Startup guide to AI agents

Technical deep-dive into code-first development.

Google for Startups

Table of contents

Introduction

Choose your development path

Chapter 1 Lay the foundations

Chapter 2 Build with ADK and Antigravity

Chapter 3 Manage memory and context

Chapter 4 Use protocols to support multi-agent systems

Chapter 5 Deploy to production

Chapter 6 Manage in production with AgentOps

Conclusion

Introduction

As the agentic era evolves, the opportunities for AI startups are growing exponentially.

The leaders are moving away from simple, single-prompt solutions toward autonomous agentic systems capable of executing complex workflows. Yet this exciting time is not without its challenges.

To build multi-agent systems that can scale to meet the demands of today's users, startups must keep pace with the latest protocols, best practices, models, and more.

This means making the switch from prompt engineering to context engineering—replacing long, fragile prompt strings with systematic working-memory management. It means switching from monolithic, single-agent loops to modular, specialized sub-agents that can communicate and collaborate via standardized protocols. And it means evolving from single-turn reactive chats to long-running, autonomous background sub-agents running on scheduled cron jobs or event triggers.

All this, while grappling with ongoing cost constraints. AI efficiency—or "tokenomics" as it's often called—remains top of mind, with startups now building complex multi-model routing layers to run simple tasks on lightweight open models and using semantic context caching to prevent runaway token bloat.

Despite all of these shifts, the best startups continue to dream big, build fast, ship smart, and scale before the opportunity closes.

This technical guide is designed to help you on this journey. With a focus on code-first development using tools like Agent Development Kit (ADK), Antigravity, and Agents CLI, it's geared towards helping startups and developers who are racing to embrace the potential of agentic systems.

Are you ready to dive in?

Choose your development path

There are many ways to create AI agents, and Google Cloud is compatible with different agent and model types. The path you choose will depend on your technical requirements and expertise—this guide hones in on the code-first route that's fast and cost-effective for startups.

"Our startup needs ...



... rapid validation, have limited engineering resources, and want to test a hypothesis."

Low-code development

Agent Studio is a collaborative, visual workspace for discovering models, engineering prompts, and building agents without writing code. Ideal for rapid prototyping and business-centric agents.



... custom programmatic logic and secure tool execution without managing infrastructure, web servers, or container deployment pipelines."

Managed code

Managed Agents API on Gemini platform is a config-driven, REST-first API for building autonomous agents inside a fully managed sandbox environment for actions. This path bridges the gap between low-code ease and custom code flexibility.

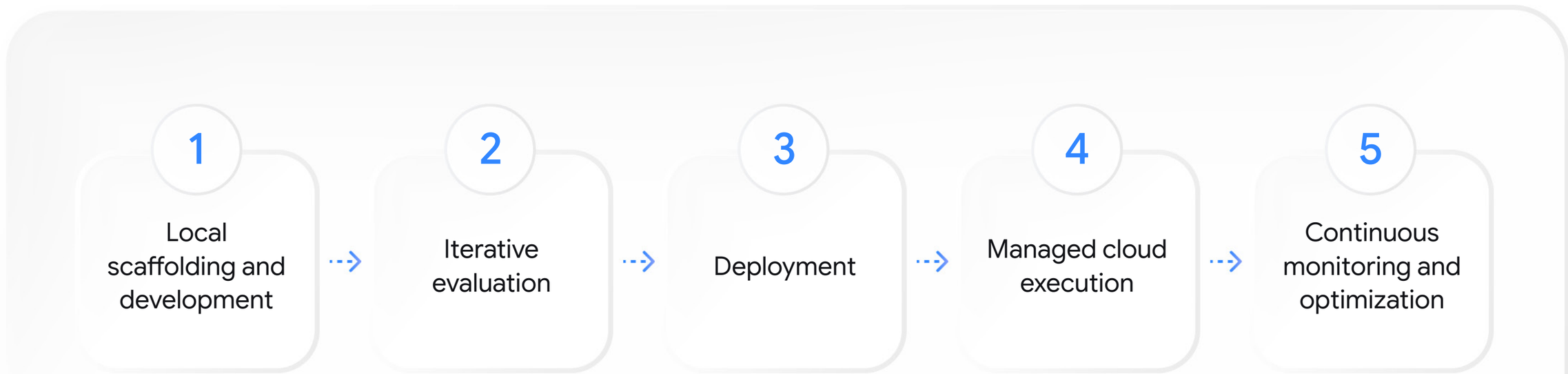


... deep programmatic control, **multi-agent coordination,** specialized tools, and full stack integration."

Custom, code-first development

ADK is a powerful framework for developers to build complex, multi-agent orchestrations with granular control over logic, tools, and environment simulation.

The path to product development



The focus of this guide

The agentic AI ecosystem offers many tools, libraries, and approaches for building cognitive architectures. This guide focuses primarily on ADK, Antigravity, and Agents CLI, sharing concepts and architectural patterns that allow you to build robust, scalable agents on Google Cloud while retaining the ability to integrate other preferred tools and libraries.

Throughout this AI agent journey, Gemini provides a single, secure platform where you can create, deploy, and govern all of your AI agents—whether made by Google, third parties, or your own teams—in one place.

Chapter 1

Lay the foundations

Model selection

Choosing the right model is not about selecting the most powerful one available, but about finding the optimal balance of capability, speed, and cost for your use case.

Typically, as a model's capability increases, so do cost and latency. The most common mistake is over-investing in capability when a use case doesn't need it, leading to inefficient spending and slower performance.

The strategy of selecting the most efficient model for any given task is most powerfully applied at a system level. Robust cognitive architectures employ multiple specialized agents that select the leanest model for their specific sub-tasks. Heavyweight models are reserved for complex reasoning, while lightweight models handle routine queries—giving you the flexibility to optimize the cost and performance of the entire system, not just a single component.

Managing models in multi-agent systems

Developers can use the Gemini family of models to break down problems, formulate plans, and use tools. By allocating more reasoning tokens to a specific call, you can direct the model to expend more computational effort. This token-level control, combined with model selection and configurable reasoning modes, offers a dynamic set of levers for sophisticated optimization. The cost and performance of entire multi-agent systems can be calibrated to meet specific business and technical requirements.

ADK is built to support this idea of treating models as interchangeable reasoning engines rather than hardcoded logic dependencies. It is model agnostic by design—so models can be swapped in code without rewriting tool definitions, memory logic, or orchestration loops.

When it comes to choice, Model Garden on Gemini platform provides a curated set of 200+ models from Google and beyond. For example, you could deploy Gemma open-weights, Meta Llama, Anthropic Claude, or Mistral under unified IAM, quota management, and Model Armor security perimeters.

Ultimately, Model Garden helps developers avoid the trap of closed multi-agent frameworks by enabling the deployment of diverse models inside the same agentic flow—an essential prerequisite for multi-model orchestration.

Use cases

Early-stage prototyping and at-scale tasks	Use a lightweight, low-cost model like Gemini Flash-Lite. This is the fastest, most cost-efficient model, excelling at high-volume, latency-sensitive tasks like translation and classification.
High-volume, high-quality applications	Use a balanced, mid-range model like Gemini Flash. This model controls the trade-off between quality, cost, and speed. Delivering strong performance on complex tasks at a lower price point than Pro, it's perfect for production applications that need to be both smart and economical.
Complex, multi-step reasoning and frontier code generation	Use an advanced reasoning model like Gemini Pro. This is Gemini's most intelligent model for multimodal understanding, and our most powerful agentic and vibe-coding model.

Capacity planning

When building and scaling multi-agent systems, capacity planning is a critical consideration.

Without adequate planning, the performance of AI workloads can be affected by things like quota limits, insufficient availability, or cost blow-outs. This is particularly true of “bursty” startup workloads where data processing and traffic demands fluctuate unpredictably.

To help startups optimize for both cost and performance, Google Cloud offers a range of consumption models. For mission-critical workloads, there’s the assured throughput and low latency of Provisioned Throughput—which also mitigates the risk of 429 errors. For other latency-sensitive requests that are more 429-tolerant, a standard pay-as-you-go model may suffice. And then there are the more cost-effective Deferred and Batch options for asynchronous jobs.

Google Cloud’s portfolio of consumption models

SLA-backed ← → No SLA (best effort)

Provisioned Throughput

Mission-critical baseload

For critical, stable traffic where performance and price predictability are essential.

Priority PayGo

Critical, variable workloads

For critical, user-facing traffic that requires high reliability and lower latency during peak periods.

Standard PayGo

Standard, general purpose workloads

For standard, variable traffic on workloads without strict availability SLAs.

Deferred execution

Substantial discounts for work that can wait

AI workloads can run on separate, off-peak capacity. You pay up to half the inference cost and bypass standard quota limits entirely.

Batch

Asynchronous batched jobs

Throughput heavy, latency insensitive workloads: entity extraction/data prep, classification, sentiment analysis.

Prompt caching
Global endpoint
In-country ML processing

Pro tip

If scaling AI is an engineering challenge, managing the bill is a critical business strategy. As you transition from building prototypes to deploying production-grade agentic systems, apply these strategies to keep your model costs lean and efficient.

Watch webinar

Chapter 2

Build with ADK and Antigravity

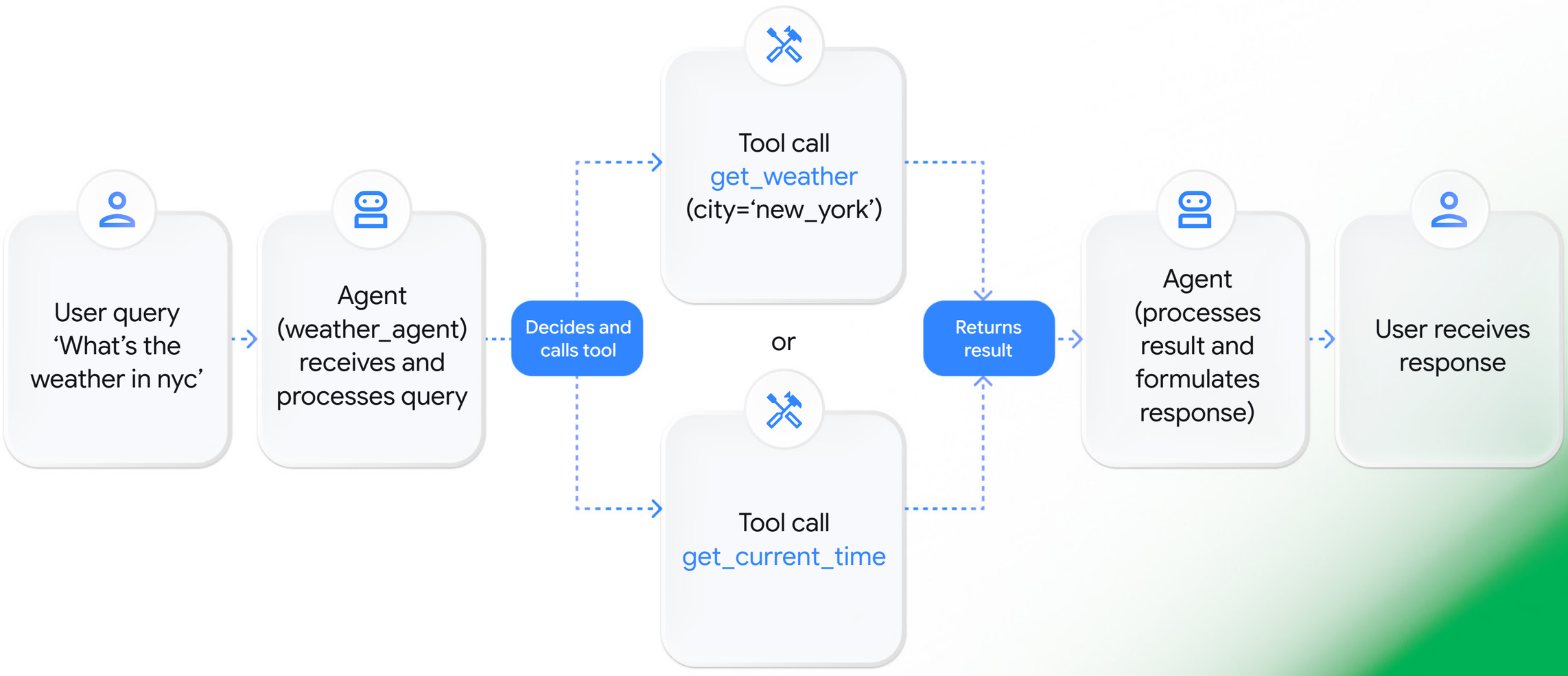
Custom, code-first development with ADK

If you need more power than a simple low-code tool but want an accelerated development process, ADK gives you the control to build a unique system of collaborative agents while simplifying complex technical challenges.

