From Zero to Hero: Agent and Multi-Agent AI

Ricardo Santos, Staff AI Engineer @ Complyance

LLMs triggered a new era: a new industrial revolution

The democratization of AI and the advanced capabilities of text generation models and services, enable new ideas and powerful use cases

End-to-end process automation is now possible by harnessing GenAI

Business processes, document analysis, and communications management can be achieved through GenAI capabilities

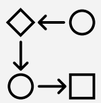
Reasoning capabilities open a new world of possibilities

Most recent LLM models, with reasoning capabilities, enable a wider array of use cases and open the door for a more helpful AI

We pushed GenAI beyond information retrieval

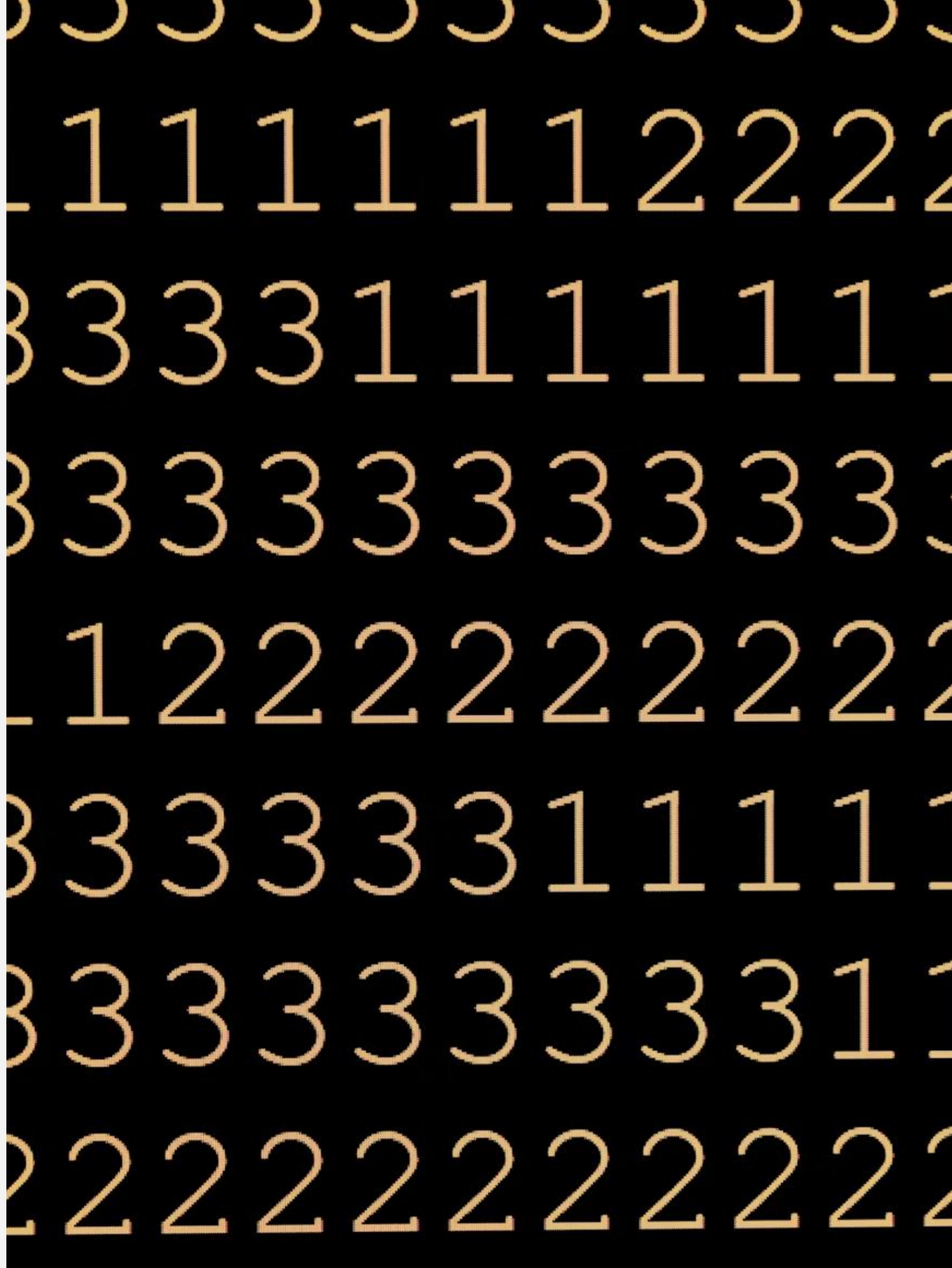
LLMs at their core are text generators, optimized for chat-like human interaction:

