The State of AI Agent Frameworks: Comparing LangGraph, OpenAI Agent SDK, Google ADK, and AWS Bedrock Agents

59 min read · Jul 10, 2025

Roberto Infante Following

Introduction

AI is moving beyond simple chatbots to more **autonomous agents** that can not only chat, but also **take actions**. These AI agents are powered by Large Language Models (LLMs) and equipped with the ability to **reason**, **plan**, call **external tools/APIs**, and **remember context** over time. In other words, an agent is like a smart digital assistant that doesn't just generate text, but can interact with other systems to accomplish goals.

One way to think about agents is to contrast them with traditional **deterministic workflows**. In a deterministic workflow, you as the developer specify a fixed sequence of steps or a flowchart to accomplish a task. The path is predetermined and doesn't change at runtime. By contrast, an **LLM-based agent** uses the model's **chain-of-thought reasoning** to dynamically decide its steps. Rather than follow a script, the agent "figures out" the workflow on the fly: it can choose which tool to use next, when to stop, or how to recover from errors by itself. This dynamic decision-making is often implemented via the *ReAct* pattern — a loop where the agent alternates between **thinking** ("thought" as a reasoning step) and **acting** (executing a tool or function), observing the result, and then thinking again, and so on until it reaches a solution.

Crucial to these agents are a few components that have rapidly become standard across the industry:

- **Tool/Function Calling:** Modern LLMs can call functions or tools through well-defined APIs. Initially, we used raw text prompting to have models output, say, "Search(query)" as text and then parse it. Now, with structured **function calling** interfaces (like OpenAI's function calls or JSON outputs), the process is more robust. The agent chooses a function to invoke (e.g. a web search, database query, calculator) and supplies structured inputs; the framework executes it and feeds the results back to the LLM. This closes the loop between the AI's reasoning and real-world actions.

- **Memory (State):** To be effective, agents need memory. This includes short-term memory (the conversation or operation history within a session) and long-term memory (knowledge retained across sessions or pertaining to a specific user). Memory enables personalization and continuity — for example, remembering a user's preferences or facts mentioned earlier. Without memory, the agent would be stateless and forgetful, limiting its usefulness in interactive or multi-step tasks.

- **Multi-Step Reasoning Loop:** The agent frameworks provide a runtime loop that lets an LLM refine its plan step by step. Instead of a single prompt → completion, the agent can have a whole dialogue with itself: it generates an idea, gets an observation (from a tool result), updates its plan, and continues. This loop continues until the task is completed or some criteria are met. It's essentially giving the LLM a scratchpad and the ability to execute as it reasons.

- **System Instructions & Role Definitions:** Each agent can be guided by initial system instructions (e.g. "You are an assistant that helps with travel planning and can use tools like flight search"). These instructions set the overall behavior and policy. Additionally, each tool comes with a description and usage instructions that inform the LLM when and how to use it. By carefully crafting these prompts and instructions, developers shape the agent's strategy and ensure it uses tools correctly.

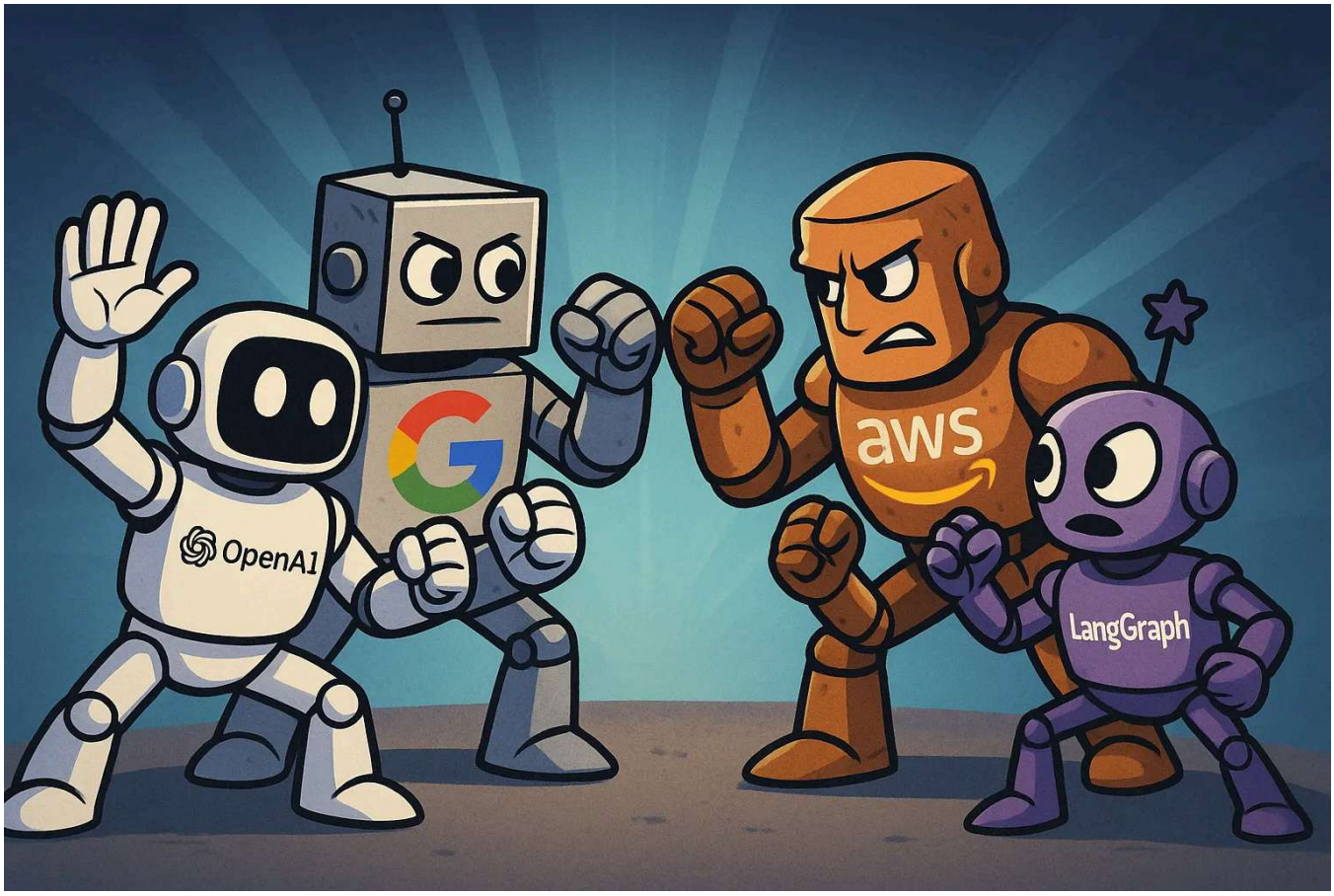
- **Evaluation & Guardrails:** As agents become more complex, there's a growing need for **evaluation** (to measure how well the agent is performing and catch regressions or errors) and **guardrails** (to enforce safety and business rules). All major frameworks now include features to evaluate agents' outputs and reasoning steps, and to impose constraints — for example, blocking disallowed content, preventing certain tool usage, or sanitizing sensitive data. Evaluations

help developers gauge success rates and failure modes, while guardrails intervene when the agent goes off-policy or produces risky content.

What's fascinating is that despite being developed by different teams, many agent frameworks are **converging in architecture**. LangChain's open-source **LangGraph**, OpenAI's new **Agents SDK**, Google's **Agent Development Kit (ADK)**, and AWS's **Bedrock Agents** all provide similar building blocks: they let you define tools/functions, give the LLM a reasoning loop to use those tools, maintain memory, and customize system behaviors. However, they differ in their approach to implementation, level of abstraction, integration with cloud platforms, and developer experience. Some are completely open and model-agnostic, while others are tightly integrated with a specific cloud ecosystem. Some give you a visual workflow interface, others are code-first. Some emphasize a minimal core, others provide a batteries-included kitchen sink of features.

In this article, we'll dive deep into these four leading agent frameworks, compare them across multiple dimensions, and highlight trends in design. By the end, you should have a clear picture of:

- The strengths and focus of **LangChain/LangGraph** (open-source, flexible),
- The approach of **OpenAI's Agents SDK** (lightweight and multi-model friendly),
- The capabilities of **Google's ADK** (powerful orchestration with GCP integration),
- The features of **AWS Bedrock Agents** (managed service convenience on AWS),
- How they differ in workflow modeling, tool integration, memory, model support, evaluation, guardrails, and more.



Whether you're building an internal automation agent, a customer support chatbot, or a complex multi-system AI orchestrator, understanding these frameworks will help you choose the right tool (or combination of tools) for the job.

Landscape of AI Agent Frameworks

Let's start with a brief overview of each framework in the landscape, to set the stage for the detailed comparison.

LangChain / LangGraph: *LangChain* is an open-source library that pioneered the idea of chaining LLM calls and tool usages. It became popular for its wide range of integrations (APIs, databases, web browsers, etc.) and for popularizing the ReAct agent pattern in practice. Building on that, the LangChain team introduced **LangGraph** as a lower-level orchestration framework for designing **stateful, multi-step agent workflows**. LangGraph uses a **graph-based model**: you can explicitly define nodes (each might be an LLM invocation, a tool call, a conditional branch, etc.) and edges (transitions) forming a flowchart or DAG of an agent's logic. This is great for making complex agent plans transparent and durable. LangGraph still allows dynamic decision-making (you can include an "LLM decide next step" node), but it wraps it in a structure that's easier to debug and maintain. LangChain/LangGraph are vendor-agnostic: they work with many LLM providers

(OpenAI, Anthropic, local models, etc.) and can be deployed anywhere. They have a thriving community and lots of pre-built components to reuse.

OpenAI's Agents SDK: Announced in early 2025, OpenAI's Agents SDK is a minimalist, open-source toolkit for building agents. Despite being from OpenAI, it's designed to be **model-agnostic** (it can work with OpenAI's models or any model that offers a Chat Completions API). The emphasis here is on simplicity and quick development. The SDK provides abstractions for an Agent that can use tools (leveraging OpenAI's function calling under the hood), a Runner to execute an agent through a conversation, and mechanisms to compose multiple agents or chain them. It doesn't come with a huge library of tools out-of-the-box — you typically plug in your own functions — which keeps it lean. However, it includes important features like **guardrails**, tracing, and a simple way to manage multi-agent workflows (for instance, handing off from one agent to another). If you want a straightforward way to turn an LLM into an agent that can take actions, and you plan to leverage the latest OpenAI API capabilities, this SDK is built for that.