The flexible and modular open-source framework for developing and deploying agents is ideal for developers, technical startups, and teams that require a high degree of control over agent behavior. It is model- and deployment-agnostic, with all the tools you need to build, manage, evaluate, and deploy highly specialized and sophisticated multi-agent systems.

Agents that are built with ADK and deployed on Agent Runtime can easily be registered with Gemini platform, which can serve as the central hub for governance and inventory of your AI ecosystem. Because these agents are containerized, they can also be deployed to any environment that runs containers, such as Cloud Run and Google Kubernetes Engine (GKE).

Building complex workflows gets easier with ADK



Core capabilities

Orchestration logic

The agent's core reasoning process, like the ReAct framework, allows it to plan and execute a sequence of tool calls and actions to accomplish a complex goal.

Tool definition and registration

An interface for defining custom functions and APIs, allowing the agent to interact with data, APIs, and external systems.

Context management

A system that provides the agent with memory, allowing you to use the agent to recall user preferences and conversational history across multiple interactions. This seamless integration of short-term conversational context with long-term knowledge helps build better customer relationships.

Containerization

The capability to package the agent into a standard, portable container, making it ready for deployment on any compatible cloud environment. This delivers a faster, more reliable path to production using standard DevOps practices.

Evaluation and observability

A suite of built-in tools to rigorously test agent quality, debug the agent's step-by-step reasoning, and monitor its performance in a production environment.

Multi-agent composition

The ability to build systems where multiple specialized agents can collaborate, delegate tasks, and work together to solve a problem.

Put theory into practice

This step-by-step workshop teaches you how to build, orchestrate, evaluate, and deploy production-grade AI agents using ADK.

Start workshop

ADK is built for startup speed and scale

With its powerful tools for building sophisticated AI agents,
ADK helps startups maintain control, predictability, and reliability.

Automate workflows, not just conversations

Implement multi-step orchestration logic to solve complex business problems. This delivers the operational leverage small teams need to scale.

Integrate AI into existing tools, agents, and workflows

ADK is built around a rich tool ecosystem that allows your agents to interact with all your existing tools and data. You can connect your agents to productivity tools like Notion, Slack, or a CRM, as well as tool frameworks like LangChain and LlamaIndex, or agent frameworks like LangGraph or CrewAI.

Ensure quality and reliability from day one

To earn users' trust, it's critical to carefully test and evaluate your agents before deploying them to production. ADK's built-in observability and evaluation tools help you:

- **Rapidly iterate:** Before deploying, you can systematically test how your agents respond to various scenarios and how well they execute complex tasks.
- **Debug agent behavior:** Inspect the full execution trace of your agent including its reasoning (thoughts), tool calls, and observations to understand its decision-making process and debug complex, multi-step workflows.
- **Benchmark your agents:** Evaluate different agent designs or model updates against predefined metrics, using a data-driven approach to continuously improve agent performance.

This moves your process beyond simple "vibe-testing," allowing you to launch professional-grade agents quickly, build user trust through proven reliability, and iterate with data-driven confidence.

Scale AI with confidence

As you grow, your AI solutions need to scale seamlessly without becoming a bottleneck. Agents CLI can automate the entire deployment phase for your ADK agents. It seamlessly injects Infrastructure as Code (IaC), sets up CI/CD pipelines, and deploys directly to Agent Runtime, Cloud Run, or GKE.

ADK core: Selecting agent architectures

Distinct agent architectures are designed for different execution patterns, and understanding the interplay between them is key to building robust and effective AI systems.

Your choice of agent architecture—typically a trade-off between the flexible, non-deterministic power of an LLM and the predictable, deterministic control of hard-coded logic—will determine how your agent reasons and operates.

ADK's agent types are organized into three categories

LLM agent LlmAgent

Core engine
LLM

Determinism
Non-deterministic (flexible)

This is the most common agent type and is commonly referred to as simply “Agent.” It uses an LLM like Gemini for complex reasoning, dynamic decision-making, and natural language understanding; and it forms the core of most conversational and problem-solving agents.

Workflow agent SequentialAgent ParallelAgent LoopAgent

Core engine
Predefined logic

Determinism
Deterministic (predictable)

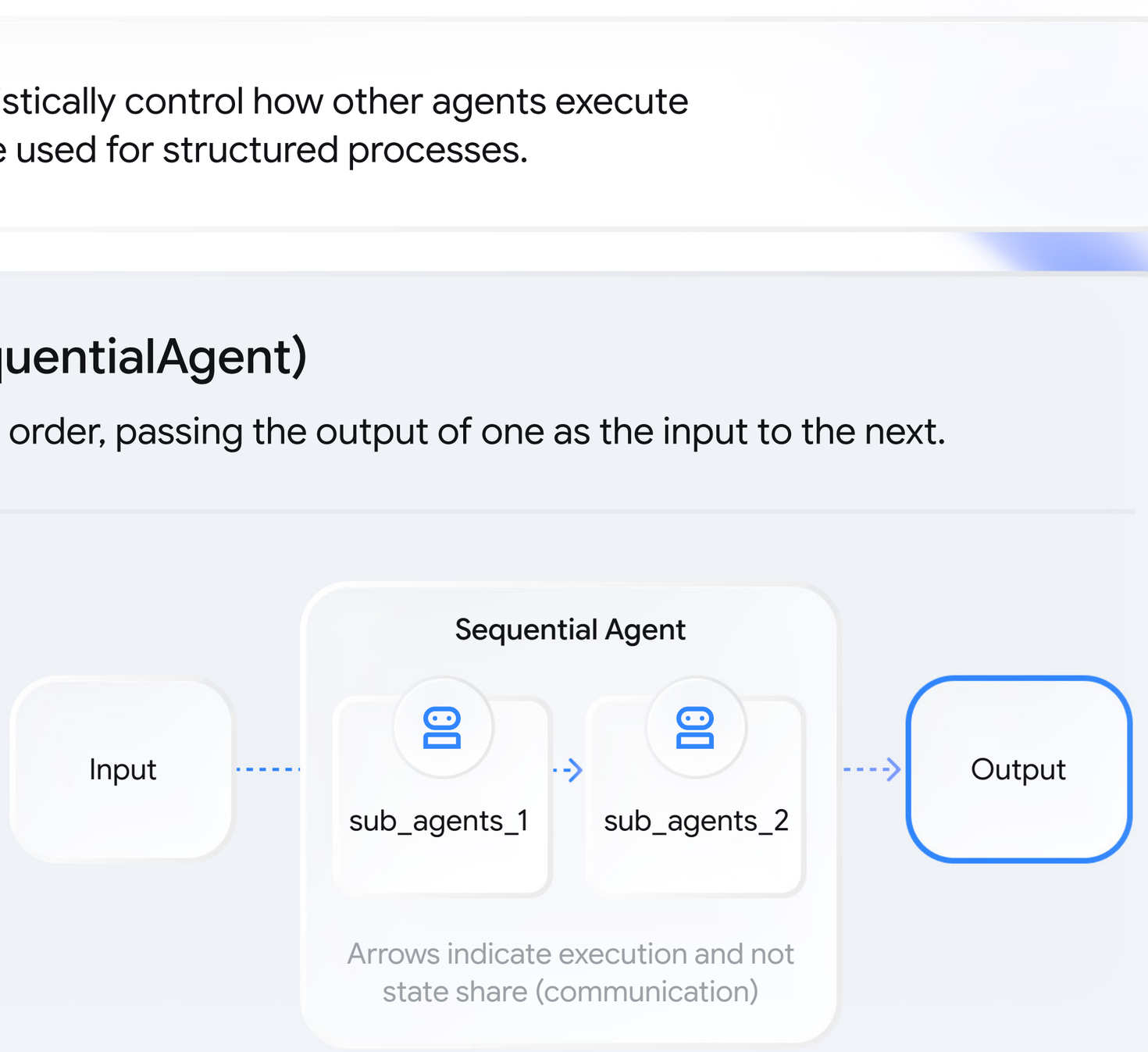
These orchestrators deterministically control how other agents execute in predefined patterns. They’re used for structured processes.

Sequential agents (SequentialAgent)

Executes sub-agents in a fixed order, passing the output of one as the input to the next.

Agent A completes **Task 1**, then passes its output to **Agent B** for **Task 2**.

Get the full code



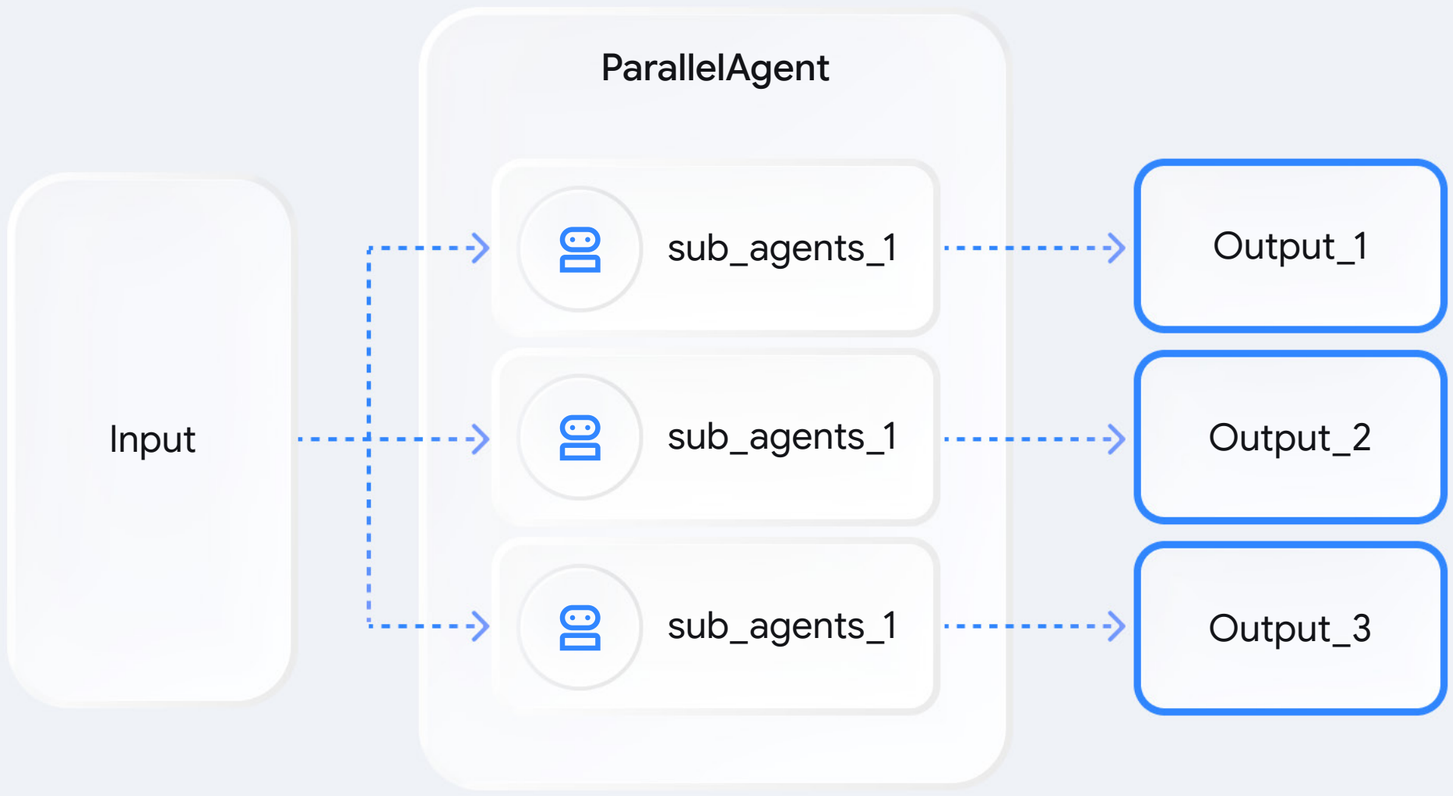
Example
You want to build an agent that can summarize any webpage, using two tools: **Get Page Contents** and **Summarize Page**. Because you can't summarize from nothing, the agent must always call **Get Page Contents** before calling **Summarize Page.**

Parallel agents (ParallelAgent)

Executes multiple sub-agents simultaneously. This is used for performance optimization when tasks are independent.