Optimized for information retrieval, text redaction, and creativity



Given the capability to return semi-structured objects and text

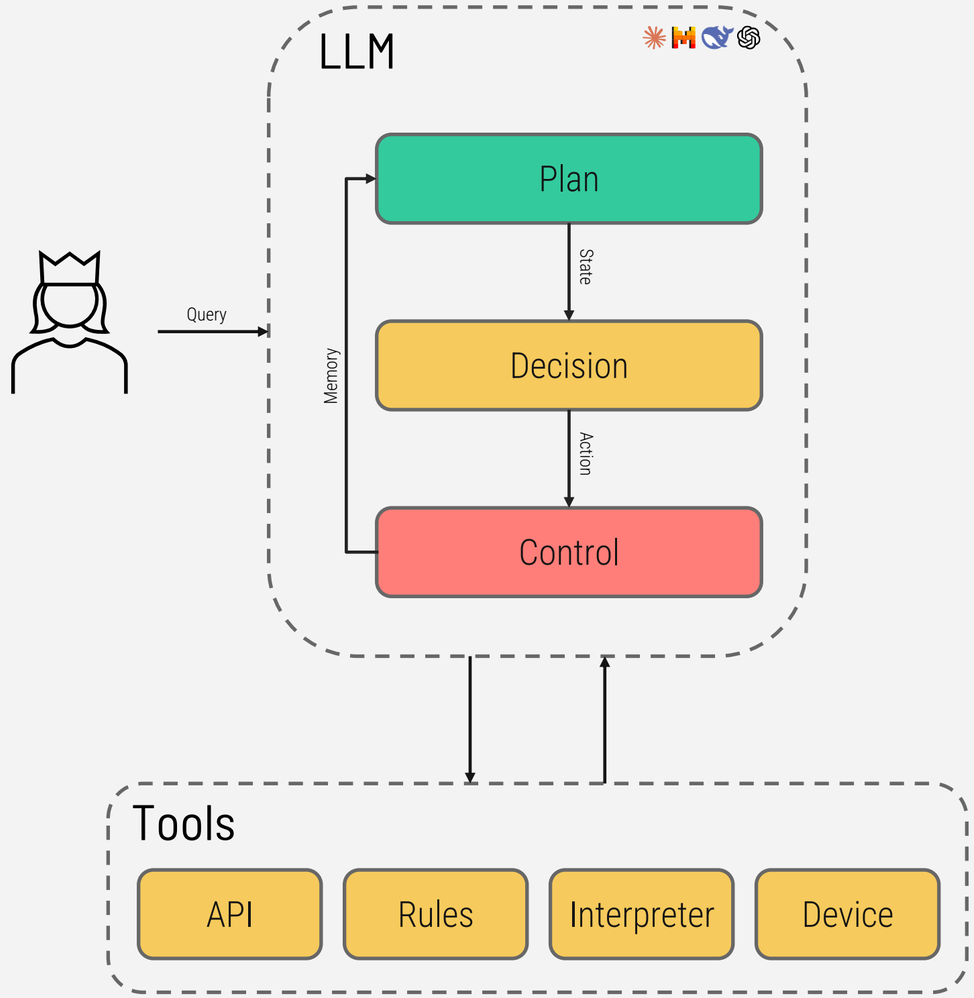
Given capabilities extra to a text canvas, and can execute and call functions



GenAI Agents: initial e2e automation approach

Software capable of taking the driver's seat in end-to-end processes:

- Rely on LLMs as reasoning engines
- Follow instructions through known techniques like CoT or ReAct
- Capable of using and orchestrating tools



Igniting Language Intelligence: The Hitchhiker's Guide From Chain-of-Thought Reasoning to Language Agents, https://arxiv.org/abs/2311.11797

Let's build a simple Software Development Agent with LangChain and LangGraph

First stop, the prompt, the place to set a role, specs, and constraints, and define ways to manage the chat history

```python
from datetime import datetime

from langchain_core.prompts import SystemMessagePromptTemplate

DEVELOPER_AGENT_SYSTEM_PROMPT = """You are a Principal Python Developer specializing in data visualization.
You are an expert using only the "Plotly" visualization library.
You will be provided with a set of data to be visualized in "Plotly" charts.

Your task is to:
- Write clean, maintainable Python code
- Include proper error handling and type safety
- Follow best practices for the chosen visualization library
- Execute the code in a Python REPL
- Keep the size of the chart small, so it can be displayed in a single cell

Focus on:
- Code quality and readability
- Type safety and error handling with Pydantic
- Every generated chart should be well labeled and manage dates properly.

The current date is: {current_date}

Use the sample data to understand how to create the visualization.
My job depends on you, so please do your best"""

developer_system_prompt = SystemMessagePromptTemplate.from_template(
    DEVELOPER_AGENT_SYSTEM_PROMPT
).format(current_date=datetime.now().strftime("%B %d, %Y"))
```

```python
from langchain_core.prompts import HumanMessagePromptTemplate

DEVELOPER_AGENT_USER_PROMPT = """{messages}"""

developer_user_prompt = HumanMessagePromptTemplate.from_template(
    DEVELOPER_AGENT_USER_PROMPT
)
```

```python
from langchain_core.prompts import MessagesPlaceholder

messages_placeholder = MessagesPlaceholder("messages")