Google's Agent Development Kit (ADK): Google's ADK is an ambitious open-source framework aimed at enterprise-grade agent systems. It provides a **code-first but structured** approach to building agents, heavily inspired by software engineering best practices. ADK supports different types of agents:

- **LLMAgent** for open-ended reasoning (the agent uses an LLM to decide next actions, classic chain-of-thought agent).
- **Workflow Agents** like **SequentialAgent**, **ParallelAgent**, **LoopAgent** for deterministic or semi-deterministic flows. These let you define explicit sequences, parallel branches, or iterative loops in an agent's operation.
- You can even nest agents or have agents call other agents as tools, enabling hierarchical multi-agent systems (e.g., a manager agent delegating subtasks to specialist agents).
  ADK comes with a rich set of **pre-built tools** (web search, code execution, etc.) and can integrate with external APIs via function tools or even ingest OpenAPI specs to auto-generate tool interfaces. It also provides a **Developer UI and CLI** that let you run agents step-by-step, inspect their state and events, and debug easily — highlighting Google's focus on making agent development feel more like normal debugging of programs. ADK is optimized to work with Google's Vertex AI (and the upcoming Gemini models), including seamless deployment to

Vertex AI's **Agent Engine** for production scaling. But it remains model-agnostic at the code level, so you could use other models if desired. Notably, ADK bakes in evaluation utilities and encourages building **safe, secure agents** with guardrail patterns and use of Google's content safety services.

AWS's Amazon Bedrock Agents: Amazon Bedrock is AWS's platform for foundation models, and **Bedrock Agents** is a fully managed service to build and run agents in the cloud. Unlike the other three, which are libraries you integrate into your app, Bedrock Agents is a service where you configure the agent in AWS and AWS runs it for you (with AWS managing the prompt orchestration, tool execution, scaling, etc.). The goal is to let you incorporate an AI agent into your application without having to write the full orchestration code or host the model yourself. You define what the agent can do by creating **Action Groups** — each action corresponds to an API or function call that the agent is allowed to invoke. Under the hood, these are backed by AWS Lambda functions or AWS APIs that you connect. For example, you might have an action group for "Travel Booking" with actions like `SearchFlights`, `BookHotel`, implemented via your Lambdas. The agent (powered by a foundation model like Anthropic Claude or Amazon's Titan) will decide which action to use when, and Bedrock will execute it and feed the result back to the model's context. Bedrock Agents also support a **knowledge base** integration: you can provide your own documents or data (automatically indexed into embeddings) so that the agent can do retrieval on your data when answering questions (this is essentially retrieval-augmented generation built into the service). Because it's a managed service, Bedrock handles the conversation memory for you (tracking the dialog state), and provides a web-based **trace viewer** and analytics so you can see each step the agent took, how long it took, etc. The trade-off is that it's somewhat AWS-centric: you're mostly using AWS's models (or partner models on Bedrock) and AWS's infra. It's the fastest route if you're already on AWS and want things like security, logging, and scaling taken care of, but it's less customizable than an open-source library.

With these overviews in mind, let's compare these frameworks more directly. In the next section, we'll examine a series of key dimensions (from how you design workflows, to tool integration, to safety features) and see how LangChain, OpenAI SDK, Google ADK, and AWS Bedrock stack up against each other.

Frameworks Compared: Key Dimensions

We will compare the four frameworks on a number of important dimensions. For each dimension, we'll highlight how each framework approaches the problem, so you can see the similarities and differences side by side.

A. Workflow Modeling

- **LangChain / LangGraph:** LangChain introduced the concept of chaining calls, but with LangGraph it now offers a true **graphical workflow model**. Developers can explicitly construct a directed graph of nodes representing steps. For example, you could have an LLM node for analyzing user input, a branching node that routes to different subgraphs based on a condition, parallel nodes to do multiple calls concurrently, etc. This visual or code-defined graph gives you **full control** and predictability. You know exactly what steps can occur. LangGraph still allows dynamic ReAct-style choices inside a node (e.g., an LLM node could decide to call a tool), but the outer structure is under developer control. Additionally, LangGraph supports **human-in-the-loop** at any point in the graph — you could designate a step where human approval is needed before continuing. This is great for high-stakes applications. In summary, LangChain/LangGraph leans toward *explicit workflow orchestration* but with the flexibility to plug in autonomous reasoning where needed.

- **OpenAI Agents SDK:** The OpenAI SDK is **code-first and minimalist**. You typically create one or more Agent classes in code and then use a Runner to execute them. There's no visual editor; you define the flow through function calls or a simple loop. By default, an agent run is a single-agent reasoning loop (the agent will use tools and think until it stops). But if you want a multi-step or multi-agent workflow, you orchestrate it in code. For instance, you could call `agent1.run(input)`, get a result or decision, then decide in your Python code to pass control to `agent2`. OpenAI supports a concept called **Handoffs** – basically an agent can output a special action that says "handoff to Agent B with this new input", effectively transferring the conversation to another agent. Using these, you can create branching logic and multi-agent pipelines, but it's all defined through code rather than a pre-defined static graph. There isn't a built-in concept of parallel execution in the SDK; if you wanted parallelism, you'd likely handle it with async Python tasks. The emphasis is on *simplicity* and letting the LLM drive the sequence, with the developer interjecting only when needed.

- **Google ADK:** ADK offers the **best of both worlds**: you can do fixed workflows or dynamic reasoning, or mix them. With ADK's **SequentialAgent**, you can list a

sequence of steps (each step might call a tool or another agent) that will be executed in order — no LLM deciding the order. With **ParallelAgent**, you can fire off multiple actions at once (useful if you have, say, two independent calls that can happen concurrently to save time). **LoopAgent** can repeat a set of steps until a condition is met (like iteratively refining an answer). On the other hand, the **LLMAgent** is the free-form ReAct agent where the LLM decides the next tool/action at each step. Crucially, you can combine these. For example, you might use a SequentialAgent as a high-level controller that always does A then B then C (deterministic), but for step B it invokes an LLMAgent to handle an open-ended subtask. ADK also supports human-in-the-loop patterns, as you can add callbacks or custom steps to pause for human review. The key advantage of ADK in workflow modeling is that it has **explicit structures for common patterns** (sequence, parallel, etc.) built-in, so you don't have to manually code those control flows from scratch — you declare them. This can make complex workflows easier to manage and reason about.

- **AWS Bedrock Agents:** With Bedrock, the workflow is largely **implicit and driven by the model**. You don't write code to sequence steps; instead, you configure the agent's capabilities and some optional rules. The agent, powered by the LLM, will decide which action to take first, then second, etc., based on the user request and what tools are available. So it's heavily ReAct style under the hood (though AWS doesn't expose the prompt chain to you directly, you can view it in the trace). There are ways to inject some determinism: for instance, you can set up *contextual hints* in prompt templates that nudge the agent to follow certain order, or use the structured **pre-processing** and **post-processing** steps AWS provides (these are like hidden workflow stages: e.g., a pre-processing step could analyze user input or do classification before the main orchestration). But compared to the others, you have less direct control over the exact sequence of tool calls — you entrust that to the Bedrock agent's reasoning. If you absolutely need a fixed sequence, you might actually not use an "agent" in Bedrock but rather orchestrate multiple InvokeAgent calls in your own code (making Bedrock agent do one step per call). However, for most use cases, you give the agent a goal and let it figure out the workflow. The benefit is simplicity and abstraction (you don't have to map out the flow yourself), but the downside is it can be harder to predict or enforce an exact procedure.

B. Tool Integration and Agent Toolkits

- **LangChain / LangGraph:** LangChain comes with a **large toolkit of ready-made tools** contributed by the community and the LangChain team: web searchers, wiki lookups, calculators, Python interpreters, SQL database query tools, etc. Adding a tool in LangChain is as simple as defining a Python function (with a docstring describing what it does) or using one of the built-ins, and then giving the list of tools to the agent. Under the hood, LangChain can use two modes for tool use: the older approach where the LLM outputs a text command like "`Action: Search\nAction Input: 'climate data'`" and LangChain parses it, or the newer approach using model's native function calling (if the LLM supports it). With function calling, LangChain will translate your Python function into a JSON schema automatically (often using type hints or a provided schema), so the LLM can call it more reliably. LangChain's ReAct implementation was one of the first, so it's pretty mature: the agent will cycle through thought -> tool -> observation -> thought, and there are configurable limits on number of iterations or stop conditions. LangChain also supports **tool priority / selection logic**: you can constrain which tools are available at certain times or have the agent use a planner to pick a tool sequence. All of this is fairly configurable but requires understanding LangChain's abstractions like Tool, AgentExecutor, etc.

- **OpenAI Agents SDK:** The OpenAI SDK makes tool integration extremely straightforward if you're using OpenAI's models with function calling. You can define a Python function and decorate it (for example, using `@function_tool` from the SDK) or register it with the agent. The function's name, description, and parameter schema (which the SDK can derive from type hints or a Pydantic model) will be exposed to the model. When the agent is running, if the model decides to call a function, the SDK will catch that, execute your Python function, and pass the return value back to the model's context. The developer doesn't have to parse any text – it's handled via the structured API. The ReAct loop is essentially built into the model's own behavior thanks to function calling support. One thing to note is that OpenAI's approach encourages fairly **literal function usage**: the model chooses exactly one function and you get the result. If you need to do more complex sequences (like call function A then B then C in one go), you either rely on the model to figure out to do that through multiple turns, or you implement a workflow around it. The SDK, being minimal, doesn't enforce a certain way – it basically says "here are your tools, here's the model, let it run." If you want to put constraints (like forcing a certain argument format or doing some validation), you might implement that either by adjusting the

function's code or adding guardrails (which we'll cover later). In short: Tools in the Agents SDK are **explicitly defined by you** and seamlessly invoked by the model, but the SDK itself doesn't provide a library of them – you bring your own or integrate external ones as needed.

