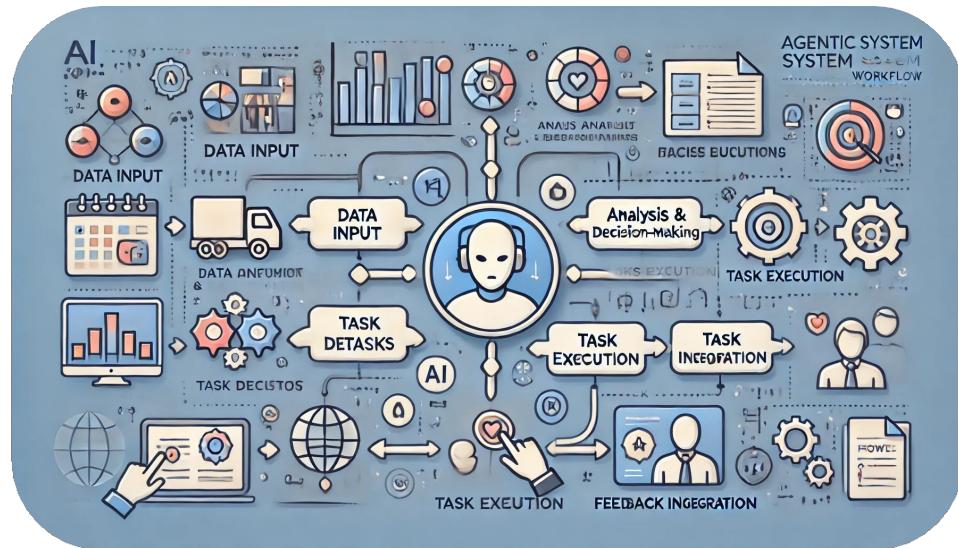
Workflows

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

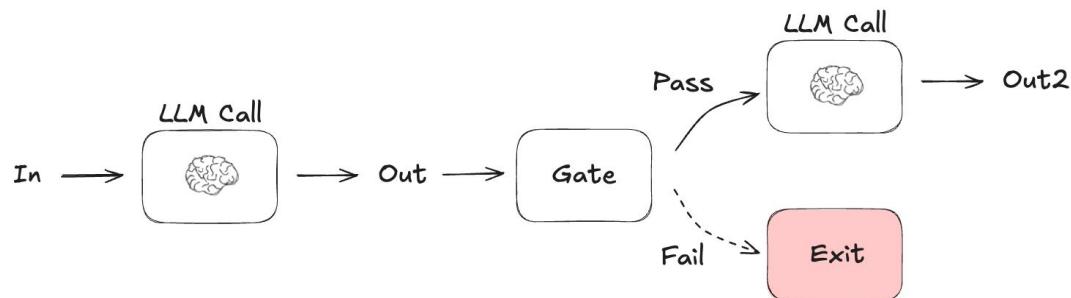
Split a task into small steps

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.



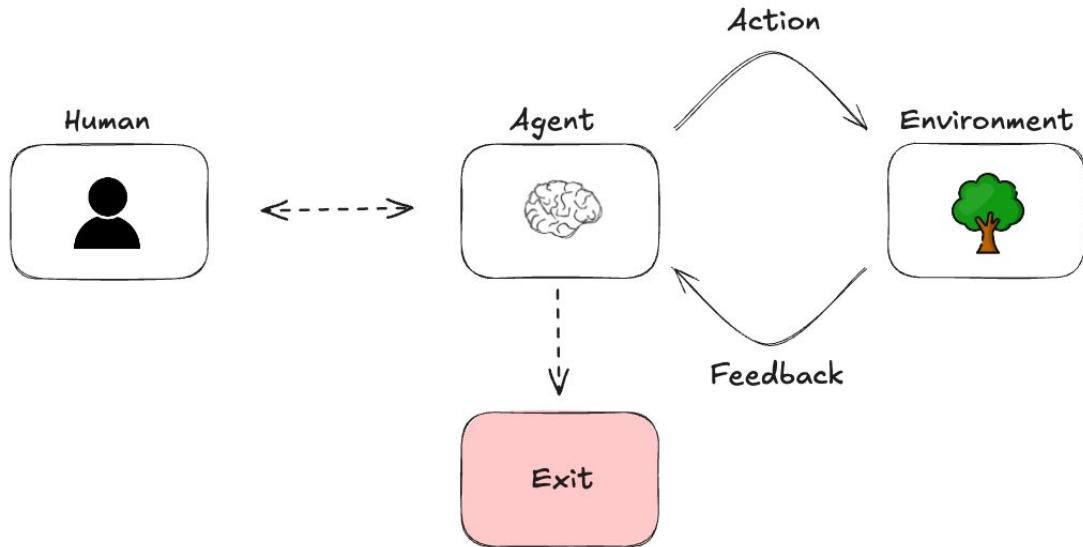
So what's an Agent?