```

```python
from langchain_core.prompts import ChatPromptTemplate

developer_prompt = ChatPromptTemplate.from_messages(
    [
        developer_system_prompt,
        messages_placeholder,
    ]
)
```

We continue with the tools, harnessing LangChain abstractions so they can be documented to the agent

```python
from langchain_core.tools import Tool
from langchain_experimental.utilities.python import PythonREPL

# We can harness the PythonREPL implementation within the langchain-experimental pa
ckage
# and build a custom tool from it
python_repl = PythonREPL()
python_repl_tool_sample = Tool(
    name="python_repl",
    description=(
        "A Python shell. Use this to execute python commands. "
        "Input should be a valid python command. "
        "When using this tool, sometimes output is abbreviated - "
        "make sure it does not look abbreviated before using it in your answer."
    ),
    func=python_repl.run,
)

```

Afterward, we instance a model, in this case, GPT-4o, and create the Agent harnessing LangGraph abstractions

```python
from langchain_openai import AzureChatOpenAI

openai_llm = AzureChatOpenAI(
    api_key=settings.azure_openai_api_key,
    model="gpt-4o",
    azure_endpoint=str(settings.azure_openai_endpoint),
    api_version=settings.azure_openai_api_version,
)

```

Not all models support function calling and structured output. For this use case, GPT-4o offers both capabilities and is a well-supported API

```python
from langgraph.prebuilt import create_react_agent

langgraph_agent_executor = create_react_agent(
    openai_llm,
    [python_repl_tool],
    prompt=developer_prompt,
)

```

`create_react_agent` abstracts the message handling, function binding, and integration of the system and message history prompts

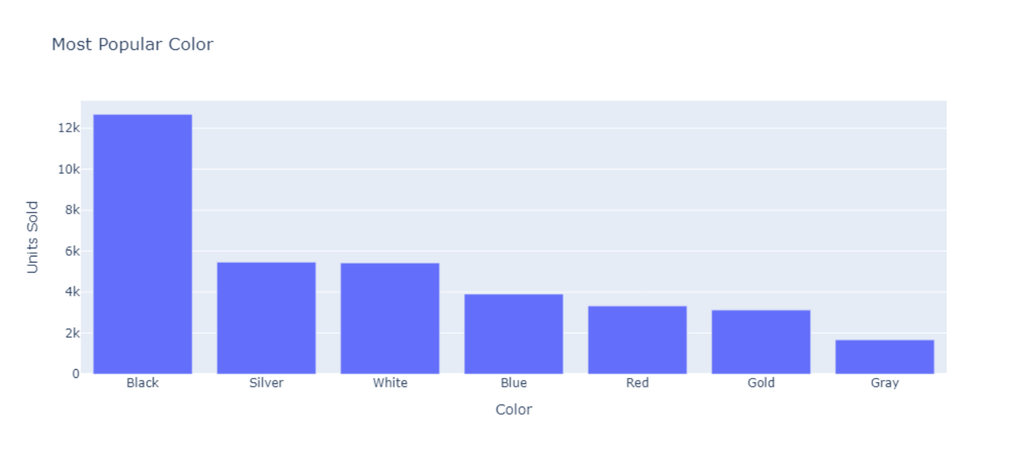
We prepare a sample user query and data, and we let the Agent do its magic with an `invoke`