- **Google ADK:** ADK shines with a **rich tool ecosystem**. It provides a variety of built-in tools out-of-the-box. For example, common utilities like web search, a code execution sandbox, calculators, etc., are readily available. It also supports **FunctionTool** (wrapping a Python function similar to others), **OpenAPITool** (you can point it at an OpenAPI spec for some service, and it will generate a tool interface for the agent to call that API's endpoints), **AgentTool** (where one agent can use another agent as a tool, effectively delegating a subtask). This means you can integrate almost anything as a tool: cloud services, databases, other agent frameworks, etc. ADK likely has **tool schemas** based on Pydantic or similar, where you define what inputs a tool expects. When an LLM agent is running, it uses a standard format to decide on a tool and the ADK runtime will execute it and handle returning the result. Internally, ADK is flexible; you can even mark tools as blocking or non-blocking, handle long-running tools (like an asynchronous database query), and more. The big advantage of ADK's approach is the **breadth of integration** (especially with Google Cloud tools if you're in that environment — e.g., calling Google Calendar API, or Cloud Storage, might be made easy). Also, because ADK is open-source, the community can add more tools to the ecosystem. Tools are registered with an agent through a toolbox or config, often when initializing the agent or via a builder pattern. ADK's approach to ReAct is standard: the LLM emits an "action" and "action input," ADK interprets it and calls the tool, then feeds back the tool's output for the next reasoning cycle.

- **AWS Bedrock Agents:** In Bedrock, the concept of tools is represented by **Action Groups and actions**. Each action is essentially an API that the agent can call. When you set up an action, you provide an OpenAPI schema (directly in JSON or via an API call to Bedrock) that defines the action's input parameters and output structure, and you associate that with a specific AWS Lambda function. When the Bedrock agent is running and the model decides to invoke an action, Bedrock will parse the model's output (which might be something like a JSON with an `invokeAction` field), match it to the correct action schema, validate the parameters, then call your Lambda. The result from the Lambda (which should conform to the output schema) is then passed back into the model's context.

This approach strongly separates the model from the actual implementation of the tool – the model just knows "there's an action called SearchFlights and it requires destination, dates, etc." and if it calls it, AWS handles the rest. One benefit here is **security and manageability**: you can leverage AWS IAM to control what those Lambdas do, monitor their usage, and keep your agent's "hands" limited to what you explicitly allow. Bedrock also provides some ready-made "utilities" (for example, it has integration with the AWS Knowledge Base for retrieval, and possibly some basic tools like a math solver or date converter might be built in, though much of it you configure yourself). Compared to others, setting up a new tool in Bedrock is a bit more involved (you have to write a Lambda and an OpenAPI spec, or use AWS's developer console to configure it), but once set up, it's very robust. Bedrock's agent will automatically handle sequences of actions by continuing the prompt until completion, similar to how others do – but remember, you don't see the intermediate chain-of-thought unless you look at the trace logs.

C. Native Integrations and Ecosystem

- **LangChain / LangGraph:** Being open-source and community-driven, LangChain probably has the largest ecosystem of integrations. Anything from popular vector databases (Pinecone, Weaviate, FAISS, etc.) for memory, to third-party APIs (Google Search, SerpAPI, Twilio, you name it) — someone has likely written a LangChain wrapper for it. LangChain doesn't provide cloud services itself, but it *integrates with* cloud services. For example, you can easily use OpenAI's APIs, or Hugging Face models, or Azure's endpoints through LangChain modules. LangGraph focuses on orchestration, but because it's built on LangChain, you have access to that whole ecosystem of tools and connectors. Also, LangChain has *LangChain Hub* (a place to share prompts and chains) and is quite extensible. If your company has internal tools, you can wrap them as LangChain Tools fairly easily. The openness and strong community support mean if there's a tool you need, chances are it's either already available or you can create it and contribute it back.

- **OpenAI Agents SDK:** The Agents SDK is maintained by OpenAI and is relatively new, so its out-of-the-box integrations are limited mainly to what OpenAI provides or what you code yourself. OpenAI itself, however, is expanding its offerings — for example, their new "Responses API" includes some built-in tools like a file search and a browser (for web browsing) that the model can use if you

enable them. The Agents SDK can definitely call those OpenAI-provided tools. But beyond that, if you want to integrate with, say, a database or a non-OpenAI API, you'll be implementing that function. The SDK isn't bundled with connectors to third-party services (unlike LangChain). That said, because it's simple and Pydantic-friendly, it's straightforward to integrate. It's more of a **DIY approach** or use it in conjunction with other libraries. Also, keep in mind OpenAI's ecosystem: there is an **OpenAI evals** library for evaluation, and the Agents SDK is designed to work nicely with OpenAI's platform (for example, logging traces to the OpenAI cloud or using OpenAI's content moderation endpoint as a guardrail). But if you're looking for a one-stop shop of community extensions, it's not yet as rich as LangChain's ecosystem.

- **Google ADK:** Google's ADK, while open source, is part of a bigger Google ecosystem. It has native integrations with **Google Cloud services**. For instance, ADK can integrate with Google Search (via an API), Google Calendar, Google Drive, etc., through provided tools (the "Google Cloud tools" section suggests tools for GCP services might be readily available). If you deploy on Vertex AI's Agent Engine, it can directly use Vertex's endpoints and data services with authentication handled. Also, ADK mentions integration with third-party frameworks like LangChain — meaning you could potentially incorporate LangChain tools or chains within ADK if you wanted (though the details of that would require some adapters). ADK also has a concept of **Artifacts** (for handling files/images), which is quite useful if your agent needs to handle PDFs or images as part of its workflow (e.g., parsing a PDF then answering questions; ADK might integrate with Google's Document AI or Vision AI for such tasks). The community around ADK is growing, and given Google's push, we might see more community-contributed integrations over time. In summary, ADK is strong if you're in Google's world and want plug-and-play support for those services; it's also flexible enough to call anything else, but you might have to implement some connectors manually or reuse existing Python libraries.

- **AWS Bedrock Agents:** Since Bedrock Agents is a managed service, its integrations are very AWS-centric. The primary way to integrate external systems is via AWS Lambda. Essentially, anything you can do in a Lambda, you can expose as an action to the agent. This means the door is open to connect to databases, call other AWS services (DynamoDB, S3, etc.), or even call external APIs from within your Lambda code. AWS has done a lot to make their AI services connect with their existing cloud ecosystem: for example, a Bedrock

agent can be given permissions (IAM roles) to, say, read from an S3 bucket or write to a database when a certain action is invoked, which is powerful for enterprise apps. Additionally, the **Knowledge Base** integration allows connecting data from S3 or other sources through Amazon Kendra (for search) or vector indexes that AWS manages — which is an out-of-the-box way to add RAG capabilities. Unlike others, you won't see a big library of pre-built "Bedrock tools" floating around (because each action has to be registered in the service). Instead, AWS provides examples and templates (like a sample travel booking agent, a sample QA agent, etc., in their docs) that show how to set up common integrations. The benefit is that you're likely using well-architected AWS components (with security, monitoring), but the drawback is you might need to write more glue code (in Lambdas) for each integration compared to an open-source library where someone might have already packaged it for you.

D. Prompting, Instructions, and System Messages

- **LangChain / LangGraph:** LangChain gives you a lot of flexibility in how you prompt your agents. Typically, you provide a **system prompt or instructions** when creating an agent (e.g., "You are a helpful travel assistant…"). LangChain's prompt templating features let you easily inject variables or tool descriptions into these prompts. In LangGraph, since you might have a complex workflow, you can assign prompts at different nodes. For example, one node could have a specific prompt guiding the LLM for that step. Tools in LangChain are accompanied by descriptions (and sometimes example usage) which are appended to the prompt context so the LLM knows what they do. LangChain doesn't enforce a specific prompting strategy — you can do zero-shot (just give tools and ask the model to figure it out) or few-shot (provide exemplars of how to use tools), or even hand it a "planning" prompt if you use a Plan-and-Execute style. Because LangChain is relatively low-level, you have to manage these prompts yourself (though they provide templates to start). There isn't a global "system policy" beyond what you put in the prompt, nor a formal separation of agent-level vs tool-level instructions (beyond the tool descriptions). However, since you can always add a system message before a tool invocation or after, you do have control if needed. In practice, LangChain agents often rely on a standard template: system message with overall role, then the conversation and a scratchpad section where the chain-of-thought accumulates. LangGraph's structure might instead manage that scratchpad for you behind the scenes.

- **OpenAI Agents SDK:** Creating an agent with OpenAI's SDK involves specifying a few key strings: the agent's **name** (just an identifier) and its **instructions** (which function as the system message guiding the agent's overall behavior). The SDK takes care of formatting that into the prompt along with user input and function/tool info. Tools' docstrings serve as the content for the function definitions that the model sees. The SDK likely uses a certain prompt format under the hood (possibly similar to OpenAI's own few-shot style for function calling, but details are abstracted). It's relatively straightforward: you can set `agent.instructions = "Your role is XYZ."` and maybe also give it an `agent.base_prompt` or context if needed. Because it's minimal, if you need fancy prompting logic (like injecting different system messages at different conversation turns or dynamically altering prompts), you may have to do that manually by updating the agent's instructions or using the Runner to add system messages. The OpenAI SDK also supports a concept of **agent memory via the Runner** (the Runner will accumulate messages from previous turns unless you clear it). But there isn't a separate memory object – it's basically chat history fed back in each time. Regarding tool-specific prompts: OpenAI's function calling means the model automatically gets a JSON schema and description for each tool, so that's handled implicitly. One neat feature is that because you can chain agents, you could have different agents with different system instructions in a pipeline (for example, an agent specialized in one task hands off to another with a different skill set and instructions). Summarily, OpenAI's approach to prompting is **simple and message-based** – you mostly rely on the model's ability to follow the given instructions and use the function definitions.

- **Google ADK:** ADK provides more **structured prompting controls**. Each agent in ADK has an **Instructions** setting (like a system prompt) and possibly separate prompt templates for different phases:

- When using the **LLMAgent**, ADK might let you customize how the prompt is built with the tools' info, how thoughts and observations are formatted, etc. Google's documentation mentions advanced prompt templating: for instance, ADK distinguishes between **pre-processing prompt**, **orchestration prompt**, and **post-processing prompt**. These correspond to stages in the agent's internal process. In Vertex AI's terms, pre-processing might classify or prepare the input, orchestration is the main ReAct loop, and post-processing finalizes the answer.