The Agent as a flexible task handler

The term **Agent** sometimes refers 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.



Example: Routing

LLM for routing

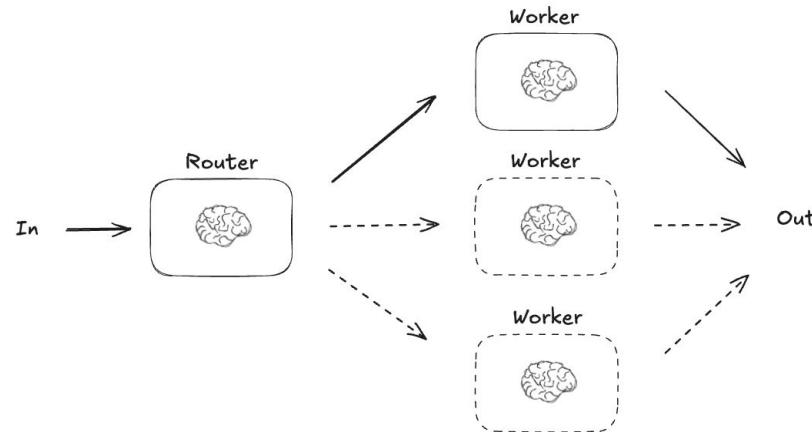
Routing offers more dynamic workflows by having an LLM decide which specialized LLM to call for different requests.

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: Evaluator–Optimizer

LLMs in a loop

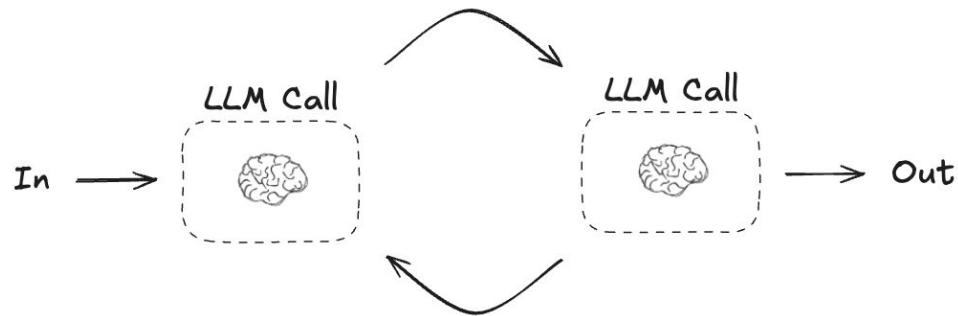
LLMs can also be chained **back and forth** to iterate on a solution until a satisfactory output is reached.

In this setting we have:

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

Practical example:

- Iterations for code development.





