Appendix C - Quick overview of Agentic Frameworks

LangChain

LangChain is a framework for developing applications powered by LLMs. Its core strength lies in its LangChain Expression Language (LCEL), which allows you to "pipe" components together into a chain. This creates a clear, linear sequence where the output of one step becomes the input for the next. It's built for workflows that are Directed Acyclic Graphs (DAGs), meaning the process flows in one direction without loops.

Use it for:

- Simple RAG: Retrieve a document, create a prompt, get an answer from an LLM.
- Summarization: Take user text, feed it to a summarization prompt, and return the output.
- Extraction: Extract structured data (like JSON) from a block of text.

Python

```
# A simple LCEL chain conceptually
# (This is not runnable code, just illustrates the flow)
chain = prompt | model | output_parse
```

LangGraph

LangGraph is a library built on top of LangChain to handle more advanced agentic systems. It allows you to define your workflow as a graph with nodes (functions or LCEL chains) and edges (conditional logic). Its main advantage is the ability to create cycles, allowing the application to loop, retry, or call tools in a flexible order until a task is complete. It explicitly manages the application state, which is passed between nodes and updated throughout the process.

Use it for:

- Multi-agent Systems: A supervisor agent routes tasks to specialized worker agents, potentially looping until the goal is met.

- Plan-and-Execute Agents: An agent creates a plan, executes a step, and then loops back to update the plan based on the result.
- Human-in-the-Loop: The graph can wait for human input before deciding which node to go to next.

Feature	LangChain	LangGraph
Core Abstraction	Chain (using LCEL)	Graph of Nodes
Workflow Type	Linear (Directed Acyclic Graph)	Cyclical (Graphs with loops)
State Management	Generally stateless per run	Explicit and persistent state object
Primary Use	Simple, predictable sequences	Complex, dynamic, stateful agents

Which One Should You Use?

- Choose LangChain when your application has a clear, predictable, and linear flow of steps. If you can define the process from A to B to C without needing to loop back, LangChain with LCEL is the perfect tool.
- Choose LangGraph when you need your application to reason, plan, or operate in a loop. If your agent needs to use tools, reflect on the results, and potentially try again with a different approach, you need the cyclical and stateful nature of LangGraph.

Python

```
# Graph state
class State(TypedDict):
   topic: str
   joke: str
   story: str
   poem: str
   combined_output: str

# Nodes
def call_llm_1(state: State):
   """First LLM call to generate initial joke"""

   msg = llm.invoke(f"Write a joke about {state['topic']}")
   return {"joke": msg.content}
```

```
def call_llm_2(state: State):
    """Second LLM call to generate story"""

    msg = llm.invoke(f"Write a story about {state['topic']}")
    return {"story": msg.content}

def call_llm_3(state: State):
    """Third LLM call to generate poem"""

    msg = llm.invoke(f"Write a poem about {state['topic']}")
    return {"poem": msg.content}

def aggregator(state: State):
    """Combine the joke and story into a single output"""

    combined = f"Here's a story, joke, and poem about
{state['topic']}!\n\n"
    combined += f"STORY:\n{state['story']}\n\n"
    combined += f"JOKE:\n{state['joke']}\n\n"
    combined += f"POEM:\n{state['poem']}"
    return {"combined_output": combined}

# Build workflow
parallel_builder = StateGraph(State)

# Add nodes
parallel_builder.add_node("call_llm_1", call_llm_1)
parallel_builder.add_node("call_llm_2", call_llm_2)
parallel_builder.add_node("call_llm_3", call_llm_3)
parallel_builder.add_node("aggregator", aggregator)

# Add edges to connect nodes
parallel_builder.add_edge(START, "call_llm_1")
parallel_builder.add_edge(START, "call_llm_2")
parallel_builder.add_edge(START, "call_llm_3")
parallel_builder.add_edge("call_llm_1", "aggregator")
parallel_builder.add_edge("call_llm_2", "aggregator")
parallel_builder.add_edge("call_llm_3", "aggregator")
parallel_builder.add_edge("aggregator", END)
parallel_workflow = parallel_builder.compile()

# Show workflow
display(Image(parallel_workflow.get_graph().draw_mermaid_png()))

# Invoke
state = parallel_workflow.invoke({"topic": "cats"})
print(state["combined_output"])
```

This code defines and runs a LangGraph workflow that operates in parallel. Its main purpose is to simultaneously generate a joke, a story, and a poem about a given topic and then combine them into a single, formatted text output.