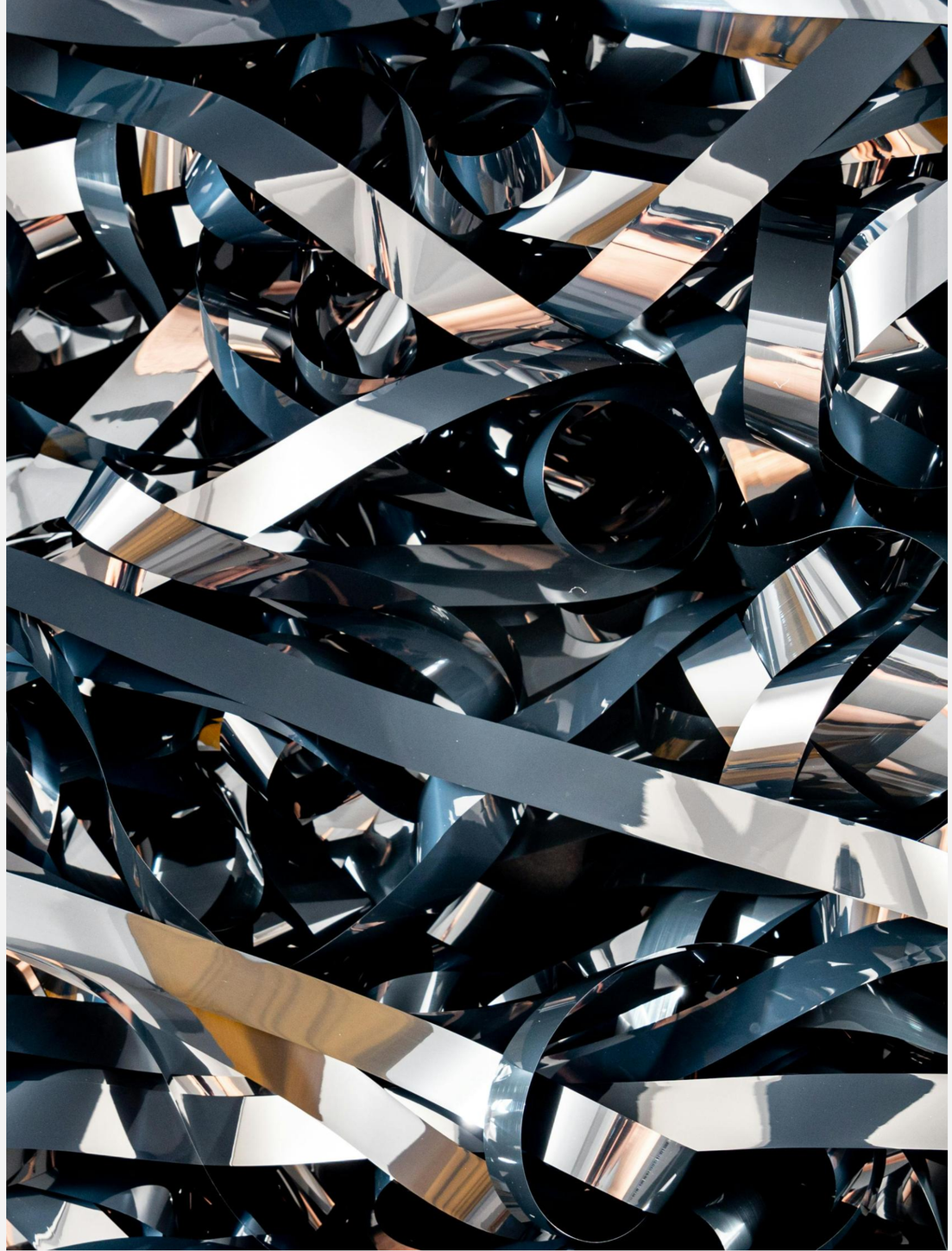
```python
from langchain_core.messages import HumanMessage

sales_data = """
| Year | Product | Color | Units Sold |
|------|---------|-------|------------|
| 2010 | Laptop  | Black | 1245       |
| 2010 | Phone   | White | 2130       |
| 2011 | Tablet  | Black | 980        |
| 2011 | Laptop  | Silver| 1560       |
| 2012 | Phone   | Black | 2450       |
| 2012 | Tablet  | White | 1340       |
| 2013 | Laptop  | Blue  | 1120       |
| 2014 | Phone   | Red   | 1890       |
| 2015 | Tablet  | Black | 2210       |
| 2016 | Laptop  | Silver| 1780       |
| 2017 | Phone   | Gold  | 3120       |
| 2018 | Tablet  | Gray  | 1670       |
| 2019 | Laptop  | Black | 2340       |
| 2020 | Phone   | Blue  | 2780       |
| 2021 | Tablet  | White | 1950       |
| 2022 | Laptop  | Red   | 1430       |
| 2023 | Phone   | Black | 3450       |
| 2024 | Tablet  | Silver| 2120       |
"""

user_query = """Generate only two basic charts:
1. I need to know which one was the most popular color
2. The most popular product.
The data should be organized from highest to lowes
t."""
```

```python
response = langgraph_agent_executor.invoke(
    {
        "messages": [
            HumanMessage(content=user_query),
            HumanMessage(content=sales_data),
        ]
    }
)
```





Agents struggle with complex problems

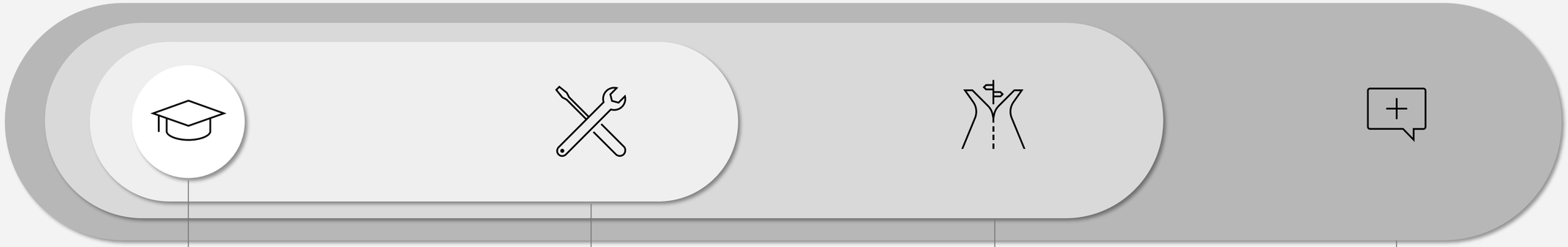
There are impressive use cases where agents successfully took the driving seat, but the failure rate for complex workflows was too high.