Agent A and **Agent B** work on independent sub-tasks simultaneously, and their results are combined.

Get the full code

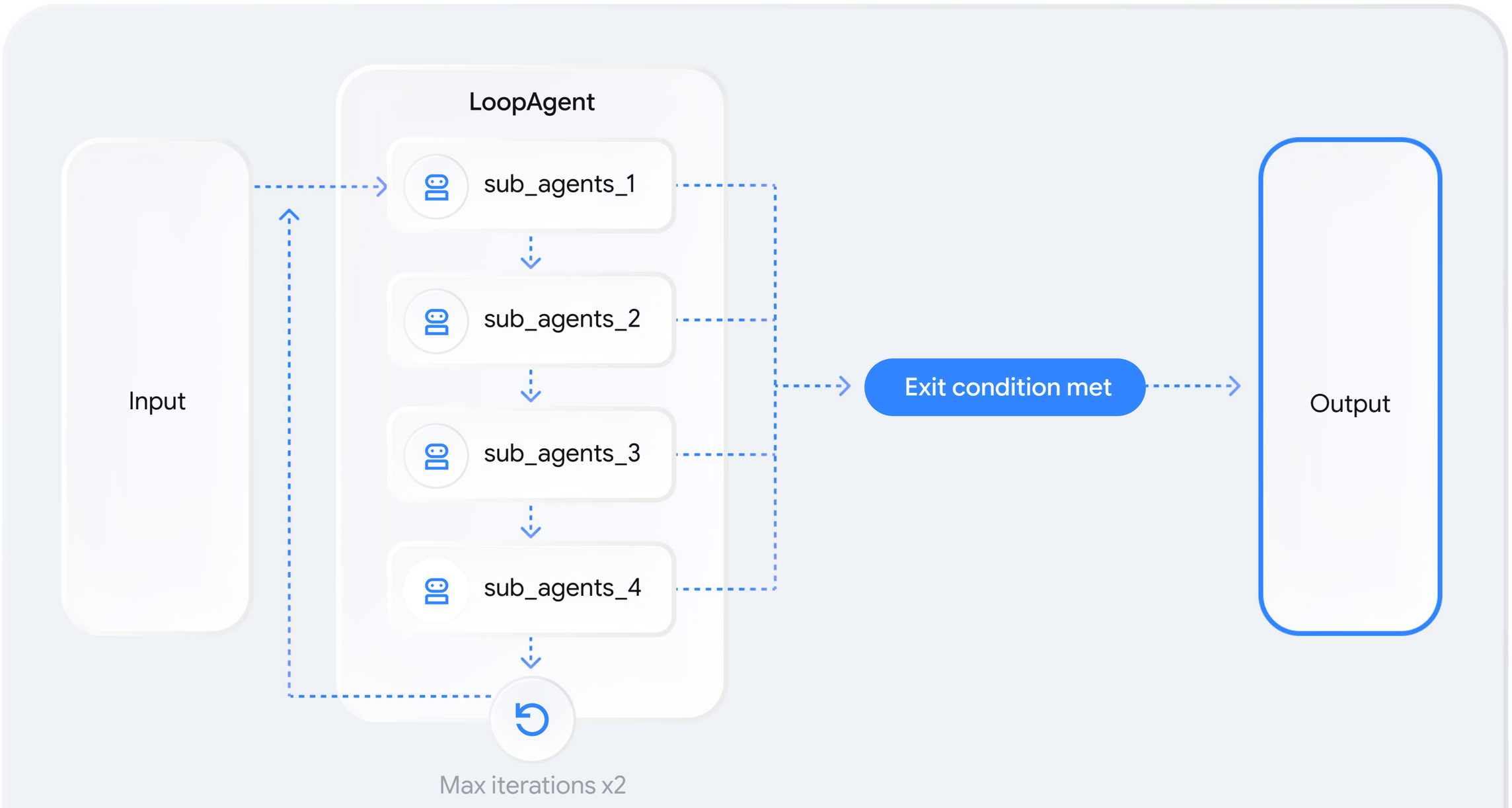


Example
For operations like multi-source data retrieval or heavy computations, parallelization yields substantial performance gains. Importantly, this strategy assumes no inherent need for shared state or direct information exchange between the concurrently executing agents.

Loop agents (LoopAgent)

A workflow agent executes its sub-agents in a loop (iteratively). It repeatedly runs a sequence of agents for a specified number of iterations or until a termination condition is met. Use the LoopAgent when your workflow involves repetition or iterative refinement, such as revising code.

Get the full code



Example

You want to build an agent that can generate images that contain specific amounts of food (e.g., five bananas), using two tools: Generate Image, Count Food Items. Because you want to keep generating images until it either correctly generates the specified number of items, or after a certain number of iterations, you should build your agent using a LoopAgent.

Custom agent BaseAgent subclass

Core engine

Custom Python code

Determinism

Can be either, based on implementation

Custom Agent (BaseAgent subclass)

For unique requirements and tailored workflows that go beyond a standard reasoning loop, you can create a custom agent by inheriting directly from **BaseAgent** and writing custom Python logic to control its behavior. This approach is necessary when an agent's actions are not determined by an LLM, but by specific, hard-coded rules.

How it's done

A developer creates a new Python class that inherits from the **BaseAgent** class. They must then implement the **_run_async_impl** method, which contains the unique logic the agent will execute. This method has full access to the session's state and can yield events to communicate with other agents or terminate a workflow.

ADK orchestration: Implementing the ReAct loop

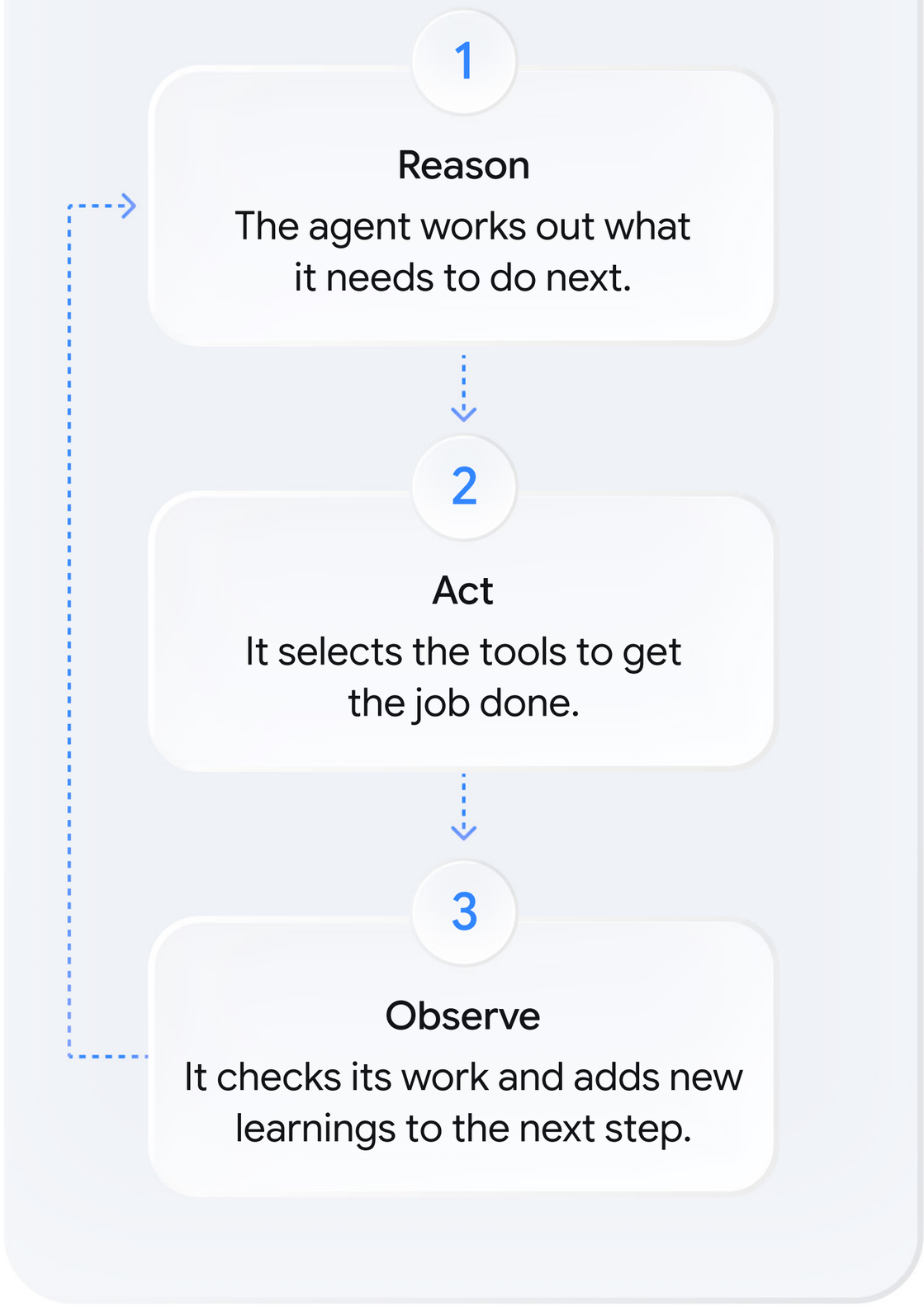
ReAct (Reason + Action) is a common and effective orchestration pattern that establishes a dynamic, multi-turn loop where the model generates both reasoning traces and task-specific actions in an interleaved manner. The reasoning helps the model track and update action plans, while actions gather information from external tools to inform the reasoning process.

ADK provides the core abstractions and classes needed to implement this dynamic, cyclical process in a structured way. Its LlmAgent is specifically designed to execute the loop and handle the transitions between the fundamental stages in the loop:

1. **Reasoning (thought):** The LlmAgent class manages this stage internally. It takes the user's prompt and its current internal state, then calls the underlying language model to form a hypothesis and determine the next best action.
2. **Action (tool use and agent delegation):** ADK enables this stage through its flexible tool system. When the LlmAgent decides to act, it can invoke a simple Python function or, for more complex tasks, delegate the work to another specialized sub-agent using the Agent-as-a-Tool pattern.
3. **Observation:** ADK automatically captures the dictionary returned by the tool or sub-agent and passes it back to the LlmAgent. This output becomes the new information that the agent integrates into its context, feeding it into the next Reasoning step of the cycle.

By providing a native implementation of this essential pattern, ADK abstracts away the boilerplate code, so you can quickly translate the powerful concept of a ReAct loop into a working, multi-step agent.

The ReAct loop in action



ADK tools: A framework for agentic action

In ADK, an agent can use tools to perform actions beyond the native capabilities of its core reasoning model. Tools bridge the gap between the agent's reasoning and its ability to retrieve new information or execute stateful operations.

A tool is a Python function (or a Java method) that can either implement self-contained logic or act as a wrapper for more complex operations.

It can include a wide variety of components:

- **Internal functions and services:** Proprietary logic written by your own team.
- **APIs:** Connections to both internal services and external, third-party services.
- **Data sources:** The ability to query databases, vector stores, or other repositories of information.
- **Other agents:** In a multi-agent system, one agent can use another specialized agent as a tool.

Designing effective tools

For a model to use a tool correctly, its definition must serve as a clear and unambiguous API contract, composed of:

Function signature

Use descriptive names for tools and their parameters. Python type hints are mandatory, as they provide the structural schema the model uses to generate valid arguments.

Docstring (the semantic core)

This is the primary source of semantic information for the model. A well-written docstring must precisely define the tool's purpose, usage criteria, parameters, and expected return schema.

Return schema

A tool must return a dictionary. While not a strict syntactic requirement, it is a best practice to include a **status** key (e.g., '**success**' or '**error**') in this dictionary. This structure is essential for the agent to reliably distinguish between successful outcomes and failures in its Observation step and reason about how to proceed.