Google's ADK

Google's Agent Development Kit, or ADK, provides a high-level, structured framework for building and deploying applications composed of multiple, interacting AI agents. It contrasts with LangChain and LangGraph by offering a more opinionated and production-oriented system for orchestrating agent collaboration, rather than providing the fundamental building blocks for an agent's internal logic.

LangChain operates at the most foundational level, offering the components and standardized interfaces to create sequences of operations, such as calling a model and parsing its output. LangGraph extends this by introducing a more flexible and powerful control flow; it treats an agent's workflow as a stateful graph. Using LangGraph, a developer explicitly defines nodes, which are functions or tools, and edges, which dictate the path of execution. This graph structure allows for complex, cyclical reasoning where the system can loop, retry tasks, and make decisions based on an explicitly managed state object that is passed between nodes. It gives the developer fine-grained control over a single agent's thought process or the ability to construct a multi-agent system from first principles.

Google's ADK abstracts away much of this low-level graph construction. Instead of asking the developer to define every node and edge, it provides pre-built architectural patterns for multi-agent interaction. For instance, ADK has built-in agent types like SequentialAgent or ParallelAgent, which manage the flow of control between different agents automatically. It is architected around the concept of a "team" of agents, often with a primary agent delegating tasks to specialized sub-agents. State and session management are handled more implicitly by the framework, providing a more cohesive but less granular approach than LangGraph's explicit state passing. Therefore, while LangGraph gives you the detailed tools to design the intricate wiring of a single robot or a team, Google's ADK gives you a factory assembly line designed to build and manage a fleet of robots that already know how to work together.

Python

```
from google.adk.agents import LlmAgent
from google.adk.tools import google_Search

dice_agent = LlmAgent(
```

```
   model="gemini-2.0-flash-exp",
   name="question_answer_agent",
   description="A helpful assistant agent that can answer
questions.",
   instruction="""Respond to the query using google search""",
   tools=[google_search],
)
```

This code creates a search-augmented agent. When this agent receives a question, it will not just rely on its pre-existing knowledge. Instead, following its instructions, it will use the Google Search tool to find relevant, real-time information from the web and then use that information to construct its answer.

Crew.AI

CrewAI offers an orchestration framework for building multi-agent systems by focusing on collaborative roles and structured processes. It operates at a higher level of abstraction than foundational toolkits, providing a conceptual model that mirrors a human team. Instead of defining the granular flow of logic as a graph, the developer defines the actors and their assignments, and CrewAI manages their interaction.

The core components of this framework are Agents, Tasks, and the Crew. An Agent is defined not just by its function but by a persona, including a specific role, a goal, and a backstory, which guides its behavior and communication style. A Task is a discrete unit of work with a clear description and expected output, assigned to a specific Agent. The Crew is the cohesive unit that contains the Agents and the list of Tasks, and it executes a predefined Process. This process dictates the workflow, which is typically either sequential, where the output of one task becomes the input for the next in line, or hierarchical, where a manager-like agent delegates tasks and coordinates the workflow among other agents.

When compared to other frameworks, CrewAI occupies a distinct position. It moves away from the low-level, explicit state management and control flow of LangGraph, where a developer wires together every node and conditional edge. Instead of building a state machine, the developer designs a team charter. While Googlés ADK provides a comprehensive, production-oriented platform for the entire agent lifecycle, CrewAI concentrates specifically on the logic of agent collaboration and for simulating a team of specialists

Python

```
@crew
def crew(self) -> Crew:
   """Creates the research crew"""
   return Crew(
     agents=self.agents,
     tasks=self.tasks,
     process=Process.sequential,
     verbose=True,
   )
```

This code sets up a sequential workflow for a team of AI agents, where they tackle a list of tasks in a specific order, with detailed logging enabled to monitor their progress.

Other agent development framework

Microsoft AutoGen: AutoGen is a framework centered on orchestrating multiple agents that solve tasks through conversation. Its architecture enables agents with distinct capabilities to interact, allowing for complex problem decomposition and collaborative resolution. The primary advantage of AutoGen is its flexible, conversation-driven approach that supports dynamic and complex multi-agent interactions. However, this conversational paradigm can lead to less predictable execution paths and may require sophisticated prompt engineering to ensure tasks converge efficiently.