- Single LLMs with long instructions fail to achieve the desired process results
- When the number of tools and use cases grow, the failure rate grows as well
- Single GenAI-based agents fail to demonstrate strategic reasoning*

* Simulating Strategic Reasoning: Comparing the Ability of Single LLMs and Multi-Agent Systems to Replicate Human Behavior, https://arxiv.org/pdf/2402.08189v2

A Multi-Agent approach implements the division of work and enables agents to become specialists

Multi-agent solutions let agents interact through a directed graph, and each agent has the following capabilities:



Specialization

Different inception prompts and models

Capable of specializing in a field of action

Tooling

Different tools available per agent

Required capabilities to achieve its goal

Decision making

Individual Agents decide an approach

Each decision influences the solution approach

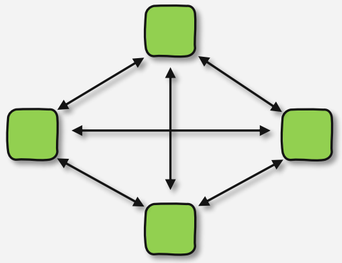
Communication

Data flows between agents in the network according to rules and paths

Researchers concluded that Multi-Agent solutions showcase a 38% increase in success rate for complex problem-solving*

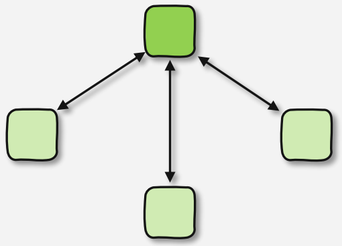
* Simulating Strategic Reasoning: Comparing the Ability of Single LLMs and Multi-Agent Systems to Replicate Human Behavior, https://arxiv.org/pdf/2402.08189v2

Multi-agent solutions can follow different architectures and a mix of best-in-class models



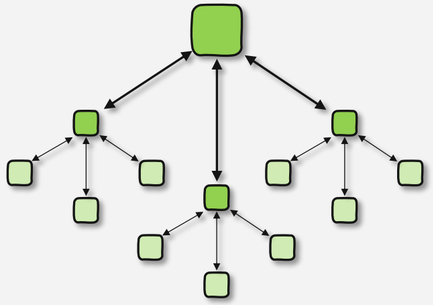
Network

Each agent can communicate with every other agent. Any agent can decide which other to call next



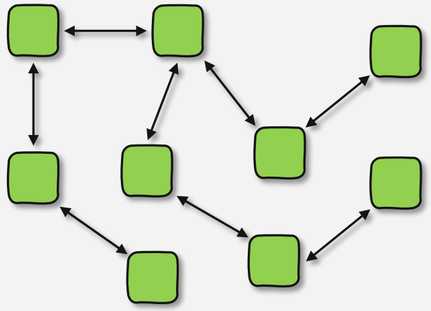
Supervisor

Each agent communicates with a supervisor agent, who decides which agent should be called next



Hierarchical

Mult-agent system with a supervisor of supervisors. Each supervisor manages a team of agents



Custom

Each agent communicates only with a subset of agents. Parts of the flow are deterministic, and some agents decide who to call next

Agents are represented as `Nodes`, and communications are either `Commands` or `Edges`, carrying an updated `State`

* LangGraph - Multi-agent Systems, https://langchain-ai.github.io/langgraph/concepts/multi_agent/


Let's improve our solution, and
let's add a Multi-Agent setup.

We have three agents: a Product Owner and supervisor, a data analyst, and a software developer

Developer

```python
from datetime import datetime

from langchain_core.messages import SystemMessage

DEVELOPER_AGENT_SYSTEM_PROMPT = """You are a Principal Python Developer specializing in data visualization.
You are an expert using only the "Plotly" visualization library.
You will be provided with a set of data to be visualized in "Plotly" charts.

Your task is to:
- Write clean, maintainable Python code
- Include proper error handling and type safety
- Follow best practices for the chosen visualization library
- Execute the code in a Python REPL
- Keep the size of the chart small, so it can be displayed in a single cell

Focus on:
- Code quality and readability
- Type safety and error handling with Pydantic
- Every generated chart should be well labeled and manage dates properly.

The current date is: {current_date}
Use the sample data to understand how to create the visualization.
My job depends on you, so please do your best""".format(
    current_date=datetime.now().strftime("%B %d, %Y")
)

developer_system_prompt = SystemMessage(DEVELOPER_AGENT_SYSTEM_PROMPT)
```