Stateful tools and ToolContext

For tools that need to read or write to a persistent session state, an optional **tool_context: ToolContext** parameter can be added to the function signature. The agent automatically injects this object, giving the tool access to a session-level state dictionary.

A taxonomy of ADK tools

ADK provides a flexible architecture for implementing tools, ranging from simple functions to interoperable multi-agent systems.

Toolsets for packaging related capabilities

Toolsets bundle a collection of related tools into a single, configurable object (e.g., **BigQueryToolset, MCPToolset**).

Hierarchical and remote tools

ADK enables the creation of complex systems by composing agents:

- **Agent-as-a-tool:** A delegation pattern where a parent agent uses another specialized agent. This allows the parent to invoke another agent, receive a response, and maintain control to handle future input. (This is distinct from a sub-agent delegation model, where full conversational control is passed to a sub-agent and all subsequent input will be handled by the sub-agent.)
- **RemoteA2aAgent:** For communication between agents in different processes, ADK provides the RemoteA2aAgent class, which uses the Agent-to-Agent (A2A) protocol to seamlessly integrate distributed systems.

Custom function tools

These are used to extend an agent with proprietary logic:

- **FunctionTool:** The standard wrapper for synchronous Python functions.
- **LongRunningFunctionTool:** A specialized tool for asynchronous tasks or human-in-the-loop workflows.

Pre-built and integrated tools

ADK includes a suite of tools and wrappers to accelerate development:

- **Built-in code execution tooling:** Native code execution is handled directly by Gemini model runtime (dynamic Python code sandboxes). When running externally via ADK, it is standard to point to Antigravity Remote Sandboxes or Cloud Run containers for isolated compute.
- **Google Cloud toolsets:** Rich integrations for services like Agent Search and BigQuery.

Pro tip

For best practices and examples of how to define parameters and tool schemas, structure effective prompts, and implement complex, multi-agent workflows, check out the ADK Samples repository.

Pro tip

For a complete discussion, including code examples and advanced usage patterns, refer to the ADK documentation.

Accelerated development with Antigravity

Google Antigravity is an agentic development platform that bridges the gap between abstract prompt engineering and concrete backend architecture, ultimately accelerating the path to production.

With Antigravity, your builders gain a dedicated mission control where they can work alongside, steer, and customize agents that operate autonomously in the background. Its browser control capabilities, asynchronous interaction patterns, and agent-first product form factor enable agents to autonomously plan and execute complex, end-to-end software tasks while keeping the human as the manager and approver.

Developers can use the Antigravity 2.0 desktop app, lightweight **agy** terminal CLI or Antigravity IDE extensions (VS, VSCode, Zed, JetBrains) as an autonomous dev companion to generate, scaffold, and test code, including ADK.

Put theory into practice

This codelab teaches you how to get started with Google Antigravity and the Antigravity IDE.

Start codelab

3 ways Antigravity drives efficiency

1

Writing prompts at the “Goldilocks” altitude

Antigravity moves away from monolithic, over-constrained prompts that break when edge cases arise. In the **agy** CLI or IDE, your prompts focus purely on intent, goals, and high-level direction. Antigravity handles the rest under the hood—dynamically injecting schema rules, runtime constraints, and role guardrails via modular manifests. You get flexible, adaptive interaction without having to micromanage rigid formatting rules in every prompt.

2

Enabling real-time backend actions

Instead of forcing the LLM to write raw execution code or manage API calls, Antigravity acts as a unified tool engine that translates the model’s structured intent into safe, deterministic backend actions. It uses three primary mechanisms to do this:

- **OpenAPI schemas:** When Gemini generates a function call based on the schema, Antigravity’s runtime validates the arguments against the OpenAPI parameter specifications before executing the HTTP request. If the arguments violate the schema, the runtime intercepts the payload and requests a correction from the LLM, shielding your external REST APIs from invalid data payloads.
- **Custom Python functions:** You can expose local Python functions directly to an agent by wrapping them with Antigravity’s tool decorators or registers. The SDK inspects Python type hints, docstrings, and Pydantic schemas to dynamically generate the schema the LLM reads.
- **Native Google Search grounding:** Antigravity includes pre-configured, native wrappers for Google Search grounding. It treats real-time web retrieval as an integrated runtime capability, appending verifiable ground-truth citations to the context, anchoring the model’s context in real-time web data. This prevents hallucinated assumptions from spilling down into your custom Python functions or OpenAPI calls.

3

Managing runtime with state transitions

To prevent agents from getting stuck in endless retries or recursive logic loops, Antigravity emphasizes deterministic state transitions using Python-driven graph structures or explicit state machines.

Your developers can define explicit state objects in Python to ensure that transitions between sub-agents occur via explicit programmatic triggers or defined exit conditions rather than unconstrained LLM autonomy. You can also enforce structural boundaries such as capped iteration loops, hard fallback branches, and system-managed instruction updates per state node.

Chapter 3

Manage memory and context

Managing your data with Google Cloud services

In an ADK-driven agent architecture, an agent's data layer is organized around how data is consumed and mutated during the execution loop. Each stage of this loop—and its supporting tools—is described below.

1

Working memory

This is the agent's high-speed, transient scratchpad for managing immediate conversational turns, sub-task isolation, and tool response deduplication. It's used for caching and real-time context.

- **SessionService:** Manages the different conversation threads (**Session objects**) —creating, retrieving, updating, and deleting individual **Session(s)**.
- **MemoryService:** Manages the long-term knowledge store (**Memory**), ingesting information (often from completed **Sessions**) into the long-term store and providing methods to search this stored knowledge based on queries.

2

Conversational state and distilled memory

To avoid dumping thousands of raw chat tokens into the prompt, this layer retains user context and history efficiently, across days, weeks, or months.

- **Memory Bank:** Provides asynchronous and agent-directed memory distillation (**GenerateMemories**, **CreateMemory**). It extracts salient facts and user preferences into compact, structured long-term memory so models don't suffer from context rot or high token costs.
- **Firestore:** Acts as the agent's serverless operational document store. In ADK, it persists conversation state, task checkpoints, and execution history across multi-session, long-running agent workflows.

3

Enterprise grounding and the open lakehouse

This knowledge base is the single source of truth for unstructured documentation, petabyte-scale analytics, and raw multimodal assets.

- **Data Lakehouse (Lakehouse for Apache Iceberg):** Unifies structured tabular data and unstructured storage into an open data plane. Agents can run federated queries across enterprise datasets without custom ETL pipelines or vendor lock-in.
- **BigQuery:** Serves as the agent's analytical engine. Using the ADK **BigQueryToolset** or standardized MCP tools, agents execute natural-language queries against massive datasets to answer analytical questions safely.
- **Agent Search on Gemini platform:** Operates as the agent's search and semantic retrieval library. In ADK, the **VertexAISearchToolset** grounds agent reasoning in internal enterprise knowledge bases and documentation.
- **Cloud Storage:** The durable multimodal object store. ADK agents reference raw assets (PDFs, audio, videos, images) directly via URIs into Gemini's multimodal context window without complex pre-processing.

4

Transactional memory and auditing

This is the agent's durable ledger or system of record, required for executing state changes, business actions, and compliance logging with ACID guarantees.

- **Cloud SQL:** Serves as the agent's standard operational relational database. ADK tools log deterministic action records to **Cloud SQL**, providing a reliable, ACID-compliant audit trail of every agent-executed step.
- **Cloud Spanner:** The globally consistent, mission-critical backbone. When an ADK tool executes an irreversible enterprise action (e.g., **process_global_order**), Spanner guarantees zero-downtime, multi-region transactional integrity.

Grounding your agentic systems

An agent's credibility and usefulness depends on its ability to provide accurate, trustworthy answers based on verifiable facts. This is achieved through various types of grounding.

RAG: A foundational first step

Retrieval-augmented generation (RAG) enhances an LLM's responses by retrieving relevant information from an external knowledge base before generating an answer. Instead of relying solely on its pre-trained knowledge, the agent performs a semantic search to find verifiable data, which is then passed to the LLM as context. This ensures a baseline of grounded, verifiable answers.

While foundational, this simple retrieve-then-generate process treats knowledge as a flat collection of disconnected facts. It's a powerful technique for direct question-answering, but it falls short on complex queries that require a deeper understanding of the relationships between data points. The benefits of RAG for agentic systems include:

Access to the latest information

The retrieved data is more current than the agent's last training date, enabling more timely and relevant agentic behavior.

Greater accuracy

RAG significantly reduces the risk of outputs that could lead to incorrect or inappropriate agentic actions.

Faster responses

Vector embeddings and specialized databases enable lightning-fast semantic searches of massive datasets, so agents can deliver more responsive, timely decisions.

More comprehensive agent awareness

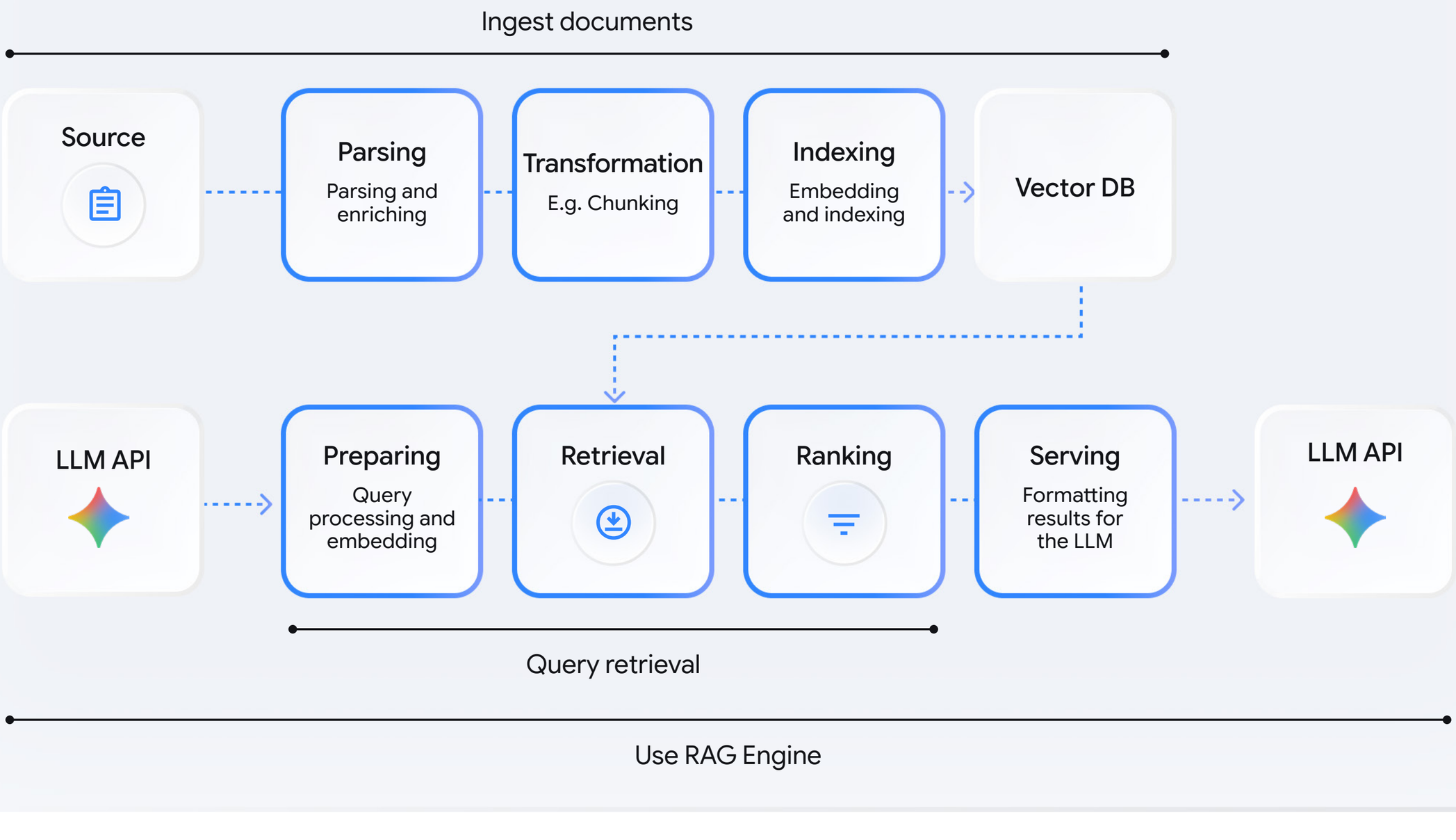
The RAG workflow, which consists of ingesting, parsing, chunking, embedding, storing, and retrieving, can be applied to text, images, and other types of data. With this deeper understanding, agents can perform more complex, multistep reasoning tasks.