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

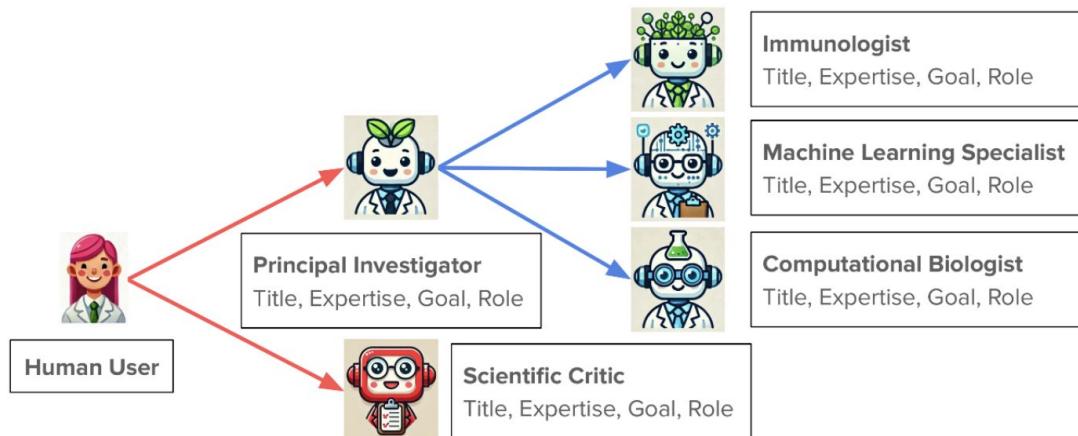
Kyle Swanson¹, Wesley Wu², Nash L. Bulaong², John E. Pak^{2,4}, James Zou^{1,2,3,4}

¹ Department of Computer Science, Stanford University

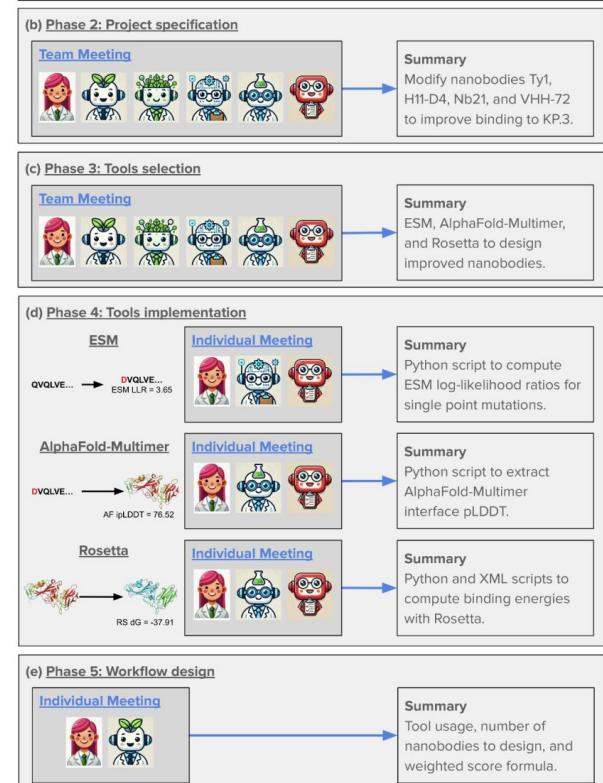
² Chan Zuckerberg Biohub - San Francisco

³ Department of Biomedical Data Science, Stanford University

⁴ Correspondence: jamesz@stanford.edu, john.pak@czbiohub.org



DARE TO BUILD YOUR DATA-DRIVEN FUTURE



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

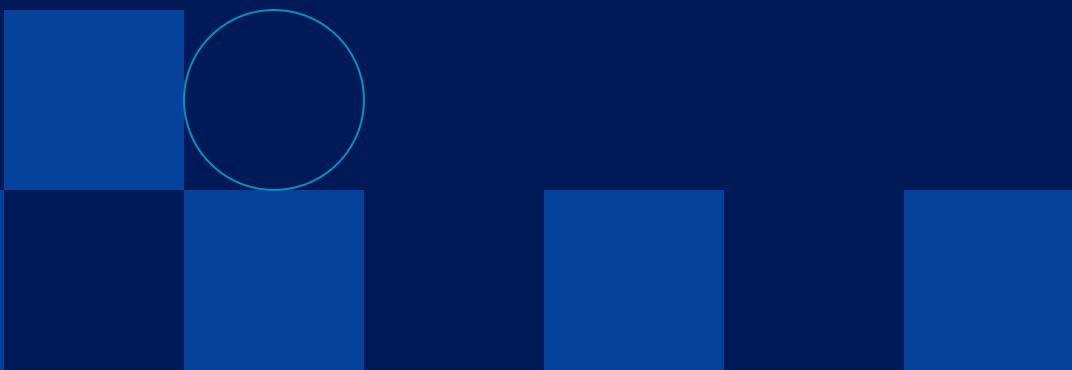
SCENE1

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.