LlamaIndex: LlamaIndex is fundamentally a data framework designed to connect large language models with external and private data sources. It excels at creating sophisticated data ingestion and retrieval pipelines, which are essential for building knowledgeable agents that can perform RAG. While its data indexing and querying capabilities are exceptionally powerful for creating context-aware agents, its native tools for complex agentic control flow and multi-agent orchestration are less developed compared to agent-first frameworks. LlamaIndex is optimal when the core technical challenge is data retrieval and synthesis.

Haystack: Haystack is an open-source framework engineered for building scalable and production-ready search systems powered by language models. Its architecture is composed of modular, interoperable nodes that form pipelines for document retrieval, question answering, and summarization. The main strength of Haystack is its focus on performance and scalability for large-scale information retrieval tasks, making it suitable for enterprise-grade applications. A potential trade-off is that its design, optimized for search pipelines, can be more rigid for implementing highly dynamic and creative agentic behaviors.

MetaGPT: MetaGPT implements a multi-agent system by assigning roles and tasks based on a predefined set of Standard Operating Procedures (SOPs). This framework structures agent collaboration to mimic a software development company, with agents taking on roles like product managers or engineers to complete complex tasks. This SOP-driven approach results in highly structured and coherent outputs, which is a significant advantage for specialized domains like code generation. The framework's primary limitation is its high degree of specialization, making it less adaptable for general-purpose agentic tasks outside of its core design.

SuperAGI: SuperAGI is an open-source framework designed to provide a complete lifecycle management system for autonomous agents. It includes features for agent provisioning, monitoring, and a graphical interface, aiming to enhance the reliability of agent execution. The key benefit is its focus on production-readiness, with built-in mechanisms to handle common failure modes like looping and to provide observability into agent performance. A potential drawback is that its comprehensive platform approach can introduce more complexity and overhead than a more lightweight, library-based framework.

Semantic Kernel: Developed by Microsoft, Semantic Kernel is an SDK that integrates large language models with conventional programming code through a system of "plugins" and "planners." It allows an LLM to invoke native functions and orchestrate workflows, effectively treating the model as a reasoning engine within a larger software application. Its primary strength is its seamless integration with existing enterprise codebases, particularly in .NET and Python environments. The conceptual overhead of its plugin and planner architecture can present a steeper learning curve compared to more straightforward agent frameworks.

Strands Agents: An AWS lightweight and flexible SDK that uses a model-driven approach for building and running AI agents. It is designed to be simple and scalable, supporting everything from basic conversational assistants to complex multi-agent autonomous systems. The framework is model-agnostic, offering broad support for various LLM providers, and includes native integration with the MCP for easy access to external tools. Its core advantage is its simplicity and flexibility, with a customizable agent loop that is easy to get started with. A potential trade-off is that its lightweight design means developers may need to build out more of the surrounding operational infrastructure, such as advanced monitoring or lifecycle management systems, which more comprehensive frameworks might provide out-of-the-box.

Conclusion

The landscape of agentic frameworks offers a diverse spectrum of tools, from low-level libraries for defining agent logic to high-level platforms for orchestrating multi-agent collaboration. At the foundational level, LangChain enables simple, linear workflows, while LangGraph introduces stateful, cyclical graphs for more complex reasoning. Higher-level frameworks like CrewAI and Google's ADK shift the focus to orchestrating teams of agents with predefined roles, while others like LlamaIndex specialize in data-intensive applications. This variety presents developers with a core trade-off between the granular control of graph-based systems and the streamlined development of more opinionated platforms. Consequently, selecting the right framework hinges on whether the application requires a simple sequence, a dynamic reasoning loop, or a managed team of specialists. Ultimately, this evolving ecosystem empowers developers to build increasingly sophisticated AI systems by choosing the precise level of abstraction their project demands.

References

1. LangChain, https://www.langchain.com/
2. LangGraph, https://www.langchain.com/langgraph
3. Google's ADK, https://google.github.io/adk-docs/
4. Crew.AI, https://docs.crewai.com/en/introduction