- ADK likely has defaults that are pretty good (especially if using Google's models which have particular formatting needs), but you can override them. For example, you could insert a company-specific disclaimer at the end, or enforce that the agent's final answer is in JSON by adjusting the post-processing template.

- ADK also supports something called **State** and **Tool Context**, which means you can carry structured data through the conversation outside of the text prompt. For instance, you might store the user's account ID in the session state and not rely on the model's memory for that — the tool context can then incorporate it when a tool is called. This gives a non-prompt way to maintain certain contextual info (less chance for the model to ignore or alter it).

- For multi-agent scenarios, ADK's agents communicate via a defined protocol, which likely has a message structure. But as a developer, you mostly configure how they talk through the agent settings rather than writing the prompt by hand each time.

- Since ADK is aiming for reliability, it might also encourage use of **structured outputs** (like you can define the expected output schema from an agent in certain cases, similar to how you'd define a Pydantic model for output and then ADK ensures the LLM's answer is parsed into that). This way you're not just getting free-form text when you need a specific format.

- In short, ADK gives you **granular control** if you want it, but also sensible defaults. You have a central place to set system-level instructions (like the agent's role and policy), and you can also fine-tune how prompts are constructed at each stage of the agent's reasoning pipeline.

- **AWS Bedrock Agents:** With Bedrock, you don't see or write the full prompt, but you do get to configure certain parts of it via the console or API. Bedrock defines a base prompt under the hood that handles how the agent should respond, how to format action calls, etc., but you can override sections of it:

- **User Prompt Template:** You can customize how the user's input is presented to the model. For example, you might prepend some context or system message to every user query like "You are an agent assisting with XYZ. The user says: {user_input}".

- **Orchestration Prompt Template:** This is the prompt that the model sees when it's deciding which action to take. AWS allows you to augment it, possibly by adding additional instructions like "Always check the user's location before recommending a service" or giving it hints on which action to use for certain intents.

- **Post-processing Prompt:** After the agent has done all its actions, there's a final prompt for generating the answer. You can modify this to change the answer style or add closing notes.

- These advanced prompt templates let you guide the agent's behavior without writing code. AWS provides a default that's general-purpose, but domain-specific agents benefit from tweaking these. For example, if you know the agent should always call a "VerifyIdentity" action at the start of a conversation, you could embed that instruction in the orchestration prompt.

- Since it's managed, the actual prompt combination and formatting (including the inclusion of relevant knowledge base documents or the serialization of action responses) is done by AWS. You primarily supply additional text or variables to shape it.

- On top of that, if you're using a model like Anthropic, the system prompt might include Anthropic's special tokens for safety (like the "Assistant should not do X"). AWS likely handles those automatically via model configuration and guardrails (we'll talk about guardrails soon).

- Overall, Bedrock's prompting strategy is **configuration-driven** rather than code. It's user-friendly if you prefer to tweak a few text fields in a console to adjust agent behavior, but it's less flexible than coding your own prompt logic as you would in LangChain or ADK. However, it saves time by providing a decent default that covers typical needs (like making sure the agent responds in a desired format, or says "I can't do that" when appropriate, etc.).

E. Memory and State Management

- **LangChain / LangGraph:** Memory is a first-class citizen in LangChain. They provide abstractions like **ConversationBufferMemory**, **ConversationSummaryMemory**, **VectorStoreRetrieverMemory**, etc. In a basic LangChain agent, you attach a memory module so that each time the agent is called, it has the conversation history available. LangChain will handle adding

previous user and AI messages to the prompt automatically (according to whatever window or summarization strategy you choose). With LangGraph, since agents can be long-running and multi-step, they extended this concept to **stateful graph execution**. The LangGraph state can hold arbitrary keys/values as the agent runs; for instance, intermediate results can be stored and accessed by later nodes. They emphasize **long-term persistent memory** too — meaning you could connect a database or vector store to keep information beyond just the ephemeral session. For example, using LangChain's vector stores, an agent could store facts it learned in one conversation and retrieve them in a future session when that user returns. In practice, implementing long-term memory requires hooking into events like end-of-session to save things, and start-of-session to load things, which LangChain supports. Also, memory can be entity-specific (tracking info about a particular customer) or global. The flexibility is there, but the developer must design what to store. LangChain doesn't impose limits beyond token context (if your memory is too large, you might need summarization). Tools in LangChain can also update memory; for instance, a tool's output could be written into the state for later use. Summing up: LangChain/LangGraph gives you building blocks to create both short-term (context window) memory and plug in long-term memory, with the onus on you to choose what fits your needs.

- **OpenAI Agents SDK:** The SDK itself doesn't provide a heavy-duty memory store out-of-the-box, but it's built to allow memory via **conversation history**. When you use the `Runner` to run an agent in a loop, the Runner basically feeds the agent all prior messages each turn (similar to how a chat works). So, short-term memory is handled simply by accumulation of messages until you hit the context limit. For long conversations, you might manually trim or summarize, but the SDK won't do that for you automatically. There's nothing like "built-in vector DB memory" here; however, you could easily integrate one by writing a tool (like a "Recall" tool that does a vector DB lookup) or by pre-pending relevant info to the context each time using your own logic. The expectation is that developers can manage memory however they prefer – some might use the OpenAI API's "system" message to store a running summary, or keep important facts in variables and re-inject them into the prompt. The design is intentionally minimal. In multi-agent scenarios, each agent might have its own memory of the conversation. If agent A handoffs to agent B, you may decide whether B gets A's conversation history or not (the SDK likely leaves that up to your

implementation). Because this is a lighter-weight framework, if enterprise-level memory (like persisting user profiles or conversation logs to a database) is needed, you'll be implementing that on the side. The positive is you're not constrained to any one method – you can integrate with any external storage as needed.

- **Google ADK:** ADK places a strong emphasis on **state and memory**. It distinguishes between **Session state** (short-term memory during a conversation session) and **long-term memory**. The Session uses a `SessionService` that tracks all **Events** (messages, actions, results) happening. This is effectively your conversation history. You can retrieve past events or the current state easily from within the agent or tools. ADK's `State` object allows the agent to record key-value pairs as it goes – think of this like a blackboard where different steps can write and read. For example, if the agent gets the user's name in one step, it might put `state["user_name"] = "Alice"`, and later steps can read that instead of asking again. This is safer than relying purely on the LLM to remember "Alice" – it's explicitly stored. ADK also supports plugging in **Memory Service** for long-term memory across sessions. For instance, you could connect to a vector database or a traditional database to store what the agent learns, and next time the same user comes, the agent can query that memory. Because ADK is meant for customer-facing and persistent agents, it's thought through how to maintain continuity. Another interesting aspect is **Artifact Management** – if an agent generates or receives files, those can be stored and referenced later (for example, an agent creates a report PDF and in a later session can fetch it). In terms of implementation, ADK likely provides APIs to easily save to memory or retrieve from it. The developer can still decide what to memorize (maybe through callbacks or at certain tool outputs they push to memory). The bottom line: ADK offers a robust built-in approach to memory, so you don't have to reinvent it, and it encourages separation of factual state from the pure text conversation to reduce reliance on LLM's recall.

- **AWS Bedrock Agents:** With Bedrock being managed, memory is largely transparent to the developer. AWS Agents automatically maintain a **session for each conversation**. When a user interacts with an agent, Bedrock assigns a session ID and keeps track of all previous prompts, intermediate steps, and results within that session. So if the user asks a follow-up question, the agent's model will get the prior conversation context prepended behind the scenes (up to the model's context length). This means short-term conversational memory is

handled for you. The developer doesn't need to supply the past messages explicitly — the service does it. For long-term or cross-session memory, Bedrock provides a couple of features:

- You can use session attributes or stored variables. For example, you might configure the agent with certain session-level slots (like "user_name") that once filled remain attached to that session's state.
- If you want persistence across sessions or personalization per user, you would likely need to integrate that manually. For instance, the agent could call a Lambda action to fetch user profile info at the start of a session (and then remember it in that session). Bedrock itself doesn't (as of now) have a built-in persistent memory that spans sessions unless you implement it through an external store.
- The Knowledge Base also plays a role: if there's info the agent should "remember," you could put it into the vector index so the agent can recall it via queries. But that's more knowledge retrieval than conversational memory.
- One nice thing with Bedrock sessions is that they can be referred to by an ID, so if you reconnect to the same session ID, the agent will remember context. AWS also likely automatically does things to manage context length (like if the conversation gets too long, they might drop older turns or prompt you to summarize via guardrails). But details might evolve. In short: AWS's memory handling is **automatic for sessions**, and anything beyond that you augment through tools or data in the knowledge base. This simplicity is good if you don't want to deal with memory logic, but if you want advanced memory strategies (like complex summarization or multi-session linking), you will end up writing some code in your Lambdas to achieve it.

F. LLM Support and Deployment Flexibility

- **LangChain / LangGraph:** One of LangChain's biggest selling points is model agnosticism. It has connectors for OpenAI, Anthropic, Cohere, HuggingFace Hub, Azure OpenAI, local models (via libraries like LlamaCPP or text-generation-webui), and more. So you can pretty much use any language model with LangChain, from GPT-4 to a local Llama 2, as long as you have a way to call it. This means you aren't locked into one vendor. If your needs change or you want to run on-prem for data privacy, LangChain will support that. The flipside is you have to handle the setup for those models (e.g., get API keys, possibly run

a local server for open-source models, etc.). LangGraph inherits this flexibility since it builds on LangChain's model interface. Deployment-wise, you can run LangChain/LangGraph code anywhere Python runs: your laptop, a server, a container on Kubernetes, etc. It's not a managed service, so you are responsible for scaling and reliability, but also not tied to any platform. This is great for avoiding vendor lock-in. You could even switch between providers dynamically (some apps choose model based on query, for instance). The only "lock-in" if any is the LangChain library itself, but that's open source and fairly widely adopted. Also, since LangChain is an open framework, it's easier to inspect or modify its behavior if needed (you can see exactly how it constructs prompts or handles function calling, and tweak if necessary).