LangGraph

Why LangGraph is the open-source reference for building
multi-agent systems?



Most popular frameworks for developing Multi-Agent Systems

	LangGraph	CrewAI	Autogen	OpenAI Agents (previously Swarm)
Ease of Use				
Strengths	Graph and stateful logic, lot of control over agents	Very simple to setup and large community and documentation	Middle ground between CrewAI and LangGraph. Dynamic collaboration	Clean and easy structure
Weaknesses	Documentation and learning curve steep (Graph, LangChain)	Not robust for complex and custom multi-agent systems	Works well only with the latest LLM models	Still experimental. Not entirely open-source
Summary	Structured workflow and control between agents	Quick multi-agents setup	Autonomous agents and Microsoft-backed tools	Lightweight and easy to use with few abstractions. Production-ready



What is LangGraph



An open-source framework built on top of LangChain, designed to support the development of sophisticated agent systems.



Its API is inspired by NetworkX and models agent interactions as a graph



Helps move beyond simple agents, and create complex agentic systems. Ideal for dynamic and iterative AI workflows.

Why use LangGraph?

Multi-agent systems – Ideal for building complex systems involving multiple agents
Dynamic and interactive workflows – Features like looping, branching, and revisiting steps



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.



```
1 import operator
2 from dataclasses import dataclass, field
3 from typing import Annotated
4
5 from langchain_core.messages import BaseMessage
6 from langgraph.graph import StateGraph
7 from langgraph.graph.message import add_messages
8
9
10 @dataclass(kw_only=True)
11 class WorkflowState:
12     """Graph state tracking for the financial analysis workflow."""
13
14     messages: Annotated[list[BaseMessage], add_messages] = field(default_factory=list)
15     analyses: Annotated[list[str], operator.add] = field(default_factory=list)
16
17
18 graph_builder = StateGraph(WorkflowState)
19
20 graph_builder.set_entry_point("starting_node")
21
22 graph_builder.add_node("starting_node", action)
23 graph_builder.add_node("some_node", other_action)
24
25 # The edges are defined by the actions (commands) !
26
27 app = graph_builder.compile()
28
```

LLMs with Structured Output



Ensuring consistency

You can use **Pydantic** in combination with LangChain 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.

```
● ● ●
1  from pydantic import BaseModel, Field
2
3
4  class ResearchTask(BaseModel):
5      """Task for the financial analysis workflow."""
6
7      topic: str = Field(description="Topic of the research task.")
8      description: str = Field(
9          description="Brief description of the task and its objectives."
10     )
11
12
13 class OrchestratorDecision(BaseModel):
14     """List of research tasks for the financial analysis workflow."""
15
16     response: str = Field(description="Rationale for the decision and research tasks.")
17     in_scope: bool = Field(
18         description="Whether the client request is in scope for the financial analysis."
19     )
20     research_tasks: list[ResearchTask] | None = Field(
21         description="List of research tasks to be completed."
22     )
23
24
25 orchestrator_model = base_model.with_structured_output(OrchestratorDecision)
26
```

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.

```
1  from typing import Literal
2
3  from langchain_core.messages import SystemMessage
4  from langgraph.constants import Send
5  from langgraph.graph import END
6  from langgraph.types import Command
7
8  orchestrator_model = base_model.with_structured_output(OrchestratorDecision)
9
10
11 def orchestrator_node(state: WorkflowState) -> Command[Literal["research", END]]:
12     """Orchestrator that generates a plan for the report."""
13     orchestrator_output = orchestrator_model.invoke(
14         [
15             SystemMessage(CIA_PROMPT),
16             *state.messages,
17         ]
18     )
19
20     return Command(
21         update={"messages": orchestrator_output.response},
22         goto=[Send("research", task) for task in orchestrator_output.research_tasks]
23         if orchestrator_output.in_scope
24         else END,
25     )
26
```