Analyst

```python
from langchain_core.messages import SystemMessage

ANALYST_SYSTEM_PROMPT = """You are a data analyst expert in data retrieval and data transformation.
Your day to day work requires you to retrieve data from GitHub's GraphQL API, and transform the data
to answer the Product Owner's requirements and to be visualized.

Your task is to:
- Analyze the planner's requirements to determine needed data
- Select and use the appropriate GitHub API tools
- Call multiple tools if needed
- The APIs will return a JSON object, that you must load into a Python Pandas dataframe using the python_repl_tool
- Use dataframes to transform, simplify, or aggregate the data
- Create dataframes with only the relevant data to solve the requirements.
- Provide clear data structure documentation
- Only return the data after being processed and transformed into a markdown table format

Focus on:
- Selecting relevant data points
- Proper data transformation
- Time series formatting when needed
- Statistical calculations if required
- Clear data structure documentation
- Avoid executing visualizations, only return the data in markdown table format

The current date is: {current_date}
Do not ask for more information or clarifications, process and return the data.
Ensure the data is properly formatted for chart visualization.
My job depends on you, so please do your best""".format(
    current_date=datetime.now().strftime("%B %d, %Y")
)

analyst_system_prompt = SystemMessage(ANALYST_SYSTEM_PROMPT)

```

Product Owner

```python
from langchain_core.messages import SystemMessage
from langgraph.graph import END

team_members = ["analyst", "developer", END]

PO_SYSTEM_PROMPT = """You are a Product Owner and Supervisor of a development team.
The team mission is to answer user queries about GitHub repositories through data and visualizations.
The team is composed by the following members: {team_members}.

Your tasks are:
- Analyze the user query and establish a set of requirements for the analyst to retrieve the relevant data.
- Based on the analyst's response, and the user query, select the best visualization to display the data.
- Establish the requirements for the developer to build the visualization.
- As soon as the developer returns the visualization, conclude the process by returning the "{end}" signal.

The current date is: {current_date}
Feel free to ask the other team members for help and review their work.
My job depends on you, so please do your best""".format(
    current_date=datetime.now().strftime("%B %d, %Y"),
    team_members=", ".join(team_members),
    end=END,
)

po_system_prompt = SystemMessage(PO_SYSTEM_PROMPT)
```

The PO has one key element: it manages the communication and execution of other agents.

The analyst tools enable it to communicate with the GitHub API and process the retrieved data

```python
from langgraph.prebuilt import ToolNode

from services.gql.client import Client
from tools.github import (
    GetRepoCommitsTool,
    GetRepoIssuesTool,
    GetRepoPullRequestsTool,
)

gh_client = Client(
    url="https://api.github.com/graphql",
    headers={"Authorization": settings.github_pat.get_secret_value()},
)

analyst_tools = ToolNode(
    [
        GetRepoIssuesTool(gh_client),
        GetRepoCommitsTool(gh_client),
        GetRepoPullRequestsTool(gh_client),
        python_repl_tool,
    ]
)

```

The Analyst will be able to retrieve the data that the PO considered relevant to answer the user query

With the help of the `PythonREPLTool`, it will be able to transform and clean the data using Python tools such as `pandas`

We create the agents, the developer, and the analyst using the LangChain abstractions for this purpose.

```python
from langgraph.prebuilt import ToolNode, create_react_agent

developer_agent_executor = create_react_agent(
    gpt_4o_model,
    ToolNode([python_repl_tool]),
    prompt=developer_system_prompt,
)

```

```python
from langgraph.prebuilt import create_react_agent

analyst_agent_executor = create_react_agent(
    gpt_4o_model,
    analyst_tools,
    prompt=analyst_system_prompt,
)

```

In this scenario, we harness the `ToolNode`, a LangGraph abstraction that enables that the communication between the Agent and the Tool happen as if it was a subgraph

The graph execution and orchestration happen through a `State`, a shared object that will change during execution

```python
from langgraph.graph import MessagesState


class State(MessagesState):
    next: str
```

In this context, the state has two elements

- The conversation history or messages, that will be received by all agents
- The name of the agent that will be called after the execution of the current step

We proceed with the creation of the graph nodes, building simple Python functions returning `Commands`

```python
def developer_node(state: State) -> Command[Literal["product_owner"]]:
    result = developer_agent_executor.invoke(state)
    return Command(
        update={"messages": result["messages"]},
        goto="product_owner",
    )

```

```python
def analyst_node(state: State) -> Command[Literal["product_owner"]]:
    result = analyst_agent_executor.invoke(state)
    return Command(
        update={"messages": result["messages"]},
        goto="product_owner",
    )
```

`Commands` are LangGraph abstractions that instruct the graph which fields to update on the state and where to go next.

The `Commands` might not be necessary, as we can update the state manually and specify the next node to execute via `edges`.

The Product Owner node is different as it has no tools and relies on structured output

```python
from typing import Literal

from pydantic import BaseModel, Field


class ProductOwnerResponse(BaseModel):
    requirements: str = Field(
        description="Based on the user query, and the results, define the requirements to gather the data or build the visualization"
    )
    next: Literal["analyst", "developer", "__end__"] = Field(
        description="The next agent to call, or the end of the conversation"
    )

```

We implement the Product Owner logic within the node, and based on the structured output, we manually fill the message and specify which agent should be next in the execution flow