- **OpenAI Agents SDK:** Despite the OpenAI name, the Agents SDK was built to be model-agnostic as well. It primarily expects a model with a chat completion API and supports the new OpenAI **Responses API**, but you can configure it to use other endpoints. In fact, the SDK uses a layer called **LiteLLM** that can interface with various model providers (OpenAI's docs mention it supports over 100+ models/providers). So you could use it with something like Azure OpenAI or even with Anthropic if you implement the interface. The caveat is, because it's oriented around the OpenAI API schema, it might require that other models behave similarly (e.g., return function-call messages, etc.). Many newer LLM APIs do, so it's not a big issue. The multi-model support is handy if you plan to compare model outputs or want a fallback model if one fails. Deployment of the Agents SDK is flexible: it's just a Python library, so you can run it on your server, or as part of a serverless function, etc. There is no cloud service you must use (though OpenAI likely envisions you using their platform and maybe logging traces there). You can also run it with local models by hooking into an API wrapper that points to your local model. In terms of vendor lock-in: you're not locked into using OpenAI models, but you are somewhat encouraged to use the OpenAI ecosystem (like their evals, their trace tooling). However, those are optional. Since it's MIT licensed (open source), you could fork or modify it if needed. Overall, it's quite flexible, but since OpenAI's business is selling API calls, you can bet it will be optimized to work best with their services.

- **Google ADK:** ADK is open source and model-agnostic by design, but with strong optimization for Google's own models. It has an interface for LLMs (so theoretically you can plug in OpenAI or local models) — in fact, the documentation explicitly says it's not limited to Gemini and can work with other

LLMs. However, certain features shine with Google's Vertex AI: for example, streaming support leverages Google's bidirectional streaming API nicely, safety features tie into Gemini's safety filters, and deploying on Agent Engine is for Google's ecosystem. If you want to run ADK elsewhere, you absolutely can. You can run it on a local machine using an open model or on AWS calling OpenAI API. There is no required cloud component to use ADK — you get all its dev tools and local runtime. For deployment, you have choices: run on your own infrastructure (self-managed in a container or VM) or use **Vertex AI Agent Engine** which is GCP's managed service for hosting agents built with ADK. Agent Engine takes care of scaling, auth, monitoring etc. (similar to Bedrock's service but specifically for ADK agents). If you're on GCP, that's a big plus for production. If you're not, you still have the library. In terms of lock-in, using ADK doesn't force you to use Google Cloud, but it certainly tempts you with convenience if you do. Also, as Google releases their next-gen models (like Gemini), ADK will surely support their unique features (like multimodal input maybe) out-of-the-box. But since ADK is open, the community can extend it to support other models or improvements without waiting for Google.

- **AWS Bedrock Agents:** Bedrock Agents is inherently tied to AWS's platform. When you create an agent on Bedrock, you choose a foundation model from the Bedrock service (e.g., Amazon's Titan, an Anthropic Claude variant, Stable Diffusion for image tasks, etc. — basically whatever Bedrock offers). You can't directly use a model that's not in Bedrock's catalog. If you wanted to use an external model, you'd have to call it via a tool (e.g., have an action that calls an OpenAI API or some custom model endpoint). But the core agent's own reasoning model has to be one of the Bedrock hosted models. This is a limitation if you have a specialized model or want the latest GPT-4 — unless they're available on Bedrock (OpenAI models currently are not on Bedrock). AWS is likely to add more models over time, possibly including open source ones they host, but keep this in mind. Deployment wise, Bedrock Agents run on AWS-managed infra, so you don't worry about scaling or performance (to a degree; you might need to manage concurrency by provisioning throughput units or similar). It's great if you are already using AWS — it integrates with CloudWatch for logging, IAM for permissions, etc. But it's not portable: you can't run a Bedrock agent outside AWS. If multi-cloud or on-prem is a requirement, Bedrock Agents would be a poor fit. Essentially, it trades flexibility for ease-of-use in an all-in-one AWS environment. For some companies, that's fine, especially if

they're all-in on AWS and need a fast solution with compliance and security features ready. For others who want freedom to choose models and environment, it's restrictive.

G. Evaluation and Observability

Building an agent is one thing — knowing how well it works (and improving it) is another. All these frameworks recognize the need for **evaluating agent performance** and providing **observability** into the agent's decisions.

- **LangChain / LangGraph (with LangSmith):** LangChain has developed **LangSmith**, a platform for testing and monitoring LLM applications. With LangSmith, you can log each run of your agent (each "trace") and then inspect it in a UI — you can see each step the agent took, which tool it called with what inputs, what the LLM was thinking at each point (the intermediate prompts and outputs). This is invaluable for debugging when the agent does something weird. LangGraph is designed to work seamlessly with LangSmith: it can capture the execution graph and state and send it to the trace viewer. On the evaluation side, LangSmith allows both **automated tests** and **human feedback**. For example, you can define evaluation criteria (was the answer correct given some ground truth? Did the agent avoid certain forbidden words?). LangSmith has a library of evaluators (some use LLMs to judge outputs, some use regex or heuristic checks). You can batch-run these evaluations on test data to get metrics like accuracy, or run them continuously in production to catch regressions. LangChain also introduced the idea of **guardrails via evaluators** — basically using the same eval functions to act as runtime guards (e.g., an evaluator that checks for sensitive info could be run on each output as a guardrail and decide to refuse output if triggered). Because LangChain is open, there are community recipes for eval too (like using GPT-4 to rate the quality of a response). The developers can integrate these evals into their CI/CD — e.g., before deploying a new version of an agent, run a suite of eval queries to see if it's improved or worsened. Observability also includes metrics like token usage per step, latency per tool call, etc., which LangSmith can show. Summing up: LangChain's approach is **evaluation-driven development** — treat the agent like code that needs tests. The combo of LangGraph + LangSmith gives a powerful way to test, tune, and watch agents in action.

- **OpenAI Agents SDK:** Since OpenAI provides both the model and the SDK, they've started offering integrated tools for observing agent behavior. When

using the Agents SDK, you can enable **tracing** — this will capture the sequence of calls and possibly send them to OpenAI's servers where you can view them. OpenAI had earlier a "developer console" (in their platform website) for viewing request logs and such; it's likely extended to show agent traces as well, especially since the Agents SDK is a focus. Additionally, OpenAI released an **OpenAI Evals** library (open source) for evaluating model outputs. The Agents SDK presumably can interface with that: you could, for instance, use OpenAI Evals to run a benchmark on your agent (maybe by simulating user conversations and then checking the answers). The mention of "guardrail metrics surfaced alongside traces" suggests that when guardrails (which are part of the SDK) run, their outcomes (like whether a tripwire was triggered, or how much "risk" was detected) are logged. So in the trace UI, you might see not only the model's thoughts and tool calls, but also notes like "Output guardrail flagged disallowed content — agent response halted." This level of detail helps pinpoint where safety rules kicked in. The OpenAI evals library can be used to compute things like success rate on tasks or compare outputs between model versions. While not as full-featured as LangSmith (which is a dedicated product), OpenAI's approach benefits from being more **built-in** if you're already on their platform — you don't have to sign up for another service, just use their dev tools. One potential downside is it might not capture everything if you're using non-OpenAI parts (e.g., if you integrated a non-OpenAI model, not sure if their UI supports that scenario fully). But overall, they aim to provide a **one-stop view** of your agent in action and how it fares against tests or metrics.

- **Google ADK:** Google's ADK, especially when used with Vertex AI, puts a lot of focus on telemetry and monitoring. If you deploy an agent on Vertex AI's Agent Engine, you get access to logs for each step the agent takes. The **Vertex AI console** likely shows each tool invocation, the inputs/outputs, model latency, etc. Google's docs also mention **safety attribute scoring** — meaning for each response, the system might output scores for categories like toxicity, violence, etc., which you can monitor (these come from the model's safety filters). ADK's own Developer UI (for local dev) lets you step through an agent's run event by event, so you can visually see the chain-of-thought (this is similar to trace viewers). On the evaluation front, ADK includes a built-in **evaluation module**. They highlight being able to evaluate "both final response quality and step-by-step execution" against test cases. This implies you can create test conversations (with expected actions and results at each step) and ADK can run the agent and

compare what it did versus what it should have done. That's great for regression tests — e.g., you want to ensure your agent always calls the "ValidateUser" tool before accessing account info; you can have an eval that checks a sample interaction to see if that tool was called. The ADK CLI might provide commands to run these evals and output metrics. Also, because ADK is open source, you can integrate it with other observability tools. In fact, their docs reference integrating with external observability platforms like Arize or Phoenix for analyzing model outputs. As for continuous improvement, you could imagine hooking ADK agents with feedback loops (like logging incorrect answers and using them to fine-tune models, though that's outside ADK's direct scope). Google's forthcoming additions (as hinted by "upcoming eval recipes") likely include more pre-built evaluation scenarios or templates specifically for common agent tasks. In summary, ADK offers **comprehensive telemetry** and encourages systematic evaluation from development to production.

- **AWS Bedrock Agents:** AWS knows that enterprise users need to monitor everything. Bedrock provides a **trace viewer** in the AWS Console where you can see each turn of a conversation broken down. It will show events like "User said X", "Agent chose Action Y with these params", "Result from Action Y was Z", "Agent responded with answer A". This is crucial for debugging when the agent fails or does something unexpected. The trace also includes any errors (like if a tool invocation failed or the model output was invalid JSON, etc.). Additionally, AWS surfaces metrics like latency (how long each step took, how long the model inference took, etc.) and token usage per request — helpful for cost monitoring. On evaluation: while AWS doesn't provide an open library of eval prompts (since it's a service, not a dev library), they do allow you to **test guardrails and policies in the console**. There is mention of a "guardrail test harness" which likely means you can simulate inputs to see if your guardrails catch them as expected, and maybe run a suite of such tests. For example, if you have a profanity filter guardrail, you can input some profane text in a test interface and confirm the agent properly refuses. In terms of performance eval (like accuracy), AWS might expect you to use your own methods (maybe SageMaker Clarify or other ML evaluation tools) — it's not built into Bedrock Agents as a feature to run custom Q&A tests out-of-the-box. But you could certainly script something using the Bedrock API to run through a list of test queries and collect answers. Where AWS excels is **operational monitoring**: CloudWatch can capture logs of agent interactions, and you can set alarms (for instance, if an agent's error rate goes

above X, trigger an alert). They also provide audit logs for compliance (who invoked what agent, did it trigger a guardrail, etc.). One more thing: **hallucination detection** isn't explicitly solved by Bedrock, but you could implement a guardrail or use the knowledge base to double-check facts. They do provide guidelines to detect if the agent's answer came from the knowledge base or not (since knowledge base queries are part of the trace). In essence, AWS gives you the tools to monitor and some testing harness for safety, but less so an automated eval pipeline for quality — for that you'd integrate with external QA processes.