Agent Search on Gemini platform is Google Cloud's managed, out-of-the-box RAG solution. Then there's Agent Retrieval (formerly Vector Search 2.0), a self-tuning, fully managed, AI-native search engine architecture that provides a replicated, scalable storage engine—giving you a single, unified data source for your AI applications and removing the need for auxiliary data storage.

See it in action

Explore Agent Retrieval in this interactive demo, and see how the fully managed vector database delivers the search and storage features you need without having to manage infrastructure.

RAG Engine at work



GraphRAG: Smarter grounding

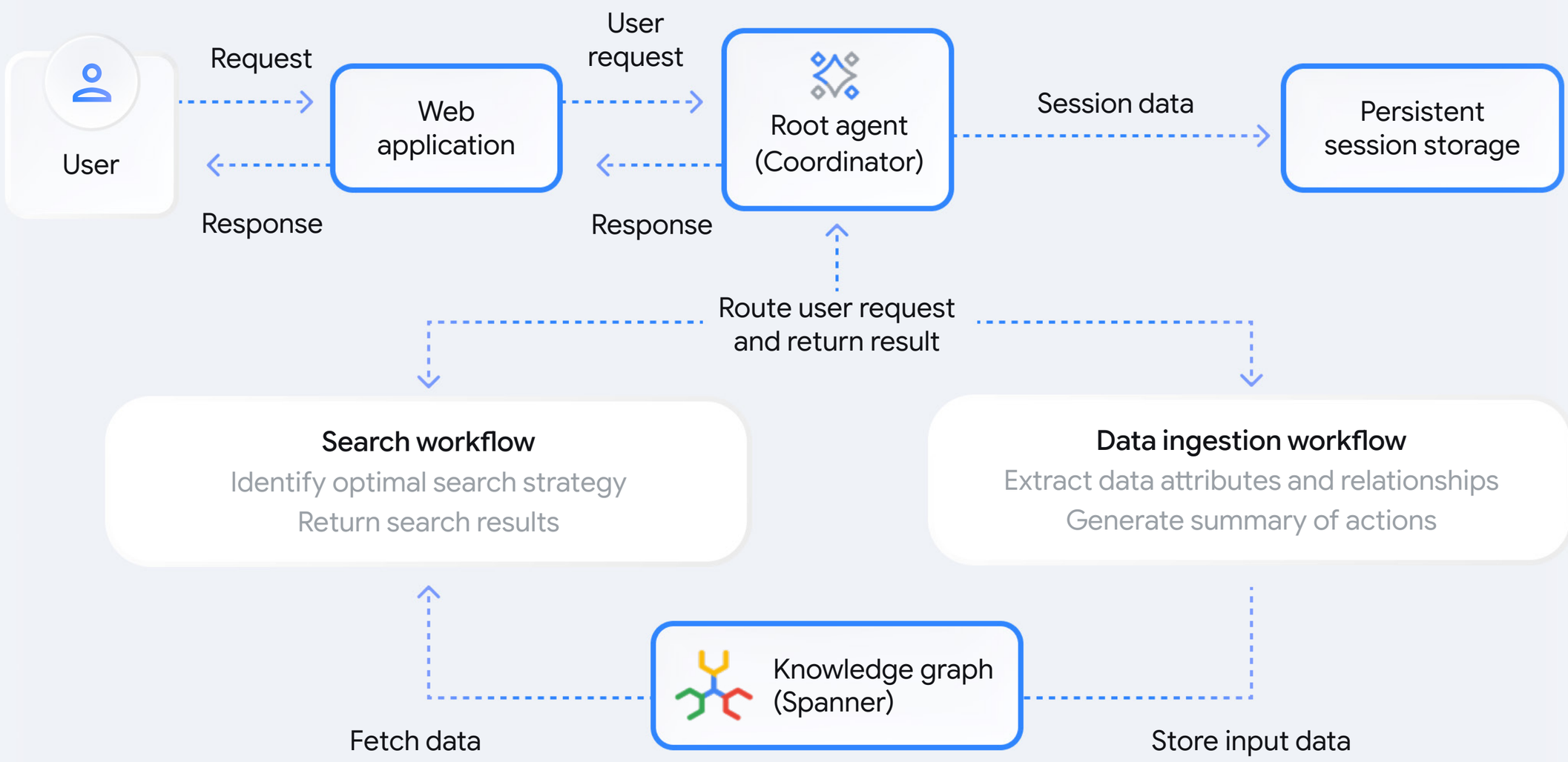
GraphRAG builds a knowledge graph, so instead of just matching similar phrases, your agent understands how concepts relate.

A multi-agent AI system that enables multimodal GraphRAG resource orchestration can produce more detailed and relevant responses.

Use case

A medical AI assistant that needs to know "symptoms → causes → treatments" and not just retrieve related snippets.

High-level view of a multi-agent system using GraphRAG



In the above diagram, the data ingestion workflow consolidates fragmented multimodal records into the unified knowledge graph. This workflow uses a sequential multi-agent pattern to upload raw data to Cloud Storage, extract complex relationships using Gemini, and update the Spanner Graph knowledge graph to ensure that the data is available for retrieval.

The search workflow lets you retrieve insights and grounded responses through a GraphRAG process. This workflow uses Gemini and specialized subagents to navigate the knowledge graph and consolidate findings based on previous session data.

Agentic RAG: Dynamic reasoning and retrieval

The most powerful approach to grounding is Agentic RAG, a technique that helps you transform the agent from a passive recipient of retrieved data into an active, reasoning participant in the search for knowledge. Following frameworks like ReAct, the agent can analyze a complex query, formulate a multi-step plan, and execute multiple tool calls in sequence to find the best possible information.

A prime example of this agentic pattern is grounding with Google Search. You can use the Gemini family of models to integrate advanced reasoning, allowing them to interleave search capabilities with internal thought processes to answer complex, multi-hop queries and execute long-horizon tasks.

Using Gemini, your agents can also perceive and reason across different data types—transforming them from data processors into problem-solving tools that understand and interact with the world in a more complete way. For example, it could analyze a photo to identify a specific species of plant and then autonomously retrieve its detailed care instructions. Or it could process an audio stream from a support call to not only transcribe the words but also determine the customer's sentiment, such as frustration, to escalate the ticket properly.

Agentic RAG represents a fundamental leap. By empowering the agent to be an active, reasoning participant, your developers can build systems capable of executing the complex, multi-step queries and long-horizon tasks that define next-generation agentic capabilities.

For a startup, mastering this approach is the key to building a defensible, truly intelligent product that can unlock novel workflows and user experiences.

An example of Agentic RAG at work

Here's how Agentic RAG could help perform a real-time inventory check.

First, you define a function called **check_inventory** that takes a **product_ID** and returns the current stock levels from your real-time inventory system. Then, when a customer asks about a specific product's availability, the AI agent:

1

Identifies the product

It uses semantic search (powered by the vector database) to accurately identify the specific shoe model the customer is asking about, even if they describe it vaguely.

2

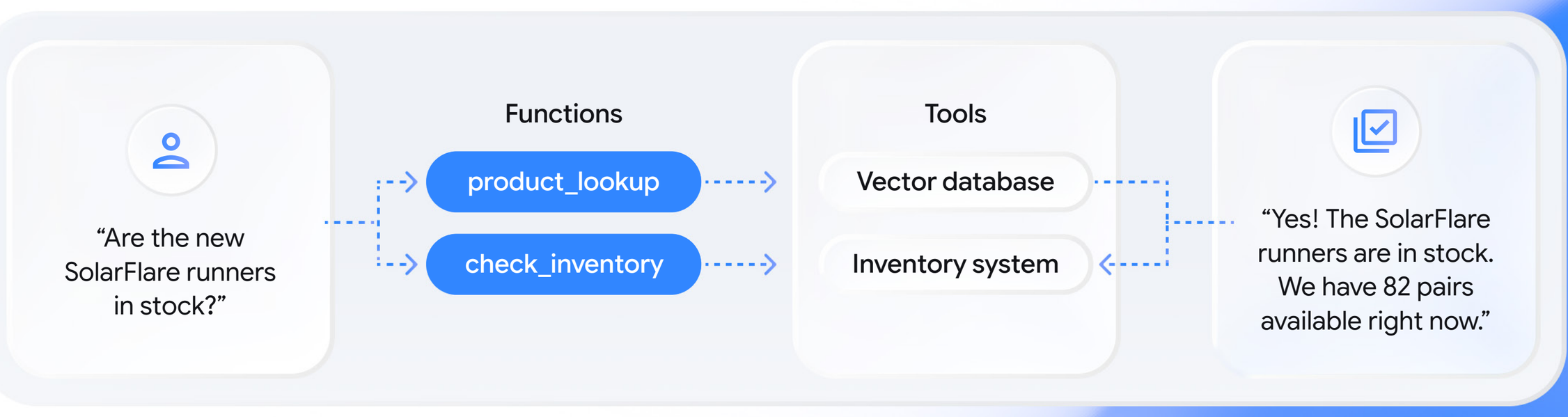
Triggers the action

It recognizes the need for real-time stock information and uses function calling to invoke your **check_inventory** function.

3

Provides real-time response

The **check_inventory** function executes, fetches the live stock data from your inventory system, and returns it to the AI agent. The agent then provides the customer with an immediate, accurate response about availability.



The Agentic RAG workflow on Google Cloud

Google Cloud provides managed services that handle the entire Agentic RAG workflow:

1

Generating embeddings and indexing

The first step is to convert your data into vector embeddings. An option is Gemini Embeddings, available in both Gemini platform and Gemini API, and supports over 100 languages.

2

Storing and indexing

The vector embeddings are then stored and indexed in Agent Retrieval. This fully managed, high-performance vector database automatically builds the specialized indexes required for fast and efficient similarity searches at scale.

3

Retrieval and reasoning

When a user submits a query, it is converted into an embedding and used by Agent Retrieval to find the most relevant information. This targeted context is then passed to the LLM to generate the final, grounded response.

Pro tip

Use the **retrieve and re-rank approach** to address the trade-off between recall (finding all relevant documents) and precision (ensuring every retrieved document is relevant).

This two-step approach firstly widens the recall aperture by configuring Agent Retrieval to retrieve a larger-than-needed set of candidate documents. Second, this larger set is passed to the LLM or a specialized re-ranking service, which identifies the most relevant documents and discards any that are irrelevant or semantically opposite.