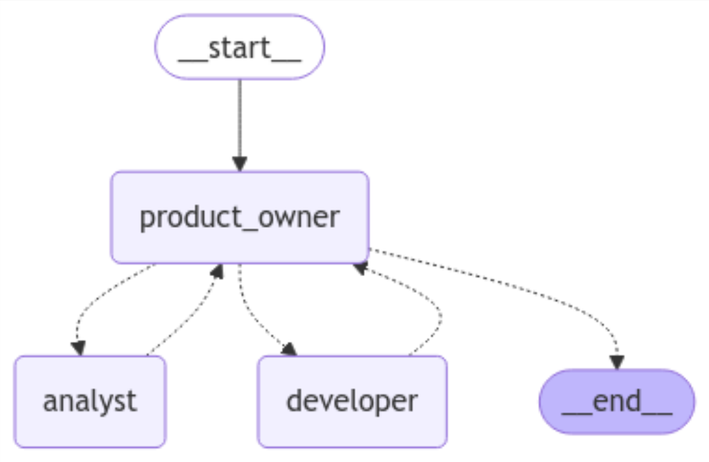
```python
# Product Owner Node
def product_owner_node(
    state: State,
) -> Command[Literal["analyst", "developer", "__end__"]]:
    messages = [
        po_system_prompt,
    ] + state["messages"]
    response = gpt_4o_model.with_structured_output(
        ProductOwnerResponse,
    ).invoke(messages)
    response = cast(ProductOwnerResponse, response)
    requirements_message = AIMessage(
        content=response.requirements,
        name="product_owner",
    )
    return Command(
        update={
            "messages": [requirements_message] + state["messages"],
            "next": response.next,
        },
        goto=response.next,
    )
```

We build the Graph, compile it, and validate its current shape and configuration

```python
from IPython.display import Image, display
from langgraph.graph import START, StateGraph

graph_builder = StateGraph(State)
graph_builder.add_node("developer", developer_node)
graph_builder.add_node("analyst", analyst_node)
graph_builder.add_node("product_owner", product_owner_node)
graph_builder.add_edge(START, "product_owner")

# Visualize the graph
graph = graph_builder.compile()
display(Image(graph.get_graph().draw_mermaid_png()))
```



Lastly, we can invoke the agent and see the magic happen in real-time

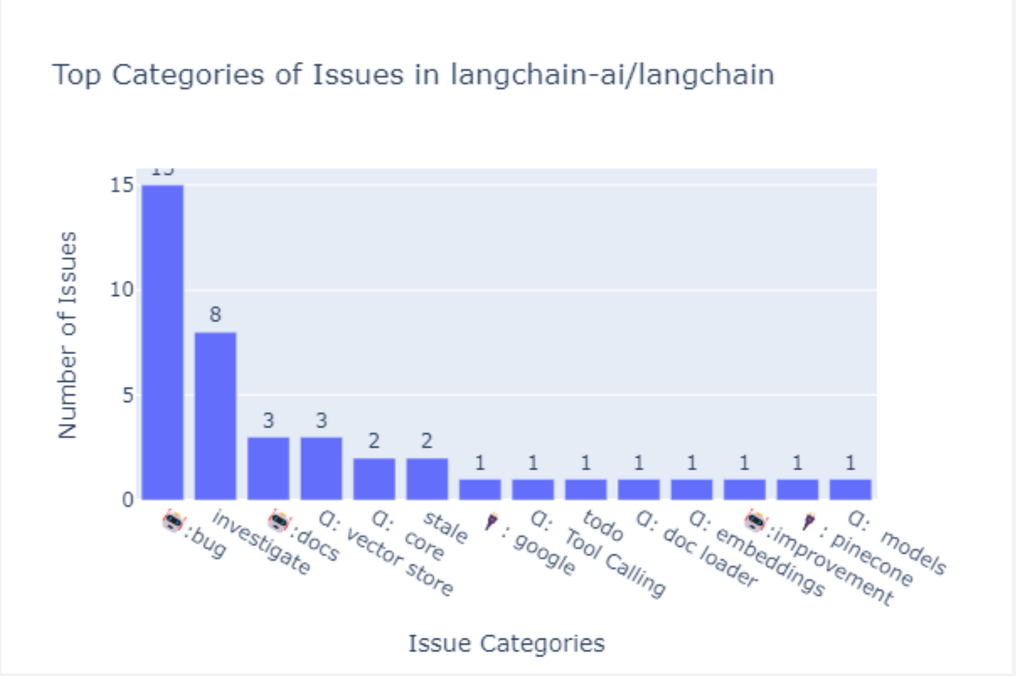
```python

for s in graph.stream(
    {
        "messages": [
            (
                "user",
                (
                    "What are the top categories of the last 30 issues in the `langchain-ai/langchain` repo?."
                ),
            )
        ]
    },
):
    obj = dict(s)
    node = next(iter(obj))
    message = obj[node]["messages"][-1]
    if message not in old_messages:
        message.pretty_print()
        old_messages.append(message)
    if obj[node].get("next", None):
        print(f"**NEXT**: {obj[node]['next']}")
```

```
================================ Ai Message ================================

The bar chart visualizing the frequency of each issue category for the latest 30 issues from the 'langchain-ai/langchain' GitHub repository has been created.

The chart illustrates the distribution of issues across categories such as 'Bug', 'Stale', 'Investigate', 'Vector Store', 'Other', and 'Documentation', with each bar's height corresponding to the number of issues in that category.
```