Across all these frameworks, certain **common metrics** are important to track:

- **Task success rate:** e.g., did the agent achieve what the user asked? (This often requires some definition of success per task — like unit tests.)
- **Tool usage accuracy:** did the agent call the right tool with correct parameters? How often does it call tools unnecessarily or miss calling when it should?
- **Hallucination frequency:** how often does the agent output something incorrect or unsupported by available data?
- **Safety compliance:** how often does it produce disallowed content or fail to follow policy (and do guardrails catch it when it does)?
- **Latency:** both user-perceived (total time to answer) and breakdown by step, to identify bottlenecks (model too slow? certain tool slow? too many steps?).
- **Cost:** especially for API-based models, tracking tokens used per conversation is essential to optimize usage.

Having good observability and eval tools makes it feasible to tune these metrics and compare different agent configurations or even different frameworks head-to-head.

H. Guardrails and Safety Features

Deploying agents in the real world requires robust guardrails to prevent misuse, avoid sensitive leaks, and enforce business rules. Each framework approaches this with a mix of technical and policy tools.

- **LangChain (LangSmith Guardrails):** LangChain itself didn't originally have a built-in guardrail system (it relied on the model and developer to be careful), but

with LangSmith and community contributions, it's gaining ground on safety. One approach is using **output evaluators as guardrails**: for example, an evaluator that checks if the output contains a phone number (PII) could be run as a guard step and if it returns a high score (meaning "likely contains PII"), the agent could refuse to output or redact it. LangSmith allows defining these evaluators fairly flexibly (some are simple keyword checks, others can use an LLM to classify content). There are also community recipes integrating the OpenAI "Guardrails" package (formerly known as Railcar) with LangChain, which can enforce response templates or stop the agent if certain regex matches are found. LangChain being open means you could always insert custom logic in between steps. For instance, you could wrap a tool call such that before executing you verify the inputs (ensuring the model isn't trying an SQL injection in a SQL query tool, for example). The emerging pattern is "**guardrails-as-code**" in LangChain: you write Python functions to intercept or validate things. It's not as off-the-shelf as some managed solutions, but it's powerful and customizable. The LangChain team is actively discussing safety, so expect more built-in hooks or easier configuration for common guardrails.

- **OpenAI Agents SDK:** OpenAI baked guardrails into the SDK as first-class objects. When you create an agent, you can attach **Guardrail** objects to it. The SDK defines two main types:

- **Input Guardrails:** run on user input before it reaches the main agent logic.

- **Output Guardrails:** run on the agent's final answer before it's sent back to the user.

- Guardrails are essentially functions (or even small sub-agents) that examine the content and output a result indicating if everything is okay or if a violation (a "tripwire") is triggered. For example, an input guardrail might check "is the user asking for disallowed content?" by running a moderation model. If yes, it raises an exception and the agent will not proceed to answer that query. Similarly, an output guardrail might scan the answer for forbidden content; if it finds some, it can halt the response or replace it with a safe message. The design is such that guardrails run **optimistically in parallel** with the agent — meaning they don't add much latency. The agent doesn't wait for guardrail approval at each step; rather, the guardrail can cut in asynchronously and stop the process if needed (like a circuit breaker). This is important because you don't want safety checks doubling your response time. OpenAI's guardrails can also do things like

sanitizing outputs (e.g., mask certain data) or enforcing structure. They mention "optimistic execution with rollback" — the idea is the agent can generate a response quickly, and if a guardrail flags it after the fact, you can throw away that response and instead return an error or a safe completion. The code-centric nature means developers can implement custom guardrails for domain-specific policies, beyond just OpenAI's content rules. Additionally, OpenAI's models themselves have system-level content moderation (especially if you use their built-in filters), but guardrails let you add your own logic on top of that.

- **Google ADK:** ADK takes a multi-layered approach that aligns with Google's Vertex AI safety best practices:

- **Deterministic guardrails via tool design:** As we saw in ADK's docs, one philosophy is to limit what an agent *can* do by construction. If you only give it safe tools that themselves enforce rules (like the database query tool example that only allows certain tables or only SELECTs), you eliminate whole classes of bad behavior (like dropping tables).

- **Callbacks for validation:** ADK allows you to inject callbacks before or after model calls and tool calls. This means you can, for instance, intercept a model's proposed tool arguments and verify them. If they violate some policy (say the agent is trying to call a "SendEmail" tool to an address not allowed), you can stop it or modify it. Similarly, after the model produces an answer, you could post-process it to mask anything sensitive. This is coding work, but very flexible.

- **Built-in safety filters:** If you use Google's Gemini or PaLM models via Vertex, you get their content filter working. That might automatically refuse to produce certain content or tag it with safety scores. ADK can use those signals. For example, if the model returns a safety flag, ADK could catch that and turn it into a safer response or trigger a different agent behavior.

- **"Orchestration controls":** Google has hinted at controls that restrict the agent's reasoning paths. This might mean you can configure the agent to not loop more than N times, or not use certain tools in combination, etc. It could also refer to structuring multi-agent interactions such that an agent can only communicate in predefined ways (preventing prompting attacks between agents).

- **Gemini as safety agent:** An interesting idea in ADK docs is using a model as a guardrail — e.g., you could have a second instance of the model (maybe a smaller/cheaper one) that judges the main agent's output for safety or

compliance. This parallels OpenAI's guardrail agent idea. Google's infra could allow that in parallel, given their scaling.

- In sum, ADK expects developers to be proactive: design safe tools, use the provided content filters, add callbacks for checks, and restrict agent capabilities thoughtfully. It gives all the hooks needed to do so, but it does require careful configuration. The advantage is **auditability and control** — you can point to the code or settings that implement your safety policy. The disadvantage is more effort, whereas a managed service might do more out-of-the-box.

- **AWS Bedrock Agents:** Bedrock makes safety a configurable part of the agent setup. When creating an agent, you can attach **Guardrails** from a catalog:

- AWS offers pre-built guardrails like "Block profanity", "Block hate speech", "Mask PII", etc. These are likely powered by Amazon's Comprehend or custom models under the hood that detect those categories.

- You can also create custom guardrails. For example, you might provide a regex or a list of forbidden words (maybe competitor names, or certain sensitive terms) and Bedrock's guardrail will ensure if those appear in output, they get removed or cause the answer to be refused.

- Guardrails in Bedrock can apply to inputs, outputs, or even tool requests. The GuardrailTrace we saw in the docs indicates the system runs guardrails as part of the agent's reasoning steps. If a guardrail triggers, the agent might stop or adjust its response. For instance, if the user asks something disallowed, the agent might respond with a canned "I'm sorry, I can't help with that request."

- Since Bedrock uses multiple models, there might also be model-specific safety in play (e.g., Anthropic Claude has its constitutional AI principles that make it refuse certain things).

- The **policy aspect** is central: enterprises can define their policy (no sexual content, no personal data leakage, etc.) and implement that with a combination of AWS's provided guardrail types. The benefit is you don't have to write code for these common policies — just toggle them on.

- On the flip side, because it's managed, you have to trust that these guardrails are doing what they say, and you might not get to customize beyond what AWS allows (though they do let you create custom ones to an extent).

- A nice feature is versioning: you can have multiple guardrail sets and update them, and monitor how often they trigger via the console. This is good for governance and auditing.

In comparing these:

- LangChain and ADK give you ultimate flexibility (code-level guardrails, integrate any logic you want).
- OpenAI SDK and AWS Bedrock give you convenience (some built-in guardrail functionality that you can just enable or use templates for).
- OpenAI's approach is still developer-centric (you write the guardrail function), whereas AWS is more configuration-centric (flip a switch or upload a list).
- For highly regulated environments, one might even use multiple layers: for example, with ADK or LangChain, you might use an OpenAI or AWS API for content moderation as part of your guardrails (so mixing approaches).

When implementing guardrails, there's always a trade-off:

- **Granularity & control** vs **ease-of-use**: writing custom code (granular) vs using pre-made filters (easy).
- **False positives/negatives**: A too-strict guard might block acceptable content or let something slip; these frameworks usually let you tune thresholds (like AWS configurable categories or OpenAI's guardrail logic can be made more or less sensitive).
- **Auditability**: Enterprises often need to know "what rules are in place and how do we prove it." Code-based guardrails are explicit, but harder for non-devs to understand. Config-based are easier to document (like "we use AWS moderate with settings X").
- **Performance**: more guardrails and complex ones can slow things down or cost extra (if they call additional models). So one must balance how many layers to check on each request.

All four frameworks recognize that without good guardrails, you can't trust an agent in production. As a developer or team adopting one, it's important to review the

safety features available and possibly combine multiple mechanisms to cover all your bases.

Architecture and Design Trends

Stepping back, let's look at the bigger picture of how AI agent architectures are evolving. Across these frameworks, a few **common trends** emerge:

- **Modularity and Pluggability:** There's a clear move towards highly modular agents. Tools are plug-and-play modules. Some frameworks treat even sub-agents or chains as modules (Google ADK allowing agents as tools, LangChain composing agents, etc.). This means you can build a complex system by assembling simpler pieces. It also means parts can be swapped out — for example, if a better code execution tool comes along, you can plug it in. Modularity extends to memory (pluggable memory backends), LLMs (pluggable model providers), and even guardrails (pluggable safety checks). This design allows developers to start simple and progressively enhance the agent's capabilities or swap components for better ones.

- **Transparent Reasoning and Tracing:** In the early days, chain-of-thought was hidden in the LLM's internal reasoning, and developers only saw final answers or had to parse verbose text reasoning. Now, frameworks capture the CoT explicitly and show it via UIs or logs. Every step, thought, and action is traceable. This not only helps developers debug, but also builds trust with stakeholders (you can show, "Here's how the AI arrived at this answer."). We're seeing that with LangSmith's traces, OpenAI's trace viewer, Google's event logs, AWS's trace JSON. Expect more polished UIs and maybe even real-time dashboards as these mature. Some are even introducing **visualizations** — like a flowchart view of an agent's thought process or timeline diagrams (think of a sequence diagram of interactions). This transparency is crucial for moving from prototype to production, where you need to explain and audit agent behavior.