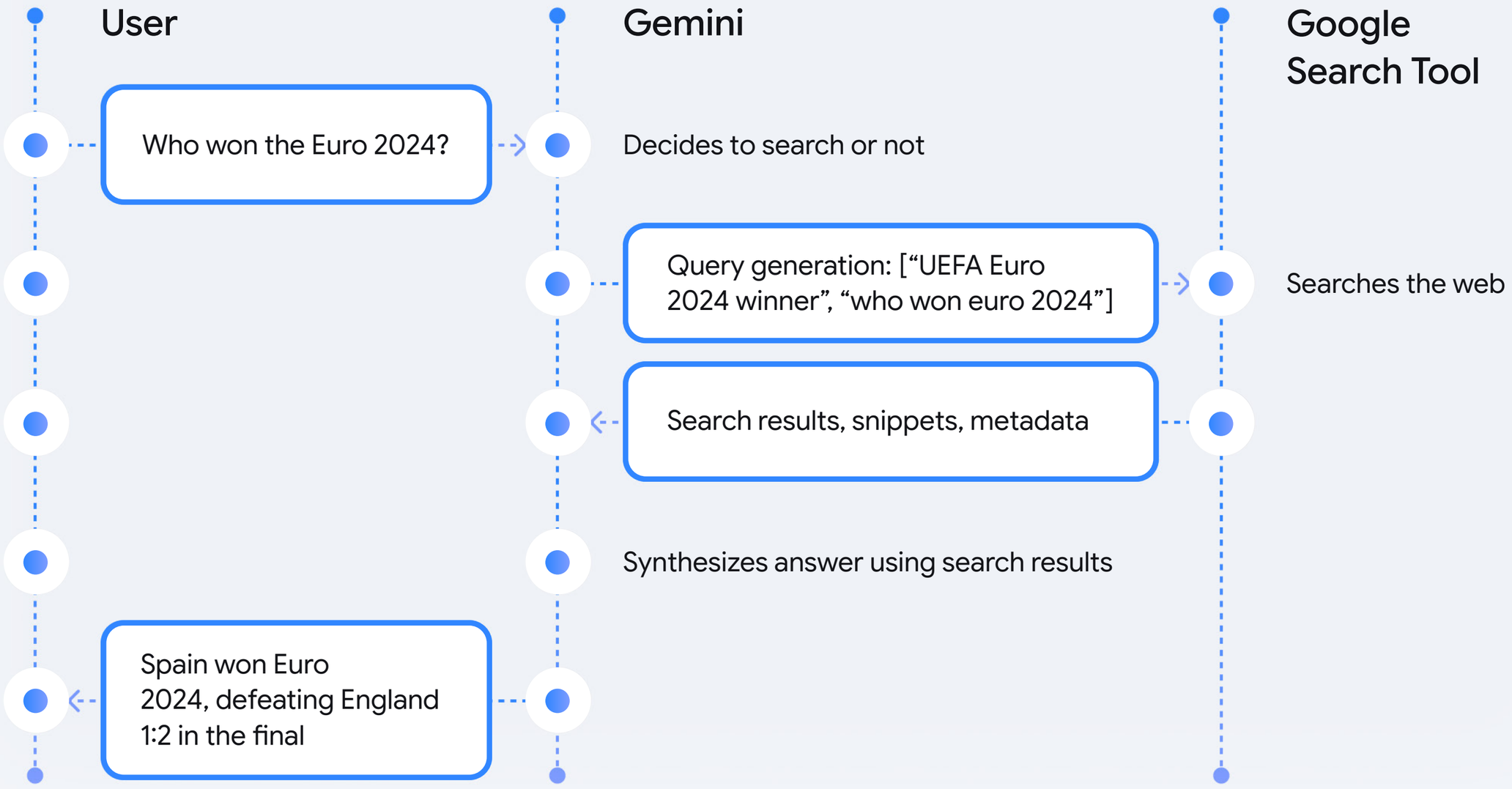
Other grounding methods

While RAG is a foundational technique, Gemini platform also offers other grounding techniques to ensure your agents deliver accurate, reliable responses. For example:

- **Grounding with Google Maps:** Use Google Maps data with your model to provide more accurate and context-aware responses to your prompts.
- **Grounding with Agent Search:** Use RAG to connect your model to your website data or your sets of documents stored in Agent Search.
- **Grounding with RAG Engine on Gemini platform:** Ground using your data through RAG Engine on the platform, which is a configurable managed RAG service.
- **Grounding with Google Search:** Connect your model to world knowledge and a wide possible range of topics.
- **Grounding Gemini with Elasticsearch:** Use RAG with your existing Elasticsearch indexes and Gemini.
- **Grounding with your search API:** Connect Gemini to your external data sources by grounding with any search API.
- **Web Grounding for Enterprise:** Use a web index suitable for highly-regulated industries to generate grounded responses with compliance controls.
- **Grounding with Parallel web search:** Connect Gemini to an LLM-optimized web index to use the most recent information from the web.

Grounding with Google Search

When you use the Google Search Tool, the model handles the entire workflow of searching, processing, and citing information automatically.



1

User prompt

The application sends the prompt to Gemini API with the Google Search tool.

2

Prompt analysis

The model analyzes the prompt and determines if a Google Search can improve the answer.

3

Google search

If needed, the model automatically generates one or multiple search queries and executes them.

4

Search results processing

The model processes the search results, synthesizes the information, and formulates a response.

5

Grounded response

The API returns a final, user-friendly response grounded in the search results, which includes the model's answer and grounding metadata (search queries, web results, and citations).

Chapter 4

Use protocols to support multi-agent systems

Standardize with Model Context Protocol

Model Context Protocol (MCP) is an emerging open standard for connecting AI and LLMs with external data sources and tools. You can plug your AI applications into various data sources and tools without the hassle of building custom point-to-point integrations for each one.

With ADK, your agents can participate in this ecosystem in two ways:

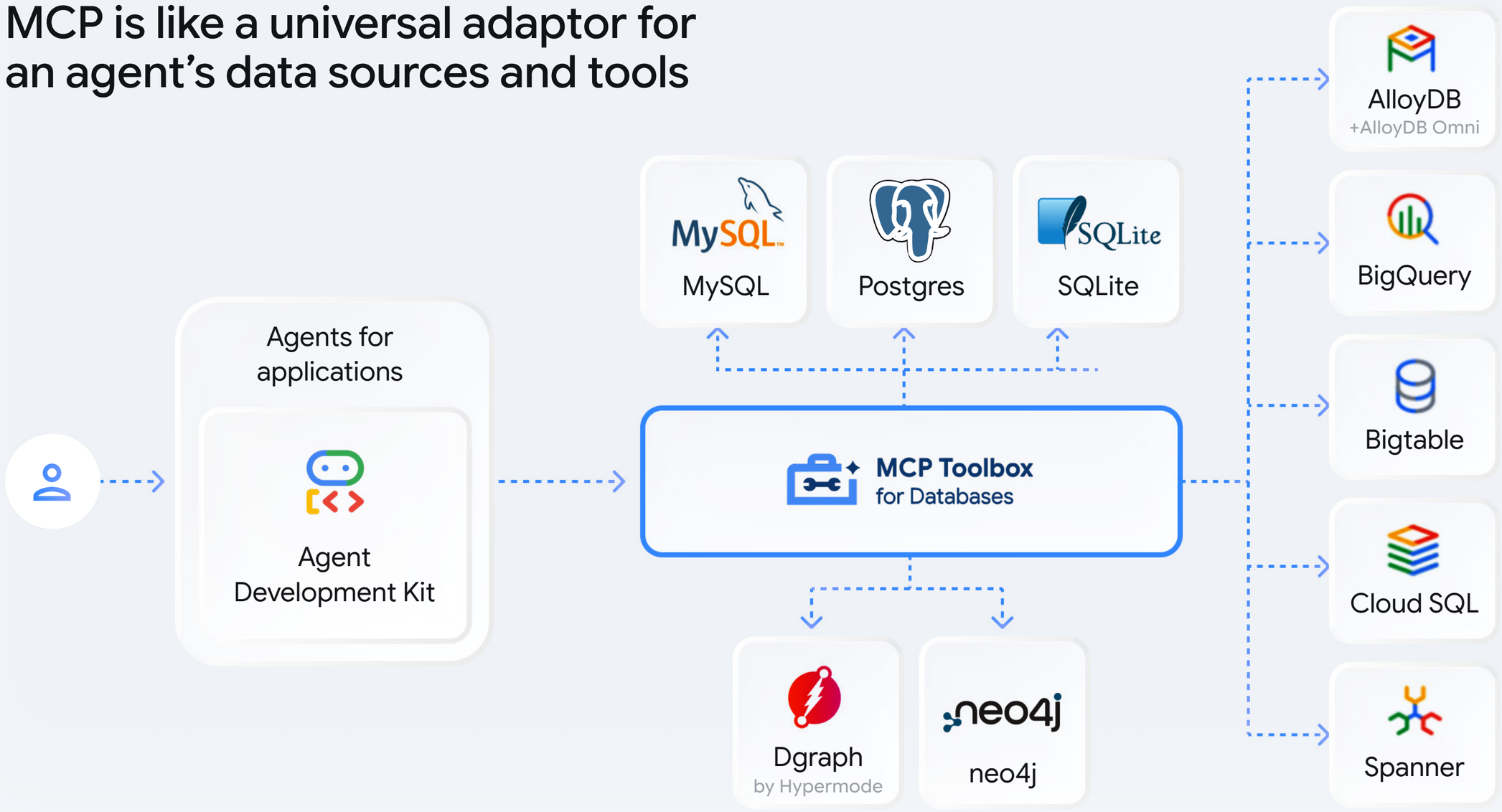
- **Consume external tools:** An ADK agent can act as an MCP client, allowing it to use any tool exposed by a third-party MCP server.
- **Expose native tools:** Developers can wrap their ADK tools in an MCP server, making them securely available to any other MCP-compliant agent or application.

MCP also supports wrappers like **LangchainTool** and **CrewaiTool,** which allow for the direct reuse of tools from popular open-source ecosystems.

Pro tip

Use the open-source MCP Toolbox for Databases to easily and securely connect your agents to a large array of popular data sources.

MCP is like a universal adaptor for an agent's data sources and tools



Collaborate with the Agent2Agent protocol

The true power of specialized agents is unlocked when they can collaborate. To enable this, Google champions the Agent2Agent (A2A) protocol, an open standard that ensures the agents you build today can discover, communicate and negotiate with, and securely coordinate actions with other agents—regardless of who built them or what framework they use.

Pro tip

Explore these A2A resources to get started:

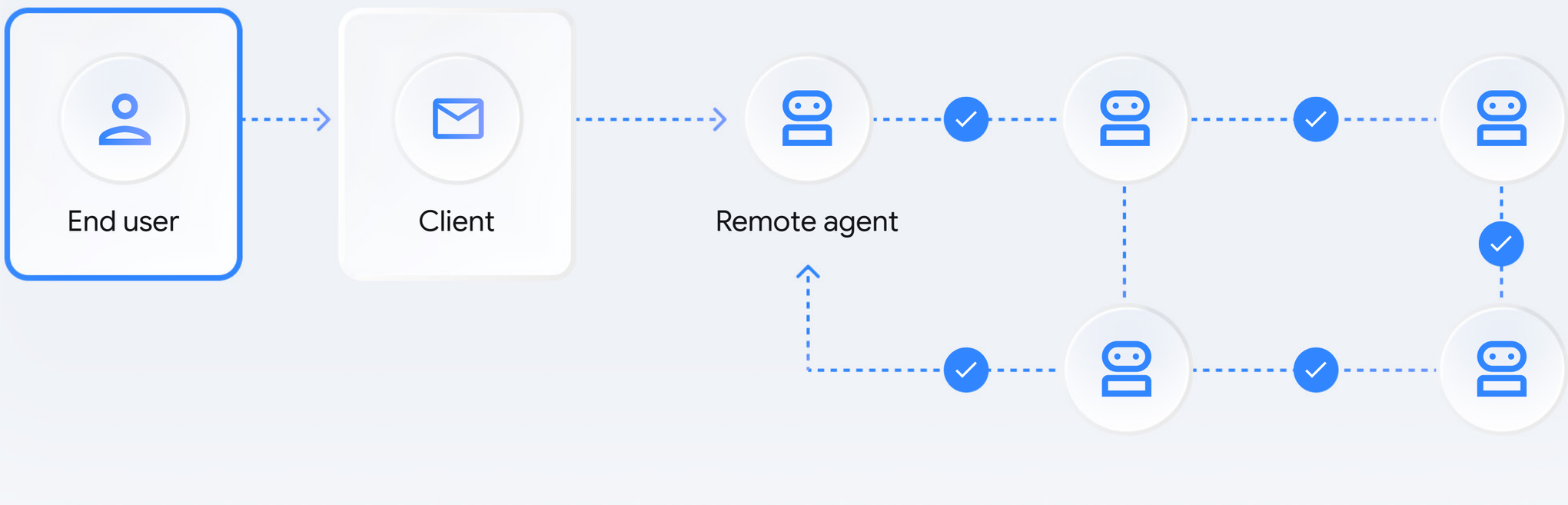
- A2A Project Github org
- A2A protocol docs
- A2A protocol specification

Key concepts of the A2A protocol include:

- **Agent card:** A digital "business card" (typically a JSON file at a well-known endpoint) that advertises an agent's capabilities, endpoint URL, and authentication requirements, enabling discovery by other agents.
- **Task-oriented architecture:** Interactions are framed as "tasks." A client agent sends a task request to a server agent, which processes it and returns a response. An agent can act as both a client and a server.
- **Modality agnostic:** A2A supports text, audio, and video communication, reflecting the evolving, multimodal nature of agent interactions.

ADK agents can natively participate in this ecosystem. They expose a standard HTTP endpoint and an **agent.json** file, allowing them to be discovered and to communicate with any other A2A-compliant agent.