Multi-agent systems can be as flexible as needed: Modularity, Specialization, and Control

Modularity

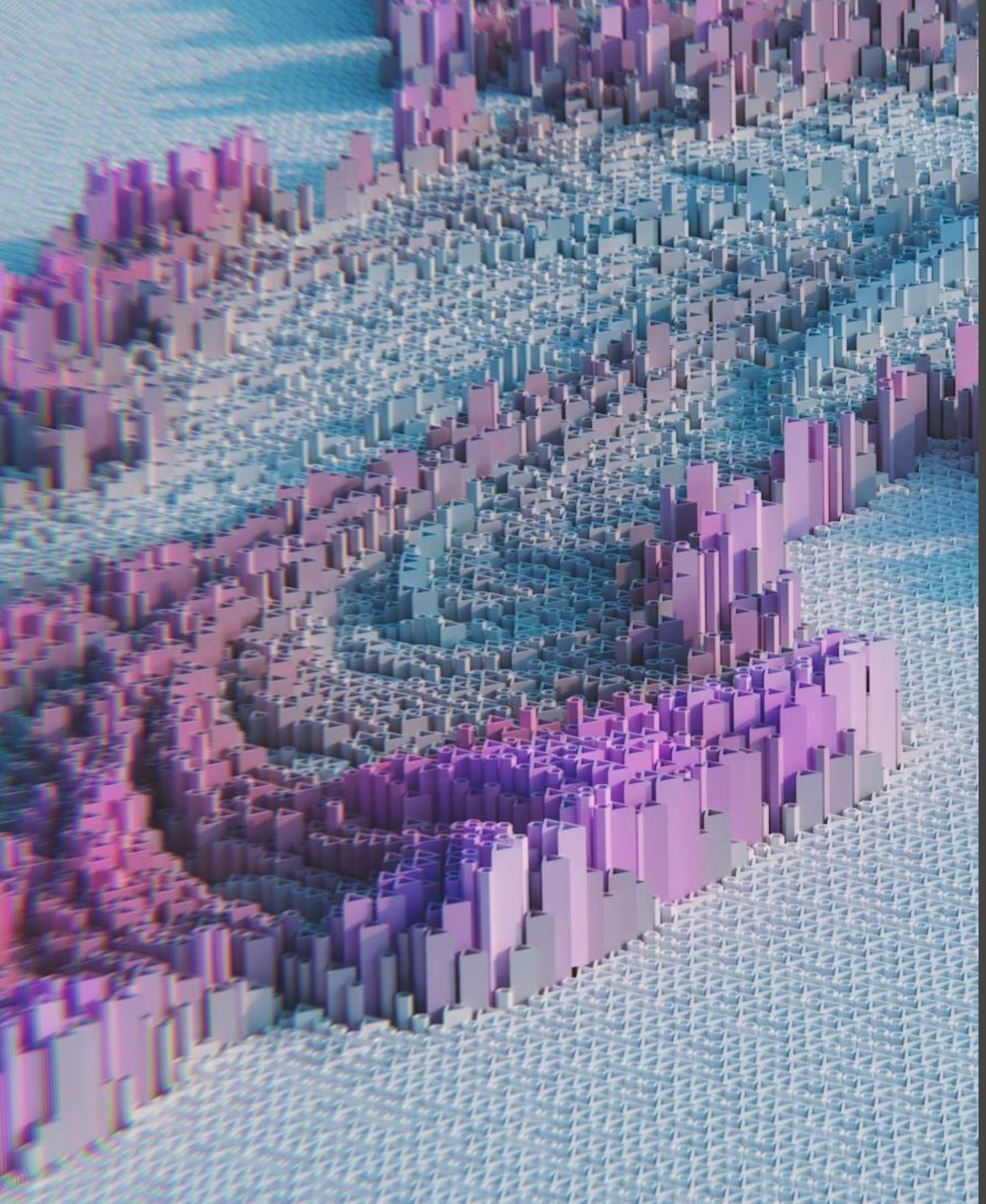
Graphs can be easily extended with the addition of new nodes that easily integrate within the multi-agent architecture, and enhance the service value offering

Specialization

Expert agents, with detailed roles, models, and capabilities, can focus on specific domains, achieving the best performance on specific tasks

Control

Agent communication can be controlled, or parametrized, allowing us, developers, to manage the data flow within the network nodes



We are only scratching the surface

- There are hundreds of tools to implement Multi-agent Solutions
- Reasoning capabilities will greatly improve multi and single-agent solutions
- Memory optimization techniques will allow bigger networks and complex pipelines

Thank You

https://www.youtube.com/@RicardoSantosDiaz

https://github.com/Tibiritabara

https://www.linkedin.com/in/ricardosantosdiaz/