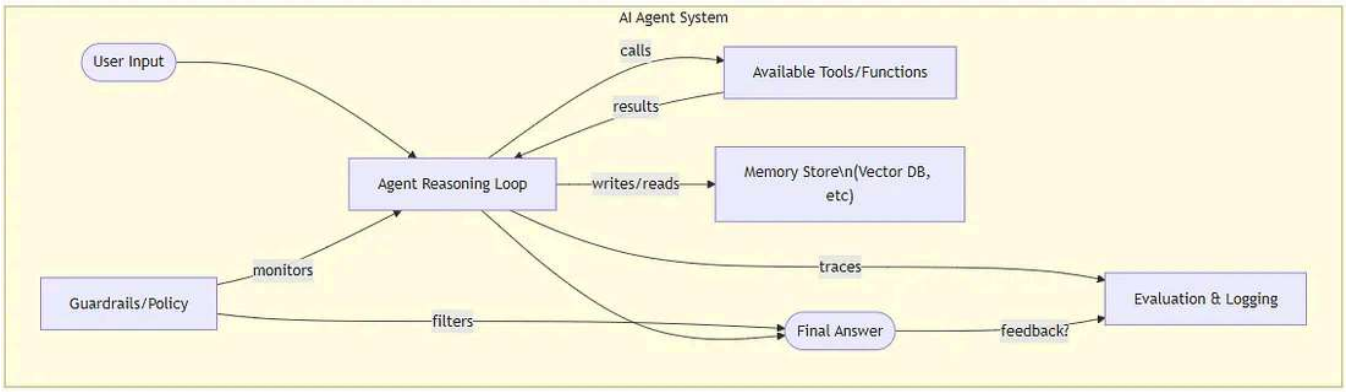
- **Evaluation-Driven Development (EDD):** Similar to how Test-Driven Development transformed software engineering, we're seeing a push for systematically evaluating AI systems. Instead of "hope it works," teams are writing evaluation suites: sets of example tasks or conversations where the expected outcome is known, and running them regularly. This is becoming necessary because LLM behaviors can change with model updates or prompt tweaks. By automating evals, you catch when a change breaks something you

care about. All frameworks now have some support for evaluation — be it integrated libraries or guidance on how to build one. We'll likely see specialized **evaluation dashboards** that gather an agent's performance across versions and scenarios, analogous to test coverage reports in traditional software. In addition, **user feedback loops** and continuous learning (collecting thumbs-up/down from users and using that to improve agents) are being productized (OpenAI's feedback API, etc.). So an effective agent dev pipeline in 2025 looks like: define metrics & evals -> build agent -> run evals -> deploy -> collect prod data -> refine -> repeat.

- **Guardrails-as-Code and Policy Engines:** There's momentum toward making safety and policy enforcement more declarative and code-driven. Rather than trusting a single magic "moderation" filter, developers are layering multiple checks and writing custom logic. This is spawning mini "policy engines" in each framework. For example, OpenAI's guardrails functions, Google's callback system, or even external libraries like Guardrail (formerly Railcar) which offer a DSL to specify what the model can/can't output. In time, we might see standardized policy definition languages for AI (some early attempts exist). The trend is that companies want to **own their safety logic** — not just rely on the base model — and frameworks are facilitating that. Also, regulator pressure (like the EU AI Act) will likely require showing these guardrails in action, so having them as code/config makes compliance easier.

- **Managed Runtimes & Cloud Agent Services:** Not every team wants to manage infrastructure for agents. AWS Bedrock Agents and Vertex AI's Agent Engine indicate a move towards cloud providers offering "Agent Hosting" as a service. These handle scaling, high availability, integration with logging/monitoring, and even optimization (like automatically caching tool results or adjusting prompts for efficiency). While open-source frameworks give flexibility, using a managed service can speed up time-to-market, especially for companies that trust a specific cloud. We can expect improved offerings here: perhaps easier deployment from open frameworks to these services (e.g., maybe one day you can export a LangChain agent to run on Bedrock, bridging the gap). Moreover, these services might evolve to have *app marketplaces* for agents or more turnkey solutions for common agent types (like "SQL query assistant" template on AWS that you just configure). The design trend is separating the agent definition (what it does) from the execution engine (how it runs at scale).

To visualize a modern agent's architecture, consider the following diagram of an agent's lifecycle and components:



General AI Agent architecture

In this schematic:

- The **Agent Core** is the LLM-driven loop that may iterate through thought/action steps.
- It has access to a set of **Tools** and possibly external APIs.
- It can query or update a **Memory Store** (short-term context or long-term knowledge).
- A **Guardrails module** keeps an eye on the process and final output, intervening if needed (either by halting or modifying outputs that violate rules).
- An **Evaluation module** logs what's happening (traces) and can run tests or metrics calculations, feeding that information to developers or even back into improving the agent.

All four frameworks implement some version of this diagram. The differences are in the specifics of each component and how much is done for you vs. by you.

We're also seeing experimentation with **sub-agents and plug-in ecosystems** (like the tools plugin model from earlier GPT Plugin systems). The architecture is moving towards something like a mini operating system: the LLM is the CPU/brain, the tools are apps it can run, memory is like disk/RAM, guardrails are the OS security, and evaluations/telemetry are the system monitoring. It's an exciting convergence of software engineering and AI research.

Use-Case Fit and Decision Guide

Given the capabilities of these frameworks, how do you choose the right one for a particular project? Let's break it down by scenarios and key factors:

1. Internal Automation (scripts, DevOps, internal tools): Suppose you want an agent to automate internal workflows, like pulling data from different databases and generating a report, or handling DevOps queries by running commands. If your environment is sensitive (maybe can't send data to external services) and you want full control, **LangChain/LangGraph** might be ideal due to on-prem deployment and wide tool integration (you can connect to all your internal systems). If you're an AWS-heavy company and the data can stay in AWS, using **Bedrock Agents** could be quick — you'd wire up Lambdas to do those internal tasks. OpenAI's SDK could work but since it doesn't come with many tools, you'd be writing a lot of functions to interface with internal systems (not a big issue if you're comfortable coding that). ADK is a contender if you want to structure it nicely and maybe use Google Cloud for some reason — but for internal automation, the overhead of ADK might not be needed unless you want its eval and multi-agent structure.

2. Customer Support Bot (answering user queries with database/API lookups): This often involves retrieval (knowledge base) + some tool usage (maybe create a support ticket, lookup an order, send an email). **AWS Bedrock Agents** are attractive here if you already host your knowledge base in AWS (or can import it) and want an easy way to integrate with your CRM via Lambdas. It gives a managed, secure setup (important for customer data). **OpenAI's Agents SDK** could be used, especially if you fine-tune an OpenAI model on support data and use function calling for actions — you'd then host it yourself, likely with LangChain helping for retrieval. But you'd need to implement retrieval and any API calls (LangChain can assist for RAG, or OpenAI's file search tool if that suits your data). **LangChain** is very popular for chatbot scenarios with Retrieval-Augmented Generation: you'd use LangChain's retrieval tools + an agent for actions like checking order status. The advantage is maximum flexibility to tune how it works. **Google ADK** could be a good fit if you foresee a complex workflow in support (like multi-step troubleshooting flows, or handing off to different specialized agents for different product lines). Also consider the **nature of your domain**: if it's highly sensitive or regulated (healthcare, finance), guardrails and audit logs are crucial — Bedrock and ADK (on Vertex AI) might have an edge since they integrate with cloud logging and have built-in content filters for safety. LangChain and OpenAI SDK can be made compliant but require more manual effort (like ensuring all logs are saved, data is encrypted, etc., which cloud solutions handle by default).

3. Knowledge-Driven Chatbot (documentation Q&A with retrieval): This is the scenario of a doc chatbot that finds info and answers questions. **LangChain** basically pioneered RAG, so using it with an Agent can work well. You might not even need a full agent — often a simpler retrieval + answer chain suffices (deterministic workflow: user question -> search docs -> get answer -> respond). A deterministic approach might be preferable here since you want reliability (less chance of hallucination if you always ground the answer in retrieved text). However, an agent could add abilities like "if the docs don't have it, use a web search tool" etc. **Google ADK** could shine if you want advanced features like multi-step reasoning when answering (e.g., break the question into sub-questions, or verify answers from multiple sources — ADK's sequential or parallel agents could help coordinate that). Also, if you plan to **evaluate answer quality** extensively, ADK's eval tools would help. **OpenAI SDK** would rely on function calling for retrieval (which is doable: one function that queries the docs). It can produce good results if using GPT-4 with a vector store function. **AWS Bedrock** again gives a ready-made retrieval setup (their knowledge base feature means you can just upload documents and the agent will automatically use them). For a quick deployment with existing docs, Bedrock might be the fastest path, provided your docs can be hosted or indexed in AWS. If your data is highly proprietary and cannot leave your environment, then an open solution (LangChain/ADK with self-hosted model) might be necessary.

4. Multi-System Workflow (agent orchestrating across multiple APIs/DBs): If you need an agent to handle, say, an end-to-end business process (e.g., process an insurance claim: read claim details, verify policy from DB, call an external verification API, then update a record and notify someone), you likely need a carefully controlled sequence plus error handling. **LangGraph** would let you explicitly model this process, ensuring each step happens in order and handling branches if something goes wrong (LLM can be used in parts where needed, like understanding a text description in the claim). **Google ADK** with a SequentialAgent would also fit, giving structure to the workflow and the ability to incorporate LLM steps for decisions (like a step where an LLM agent checks if a claim is potentially fraudulent). ADK's multi-agent ability could separate roles (one agent gathers all needed info, then hands off to another agent that makes a decision, etc.). **AWS Bedrock** could do it, but it might be risky to rely on the model to *always* pick the right sequence for such a critical process. You can influence it with prompt design, but it's not as guaranteed as an explicit workflow — so you might combine Bedrock's agent with some external control (like break the task into smaller Bedrock agent

calls in code). **OpenAI's SDK** would likely require you to script some flow control in Python if certain steps must happen in order (the agent itself might not always pick the correct next tool without guidance). So for complex workflows with clear structure and little room for improvisation, frameworks that allow **explicit flow control** (LangChain's workflows or ADK's sequential flows) are safer bets.