Financial analyst Multi-Agent Systems

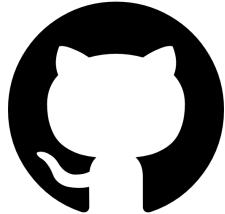
Live coding session of a financial multi-agent system using Yahoo Finance and LangGraph



Hands-On

Intro to LangChain & an implementation of a simple multi agent system

Hands on



Github repo: bit.ly/lauzhack_agentic_2025

OpenAI API Keys: <https://pastebin.com/qiZZshyy>

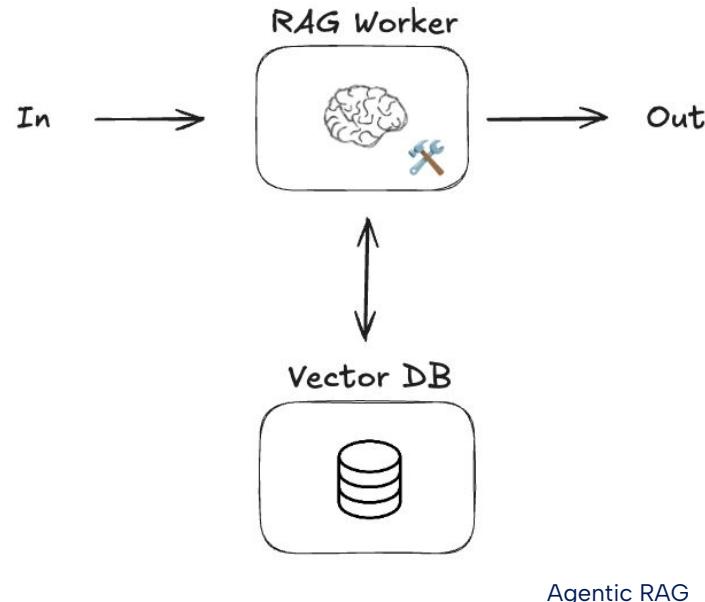
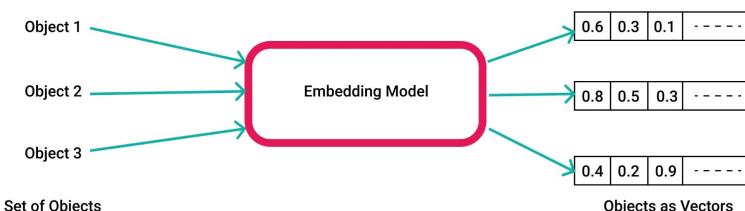
Vector Databases and RAG

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.



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...).



```
1  from langchain_chroma import Chroma
2  from langchain_openai import OpenAIEmbeddings
3
4  embedding_model = OpenAIEmbeddings(model=EMBEDDING_MODEL)
5  vector_store = Chroma.from_documents(
6      documents=document_chunks, embedding=embedding_model
7  )
8
9  vector_store = initialize_vector_store(documents)
10 retriever = vector_store.as_retriever(search_kwargs={"k": 3})
11
```

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.

```
● ○ ●
1  from langchain_core.documents import Document
2  from langchain_core.messages import HumanMessage, SystemMessage, ToolMessage
3  from langchain_core.tools import tool
4
5
6  @tool
7  def retrieval(retrieval_query: str) -> list[Document]:
8      """Retrieve documents based on a query."""
9      return retriever.invoke(retrieval_query)
10
11
12 tools = [retrieval]
13
14 rag_model = base_model.bind_tools(tools)
15
16 rag_response = rag_model.invoke(
17     [
18         SystemMessage(RAG_PROMPT),
19         HumanMessage(request),
20     ]
21 )
22
23 tool_call = rag_response.tool_calls[0]
24 documents = tool.invoke(tool_call["args"])
25 documents_str = "\n\n".join([doc.text for doc in documents])
26
27 response = base_model.invoke(
28     [
29         SystemMessage(RAG_PROMPT),
30         HumanMessage(request),
31         rag_response,
32         ToolMessage(content=documents_str, tool_call_id=tool_call["id"]),
33     ]
34 )
35
```

Hands-On 1 !