How the A2A protocol works

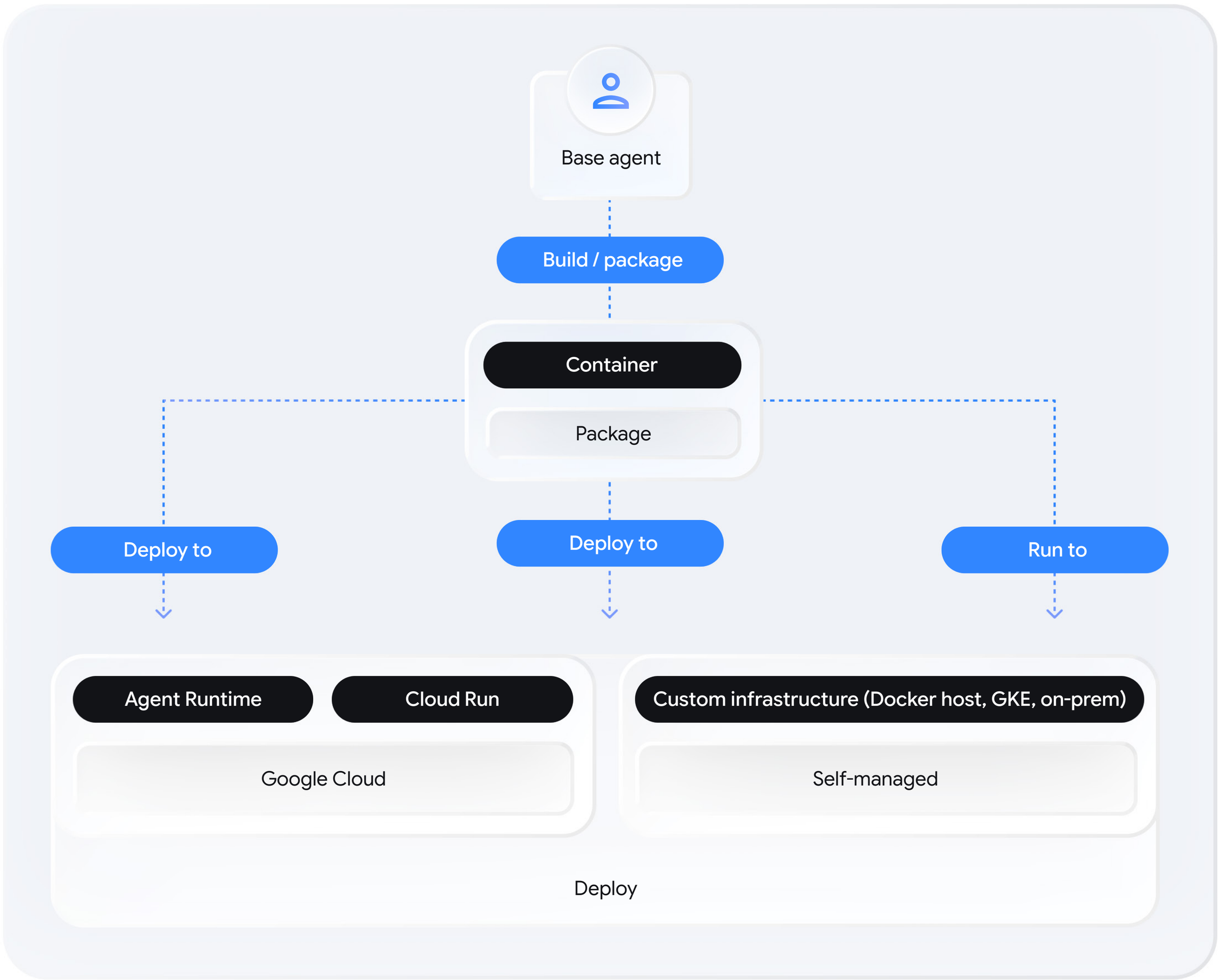


Chapter 5

Deploy to production

Choose your deployment path

ADK is deployment-agnostic. The core agent logic you define in Python is decoupled from the serving infrastructure so you can develop and test locally, then deploy the same agent to various production environments.



The three primary managed deployment targets for ADK agents are:

- **Agent Runtime:** A fully managed auto-scaling service on Google Cloud specifically designed for deploying, managing, and scaling AI agents built with frameworks like ADK. It provides deep integration with Gemini platform ecosystem for MLOps, monitoring, and security. The fastest way to deploy to Agent Runtime is with Agents CLI, an accelerated deployment path to set up a fully configured Google Cloud environment with CI/CD, infrastructure-as-code, and deployment pipelines for your ADK agent.
- **Cloud Run:** A managed compute platform for running your agent as a container-based application. This is an excellent choice for integrating your agent into an existing microservices architecture or for use cases requiring custom container configurations.
- **Google Kubernetes Engine (GKE):** This managed Kubernetes service is the best choice if you have existing Kubernetes-based infrastructure or are prioritizing long-term portability, deep architectural control, and the open-source Kubernetes ecosystem. It provides the most granular control over networking, stateful workloads, and specialized hardware like GPUs and TPUs—ideal if you're building complex, multi-service applications that need to scale.

Agent Runtime is ideal for ADK deployments

For startups using ADK, Agent Runtime is the recommended deployment target. The cost-effective auto-scaling solution provides the easiest and most direct path to a scalable, production-ready agent.

As a fully managed service designed for agentic workloads, it abstracts the underlying infrastructure—freeing your engineers to focus on core agent logic rather than operational overhead.

Core capabilities

- **Automated scalability:** Automatically handles scaling to meet varying user loads.
- **Security and authentication:** Provides integrated identity and access management.
- **Framework agnostic:** Supports agents built with various frameworks, not just ADK.
- **Agent lifecycle management:** Provides APIs for creating, reading, updating, and deleting your deployed agents.

Specialized agentic features

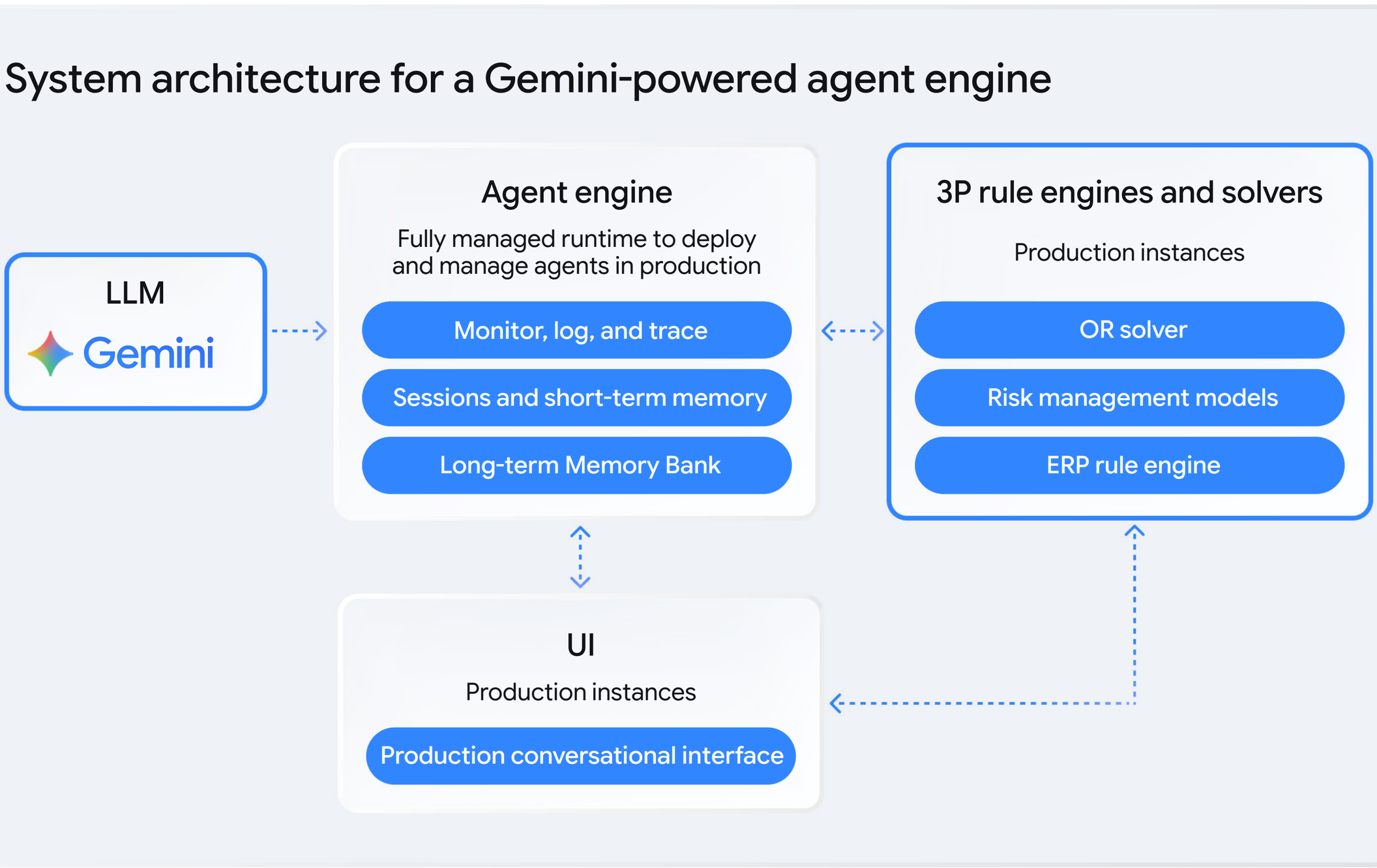
The following components are runtime-agnostic, so will also work with Cloud Run and GKE. They are natively integrated with Agent Runtime to accelerate the build process.

1. Memory Bank
   This managed service dynamically generates and retrieves long-term, personalized memories based on users' conversations.
2. Example Store
   This allows developers to provide and manage few-shot examples to improve and steer agent performance on specific tasks.
3. Agent Identity
   This provides a strongly attested, cryptographic identity for each agent, based on the SPIFFE standard. It means your agent can securely authenticate to MCP servers, cloud resources, endpoints, and other agents, acting either on its own behalf or on behalf of an end user—ultimately creating strict, auditable identity trails across systems.

Put theory into practice

This codelab teaches you how to deploy a containerized agent on Agent Runtime.

Start codelab



Chapter 6

Manage in production with AgentOps

A framework for production-ready agents

Agent Operations (AgentOps) is an operational methodology that addresses the challenges of reliability and responsibility in production. It adapts the principles of DevOps, MLOps, and DataOps to the unique challenges of building, deploying, and managing AI agents across their lifecycle. And it gives you a systematic, automated, and reproducible framework for handling the complexities of non-deterministic, LLM-based systems in production environments.

A robust AgentOps strategy systematizes the development process, providing continuous feedback loops to improve an agent's reliability, safety, and performance across its tools, reasoning capabilities, and underlying models.

Four layers of agent evaluation

Evaluating non-deterministic agentic systems is one of the most complex challenges in modern software engineering. Traditional testing often focuses on lexical correctness, but agent evaluation must address two harder problems:

- **Semantic correctness:** Did the agent understand and helpfully answer the user's intent?
- **Reasoning correctness:** Did the agent follow a logical and efficient path to its conclusion?

Given that a failure at any point can lead to an incorrect outcome, a rigorous, multi-layered evaluation framework is required. The framework is implemented using a combination of ADK for core agent logic and instrumentation, and Agents CLI for production-grade scaffolding, evaluation, deployment, and observability.

Using this comprehensive and practical methodology for agent evaluation (described in detail on the next two pages) moves teams beyond informal "vibe-testing" to a systematic, automated, and reproducible process. By dissecting evaluation into component, trajectory, outcome, and system-level monitoring, it directly addresses the core domains of AgentOps.

It delivers a competitive advantage, too. The rigorous, data-driven, and automated process helps you innovate faster, deploy with confidence, and build agents that are demonstrably safer and more effective.

Layer 01

Component-level evaluation

This layer runs deterministic unit tests, focusing on the predictable, non-LLM components of the agent system. The goal is to verify the lexical correctness of individual building blocks, ensuring that agent failures don't stem from simple bugs in its components.