5. Regulated/PII-Heavy Domains (finance, healthcare, etc.): Safety, privacy, and audit logs top the list here. **AWS Bedrock Agents** and **Google's Vertex AI (with ADK)** are appealing because they offer data encryption, access controls, and compliance certifications (HIPAA, SOC2, etc., depending on the service), plus fine-grained logging. They have robust guardrails for content (e.g., blocking PII by default, which is important if the agent could accidentally reveal something). If your company policy forbids sending data to third-party services, an open-source solution like LangChain with a self-hosted model might be the only option — that gives you full control, though you then must implement a lot of safety measures yourself. In a regulated setup, **deterministic workflows often beat dynamic reasoning** for core tasks. For example, a bank might not be comfortable with an agent that "wings it" on how to execute a transaction. They would prefer a fixed, audited sequence where the LLM is perhaps only used to parse instructions or fill in forms, but not decide whether to execute a transfer or not. Agents can still play a role in providing flexibility in conversation or data extraction, but likely under more constrained settings (maybe the agent suggests actions which a human or a deterministic process then approves and executes). Between our frameworks, **LangChain/LangGraph** can run entirely in a secure environment (no external calls except maybe the model, which could even be on-prem or an approved cloud). **ADK** can as well (with an open-source model or via Vertex AI in a private cloud setup). **OpenAI's SDK** might be challenging if data residency or isolation is required, unless you use something like Azure OpenAI (which can be in a private VNet) — the SDK can connect to Azure endpoints too, so that's an option. In any case, heavy emphasis on **guardrails** and **logging** is needed: ADK and Bedrock provide more built-in, whereas with LangChain/OpenAI you'd integrate things like OpenAI's moderation API, logging to your own databases, etc.

6. Rapid Prototyping vs Long-Term Maintenance: If you want something up quickly as a proof-of-concept, **OpenAI Agents SDK** or **AWS Bedrock** are winners. You can get an agent running with minimal code or config, and both have user-friendly interfaces (OpenAI's simple API, AWS's console). However, consider the long-term:

- If you foresee needing to switch LLM providers or run on custom infrastructure later (for cost, data locality, etc.), starting with an open framework (LangChain or ADK) might save future rework. It's easier to move from open to managed (you can often wrap an existing agent into a managed service later) than vice versa.

- If you want a large community and examples to draw from, LangChain is unparalleled right now. Many developers, blog posts, and Q&As exist for it, which can accelerate development and troubleshooting.

- If your project will involve complex interactions or expansions (like today it's one agent, tomorrow it's a network of agents collaborating), ADK's structured approach might pay off by making it easier to scale up the complexity in a controlled way.

- **Team expertise** matters: some teams have more ML/LLM know-how and prefer the control of open frameworks; others might be more application-focused and prefer letting the cloud handle the ML ops parts.

- **Vendor strategy**: Sometimes the decision is influenced by company strategy. For instance, if your company has a strong partnership or credits with a certain cloud, you might lean toward that ecosystem's solution (e.g., AWS shops will look at Bedrock first; a Google Cloud customer might try ADK/Vertex).

Often, teams prototype with one and then migrate to another as requirements clarify. For example, you might start with OpenAI's SDK for a quick demo (leveraging GPT-4's capabilities easily), but then rebuild with LangChain for production to have more control and avoid lock-in. Or start with LangChain to deeply understand the agent's needs, then switch to Bedrock for production to offload infrastructure management. Being aware of the trade-offs at the outset helps plan for these transitions.

To sum up the general guidance:

- Use a **deterministic workflow** approach when reliability and predictability are paramount, or when the task sequence is well-understood. This can be achieved with visual flows in LangGraph, sequential agents in ADK, or simply carefully orchestrated code with the OpenAI SDK or AWS's configured prompts.

- Use a **dynamic agent** approach when the task is exploratory or complex enough that pre-defining a single path is impractical. Dynamic reasoning shines in creative problem solving, open Q&A, or situations where the agent might discover new sub-tasks as it goes.

- Mind the **ecosystem fit**: align with your existing tech stack where possible (it will make integration and deployment smoother, e.g., using Bedrock if you already use many AWS services, or ADK if you are heavy on Google Cloud and want to integrate with Google Workspace apps).

- Keep an eye on **costs**: an agent that calls a large LLM many times or uses expensive tools like web search can rack up costs. Using local or open-source models via LangChain/ADK could cut costs, but requires infrastructure. A managed service might optimize model usage under the hood (for instance, Bedrock might stream results to reduce token usage on long outputs, etc.). Always test on a smaller scale and estimate costs for your usage patterns.

Conclusion

All four frameworks — LangChain, OpenAI's Agents SDK, Google's ADK, and AWS Bedrock — share the common goal of enabling **powerful LLM-based agents**, and they're converging on similar core features:

- The ability for the agent to reason about tasks and break them into tool calls.

- A set of tools or functions the agent can use to act on the world or fetch information.

- Mechanisms to maintain memory and state so the agent isn't short-sighted.

- Ways to inject instructions and control the agent's behavior.

- Features to evaluate how the agent is doing and to observe its internal decisions.

- Guardrails to ensure the agent stays within acceptable boundaries.

Where they differ is in the **philosophy and developer experience**:

- **LangChain/LangGraph** gives you maximal flexibility and a rich open ecosystem, ideal for those who want an open-source solution with lots of community support and no cloud dependencies. It offers transparency and fine control (you

can inspect and tweak every part of the chain), but you're responsible for managing it in production.

- **OpenAI's Agents SDK** focuses on simplicity and quick development, with just enough structure to build an agent without much overhead. It's great if you want a lightweight solution that leverages OpenAI's powerful models and you're okay with a code-centric approach. It integrates nicely with OpenAI's platform features but doesn't lock you into their models.
- **Google ADK** provides a comprehensive, structured framework that might appeal to developers who want to apply software engineering rigor to AI agents. It's well-suited for complex, multi-agent scenarios or applications where you need that built-in eval, debugging, and safety from the ground up. It shines in a Google Cloud environment but can be used elsewhere too.
- **AWS Bedrock Agents** offers a high-level, fully managed experience. If your priority is ease of integration and operation (and you're already on AWS), Bedrock lets you add agent capabilities with minimal ML knowledge. You trade some flexibility and are tied into AWS's way of doing things, but you gain scalability, security, and a faster path to production in many cases.

In terms of **similarities**, these frameworks are clearly influencing each other and likely will continue to. OpenAI's introduction of guardrails and multi-agent handoffs echoes features in LangChain and ADK. LangChain is adding more structured workflow capabilities (via LangGraph) which traditional workflow engines and Google's kit already emphasize. We can expect the gap between them to narrow over time, as each adopts the best ideas from the others. This is good news for developers: it means whichever you choose, you're likely not missing out on any fundamental capability, and skills learned on one may transfer to the others.

A recurring theme is the importance of **evaluation and guardrails**. No matter how fancy the agent, we need ways to measure its quality and keep it in check. The frameworks that provide convenient eval harnesses and easy-to-apply safety rules can save a ton of time and prevent costly mistakes. So, one of the best things you can do is invest early in building an eval set for your agent and configuring guardrails for known risks — all these frameworks allow it, and it will make your solution more robust.

Finally, it's worth noting the **rapid evolution** in this space. The "best" practices today might be outdated in a few months. New tools (like multimodal capabilities, more efficient models) will emerge and all these frameworks will adapt. We're likely to see convergence toward richer guardrail APIs (maybe even standardized across platforms), unified evaluation dashboards, and deeper interoperability between open-source and cloud (perhaps a common agent specification format or plugin standard). Developers should expect some churn and stay agile — today's LangChain might integrate with tomorrow's OpenAI feature and vice versa.

The good news is that the core ideas aren't too different, so learning one framework gives you insight into all. As you build AI agents, focus on the foundational concepts (reasoning loop, tools, memory, prompting, testing, safety) — those will serve you well regardless of the tool. With that mindset, you can confidently pick a framework (or mix of frameworks) that fits your needs, and deliver AI agents that are both **capable** and **trustworthy**.

Resources

- **LangChain LangGraph Documentation —** *LangChain's official docs on building workflows and stateful agents with LangGraph.* (https://langchain-ai.github.io/langgraph/)
- **OpenAI Agents SDK Documentation —** *OpenAI's official guide for the Agents SDK (Python), including usage of tools, guardrails, and multi-agent orchestration.* (https://openai.github.io/openai-agents-python/)
- **Google Agent Development Kit (ADK) Docs —** *Google's official ADK documentation covering agent types, tools, memory, safety, and deployment on Google Cloud.* (https://google.github.io/adk-docs/)
- **AWS Bedrock Agents User Guide —** *Amazon's official documentation for Bedrock Agents, explaining how to configure actions, knowledge bases, prompts, and guardrails.* (https://docs.aws.amazon.com/bedrock/latest/userguide/agents.html)
- **OpenAI Blog — "New Tools for Building Agents" —** OpenAI's announcement (March 2025) introducing the Agents SDK, function calling tools, and their vision for agentic AI. (https://openai.com/new-tools-for-building-agents)

- **Google Cloud Blog — "Making it easy to build multi-agent applications" —** Introduction to Google's ADK and its use cases by Google's developer team. (https://developers.googleblog.com/2025/03/agent-development-kit-multi-agent-applications.html)
- **LangChain Blog — "Understanding LangGraph for LLM Workflows" —** In-depth article on how LangGraph extends LangChain for complex agent workflows, with examples. (https://blog.langchain.com/langgraph-workflows/)
- **AWS Machine Learning Blog — "Building AI Agents with Amazon Bedrock" —** AWS blog post with examples of creating a Bedrock Agent and best practices for action design and guardrails. (https://aws.amazon.com/blogs/machine-learning/building-ai-agents-with-bedrock/)

Thanks for reading!

If you found this article helpful, also check out my book *LangChain in Action*, available to read for free on my publisher's website. Follow me on Medium and Subscribe to get notified for more upcoming articles.

AI | Ai Agent | Ai Agent Development | OpenAI | Langgraph

Following

Written by Roberto Infante

171 followers · 44 following

Quantitative developer, passionate about latest tech. Author of "Building Ethereum Dapps" and "AI Agents and Applications" (Manning Publications).