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...



Github repo: bit.ly/amld2025_langchain

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 :)



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)

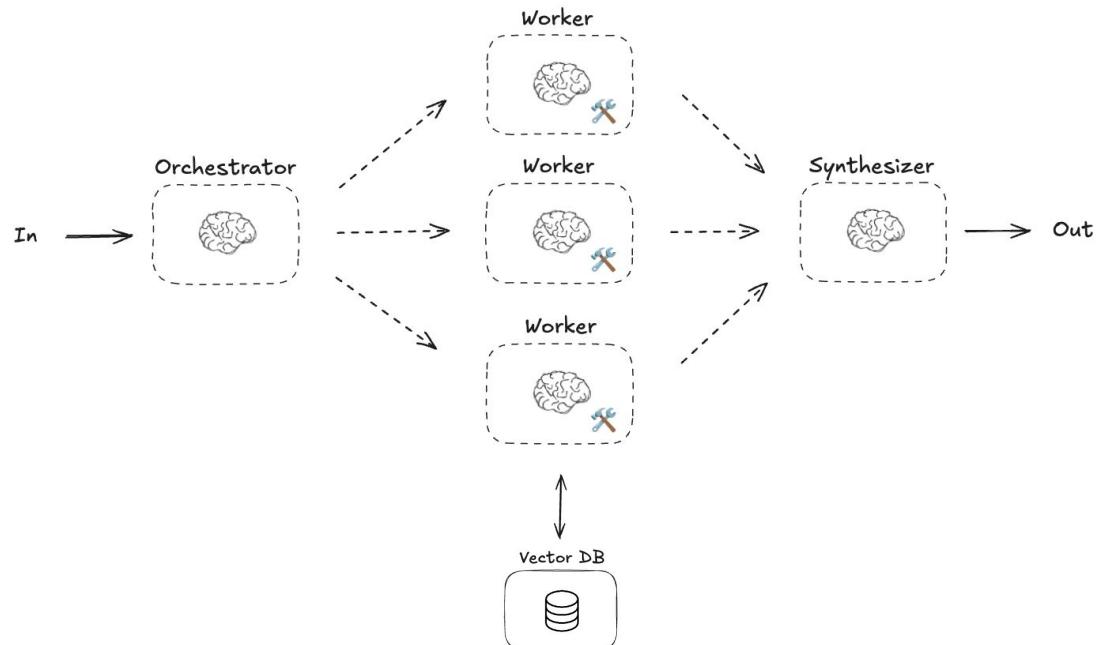
Receives client requests and plans research tasks for the research agents.

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





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**
- Implemented an agentic workflow to solve complex tasks

Key takeaways

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

- You can augment your LLM with tools
- You can easily create LLMs projects with LangChain and LangGraph

To go further...

Exciting times ahead:

- Frameworks are still young and evolving fast. Agentic AI is becoming more **accessible** every day
- RAGs
- Feedback loops, more complex workflows

Thank you for having us!

Follow these simple steps to kickstart your career at Visium!

Our recruitment process gives us the possibility to know you are the right fit



Submit your
application online



Meet with our Talent
Acquisition team to
see if we're a match



Show us your
Competency skill set



Work on a use case to
demonstrate your expertise



Celebrate — you're
now a Visiumee!

Apply today and let's build a data-drive future together!

<https://www.visium.ch/join-us/>





Contact Info

WEBSITE

www.visium.com

PHONE

+41 21 691 55 30 [Lausanne]

+41 44 201 00 09 [Zürich]

EMAIL

contact@visium.com

alejandro.bonell@visium.com

lucas.eckes@visium.com

younes.moussaif@visium.com

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](#)