What to test

- **Tools:** Expected behavior with valid, invalid, and edge-case inputs.
- **Data processing:** Robustness of parsing and serialization functions.
- **API integrations:** Handling of success, error, and timeout conditions.

Implementation

- ADK defines the agent's tools as Python functions (or Java methods). These functions are the direct subjects of component-level testing.
- Agents CLI provides the testing infrastructure. It generates a project with a standard **pytest** environment configured in the **tests/unit/** directory. Developers can immediately write unit tests for their ADK-defined tools and run them via the **make test** command.

Layer 02

Trajectory evaluation

This is the most critical layer for evaluating the agent's reasoning process and procedural correctness within the ReAct cycle. A "trajectory" is the full sequence of Reason, Act, and Observe steps the agent takes to complete a task.

What to test

- **Reason step:** Does the agent correctly assess the goal and current state to form a logical hypothesis for the next step?
- **Act step:** Does it select the correct tool (**Tool Selection**) and correctly extract and format the arguments for that tool (**Parameter Generation**)?
- **Observe step:** Does it correctly integrate the output from the tool to inform the next **Reason** step of the cycle?

Implementation

- ADK's core runtime executes the agent's ReAct loop. It integrates directly with Google Cloud Trace to instrument each **Reason, Act,** and **Observe** step so developers can visualize the entire trajectory, inspect tool inputs and outputs, and examine the model's chain of thought to debug its reasoning.
- Agents CLI automates and scales trajectory evaluation. A **tests/integration/directory** creates a "golden set" of prompts with expected ReAct trajectories. The automated CI/CD pipeline (set up with **agent-starter-pack setup-cicd**) runs these tests on every pull request to prevent regressions. This observability infrastructure captures the trace data emitted by the ADK agent.

Layer 03

Outcome evaluation

This layer evaluates semantic correctness and factual accuracy, assessing the quality of the final user-facing response generated after the ReAct loop has concluded.

What to test

- **Factual accuracy and grounding:** Is the answer correct and verifiably based on information gathered during the **Observe** steps?
- **Helpfulness and tone:** Does the response fully address the user's need in the appropriate style?
- **Completeness:** Does the response contain all necessary information?

Implementation

- ADK's toolset is key for verifying factual accuracy. Developers can create specialized tools or use APIs for grounding verification. These tools, called during the **Act** step, programmatically check if the agent's final answer is supported by the context it retrieved, providing a quantitative measure against hallucination.
- Agents CLI provides the platform for running these evaluations at scale. It integrates with Gemini platform's evaluation service for LLM-as-judge scoring. Its built-in UI playground includes feedback mechanisms that log human ratings directly to BigQuery, enabling high-fidelity HITL evaluation.

Layer 04

System-level monitoring

Evaluation doesn't stop at deployment. Continuously monitoring the agent's live performance in production is critical. The goal here is to track real-world performance and detect operational failures or behavioral drift.

What to test

- Tool failure rates, user feedback scores, trajectory metrics (e.g., number of ReAct cycles per task), and end-to-end latency.

Implementation

- The ADK agent running in production is the source of operational data, emitting events and traces for every live user interaction.
- Agents CLI provides a production-grade observability stack out-of-the-box. It automatically configures OpenTelemetry, a Log Router to BigQuery, and provides templates for Looker Studio dashboards. This allows teams to immediately track agent performance, analyze trends, and debug issues using data from real-world usage without additional setup.

Agents CLI accelerates adoption of AgentOps principles

To prevent regressions and provide continuous, objective feedback on the agent's performance and safety, automated CI/CD pipelines implement the principles of AgentOps. Any change to the agent's code, tools, or prompts triggers a standardized process of building, unit testing, and quantitative evaluation against a predefined dataset.

Agents CLI provides a production-ready reference implementation that addresses common deployment, evaluation, customization, and observability challenges with AI agents. It includes these key components:

Observability and logging (Cloud Trace and Cloud Logging)

Establishes the foundation for monitoring and debugging by integrating with Cloud Trace for in-depth analysis of agent execution traces and Cloud Logging for centralized log management.

Data integration (BigQuery)

Includes foundational components for agents that need to connect to and analyze structured enterprise data using BigQuery.

Infrastructure as Code (Terraform)

Provides reproducible templates to provision and manage the agent's cloud environment, including services like Cloud Run, IAM permissions, and networking.

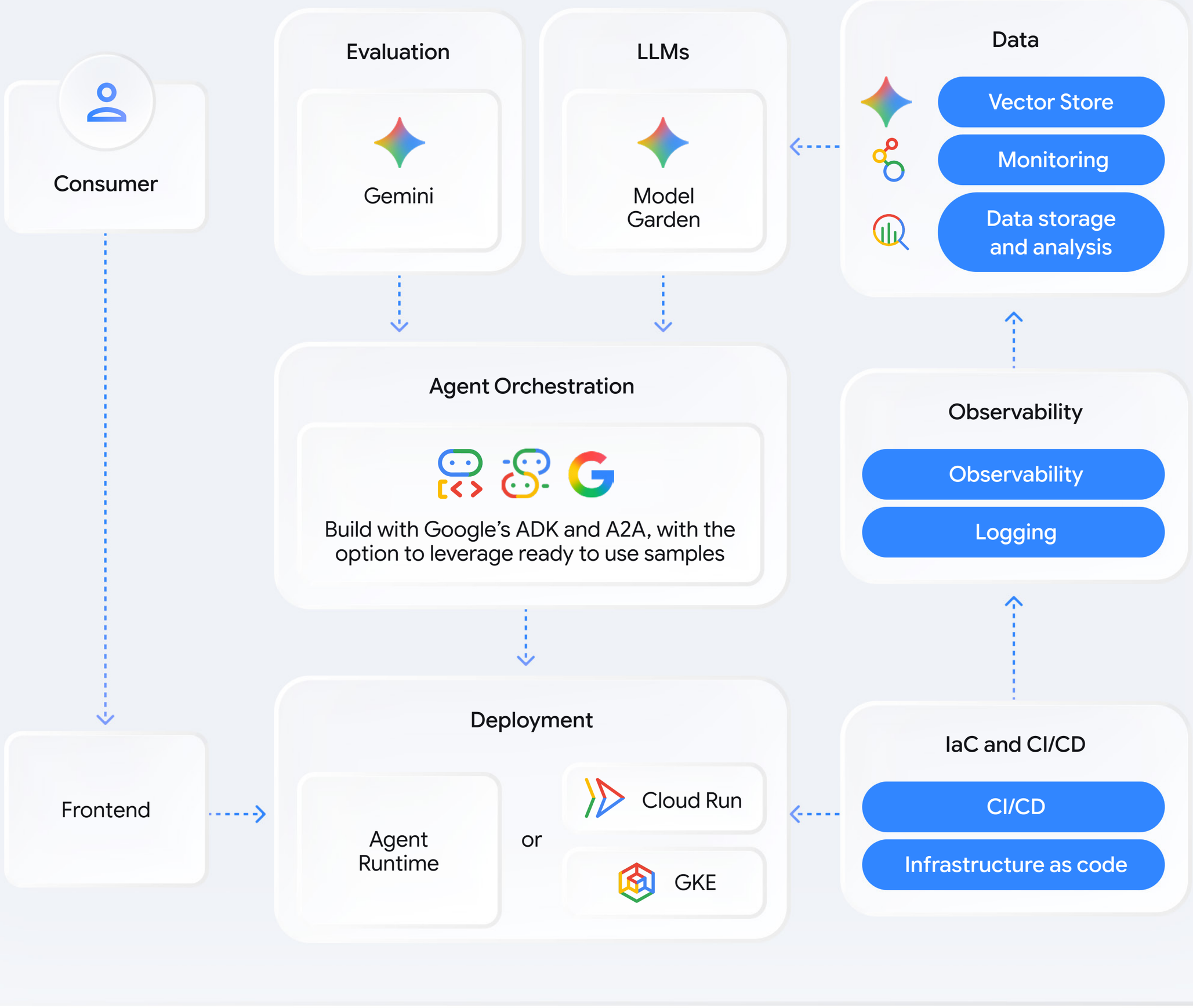
CI/CD pipelines (Cloud Build)

A pre-configured cloudbuild.yaml file automates the build, unit testing, quantitative evaluation, and deployment process, directly implementing the AgentOps CI/CD workflow.

Continuous evaluation (Gemini platform)

Integrates with Gemini platform to run evaluation datasets against agent changes, continuously measuring performance against the key domains discussed previously.

Template architecture for Agents CLI in Gemini



Better together: ADK and Agents CLI

ADK and Agents CLI are engineered to provide a clear separation between an agent's application logic and its operational lifecycle, enabling a robust and scalable development process.

Essentially, ADK is used to write the agent's application code, while Agents CLI provides the production-ready operational baseline to run and manage that code at scale. This separation manifests in a five-step workflow:

1

Bootstrap with Agents CLI

A developer runs a single command to generate a new project containing all necessary operational components, including Terraform files for infrastructure, Cloud Build configurations for CI/CD, and skeleton files for evaluation datasets.

2

Develop with ADK

Inside this structure, the developer uses ADK to write the agent's application logic, implementing custom tools, composing agents, and writing the core instructions.

3

Commit and automate

When code is committed to the source repository, the pre-configured CI/CD pipeline managed by Cloud Build is automatically triggered.

4

Evaluate continuously

The pipeline builds the ADK agent into a container and then executes a quantitative evaluation against a predefined test set, programmatically validating the agent's performance and safety.

5

Deploy confidently

Upon a successful evaluation, the pipeline automatically deploys the new, validated version of the agent to its target production environment.

Securing your AI agent ecosystem

Together, ADK and Agents CLI deliver a defense-in-depth strategy. First, you can implement fine-grained, application-level safety controls with ADK. And, second, Agents CLI automates the deployment of the hardened cloud infrastructure that enforces these controls at scale.

This combined approach addresses key aspects of safety and compliance:

- **Secure infrastructure and access control:** Agents CLI uses Terraform to provision a secure foundation, deploying your agent to environments like Cloud Run and configuring specific IAM roles to enforce the principle of least privilege. The tools you define in ADK then operate within these strict cloud-level permissions—so the agent can't access unauthorized resources even if its own logic is compromised.
- **Input and output guardrails:** Within ADK, you can implement application logic to validate prompts and filter the agent's final outputs. Agents CLI then integrates these guardrails into its CI/CD pipeline, and you can run automated security tests against every code change to check for vulnerabilities before they reach production.
- **Runtime threat interception:** Model Armor helps ensure that your AI agents follow your security and governance standards without having to implement filtering logic directly within the agents. You can enforce content security guardrails consistently across all agents governed by Agent Gateway—helping to mitigate risks such as prompt injection, jailbreak attempts, leakage of sensitive information, and generation of harmful content.
- **Auditing and monitoring:** The detailed observability in ADK creates a granular trace of every thought and tool call the agent makes. Agents CLI operationalizes this by automatically configuring log sinks that route this data to BigQuery for long-term, secure storage. This creates the durable audit trail necessary for compliance reviews and incident response.

With both ADK and Agents CLI operating within the larger Google Cloud ecosystem, you gain a formidable security posture built on a secure-by-design foundation—with integrated controls designed to defend any workload.

Conclusion

Today's AI startups are in a high-stakes, high-speed race to bring the next big thing to market. The multi-agent systems you're building—which promise to transform everything from daily workflows to entire industries—rely on deep reasoning, massive context windows, and complex tool integrations.

By using a code-first framework like ADK and the operational principles in this guide, you can iterate and innovate faster. Plus, you can scale with confidence, without compromising on safety or security.

Beyond ADK, Google Cloud delivers the complete AI startup engine designed to power your journey from idea to IPO. Our platform provides a vertically integrated yet radically open stack that allows you to inherit the same resilient foundation that powers Search and YouTube. We move beyond providing raw compute to offering an "agent-ready" stack where orchestration, grounding, and security are integrated by default.

The platform is the foundation; your unique vision and the principles outlined in this guide are the blueprint. Together, they form the basis for building the next generation of intelligent systems that will drive your startup forward.

Ready to turn your AI vision into reality?

We’re here to help.

Learn how to build more generative AI applications with on-demand Startup School sessions.

Start now

Stay connected and get all our latest updates by subscribing to the Google Cloud Startup Newsletter.

Subscribe

Questions? Contact our Startups team.

Get in touch

Google for Startups