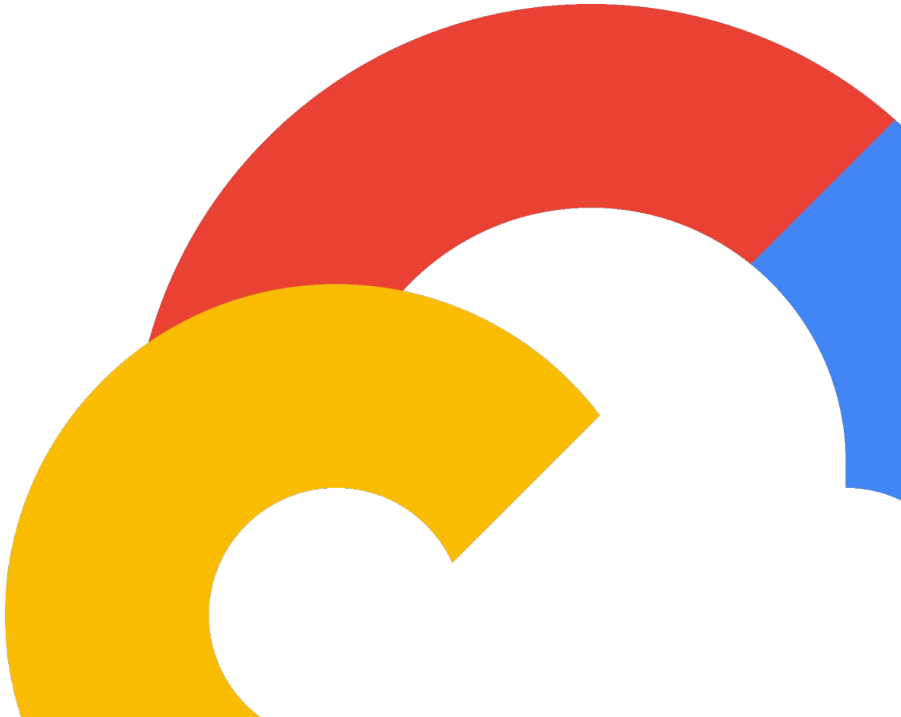
AI Agent

Oct 2024

Hangsik Shin

Google Cloud



Agenda

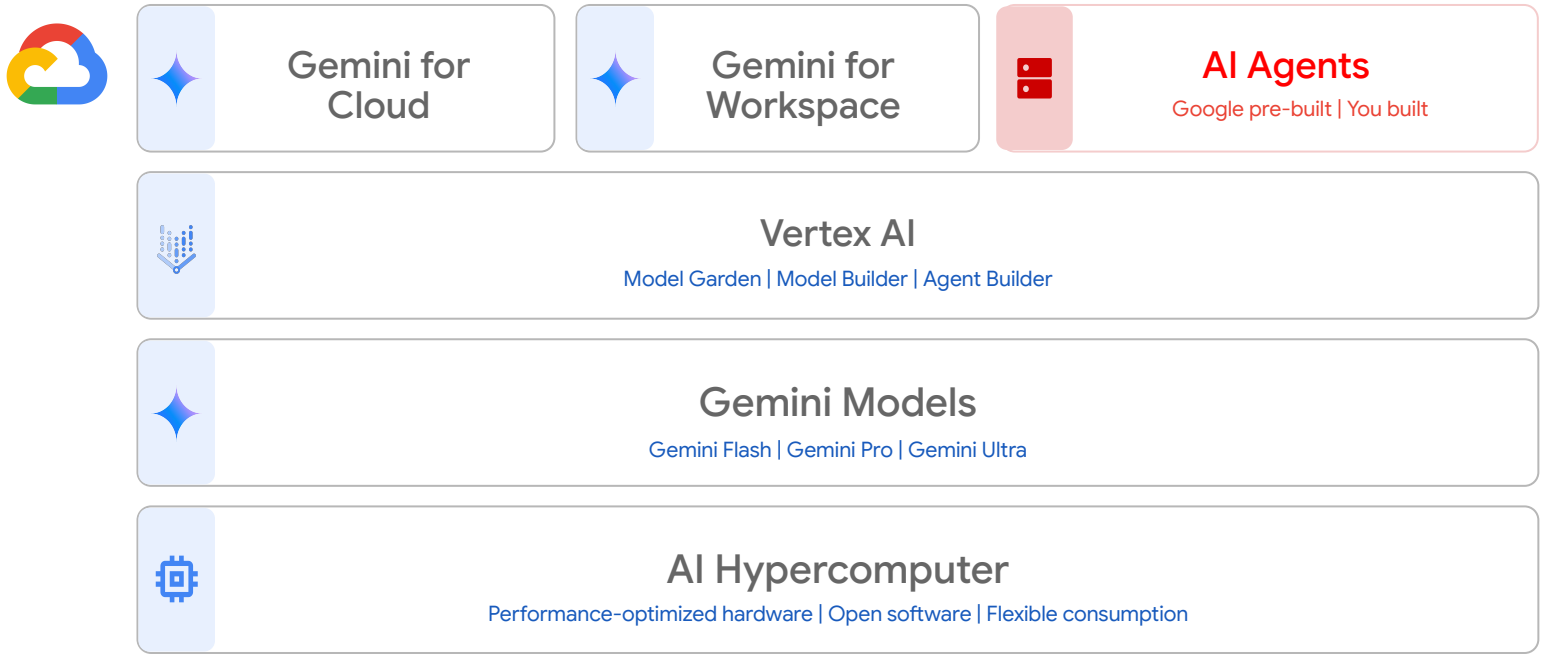
시간	Subject	Description	Time
10:00 ~ 12:00	전체적인 workshop 설명	환경셋팅, Colab 환경. AutoML training 실행.	30분
	Vertex AI Overview	Vertex AI Console. Dataset 설정	30분
	Pipeline 설명	Kubeflow, Experiments, Metadata	1시간
13:30 ~ 14:30	AutoML Pipeline 설명	AutoML, GCPC. Evaluation Log.	1시간.
14:30~16:00	Vertex AI Search and Grounding services	Search and grounding	1시간 30분
16:00~17:30	Vertex AI Agent를 활용한 Conversation Chatbot 개발	Reasoning engine / RAG engine	1시간 30분

Agenda

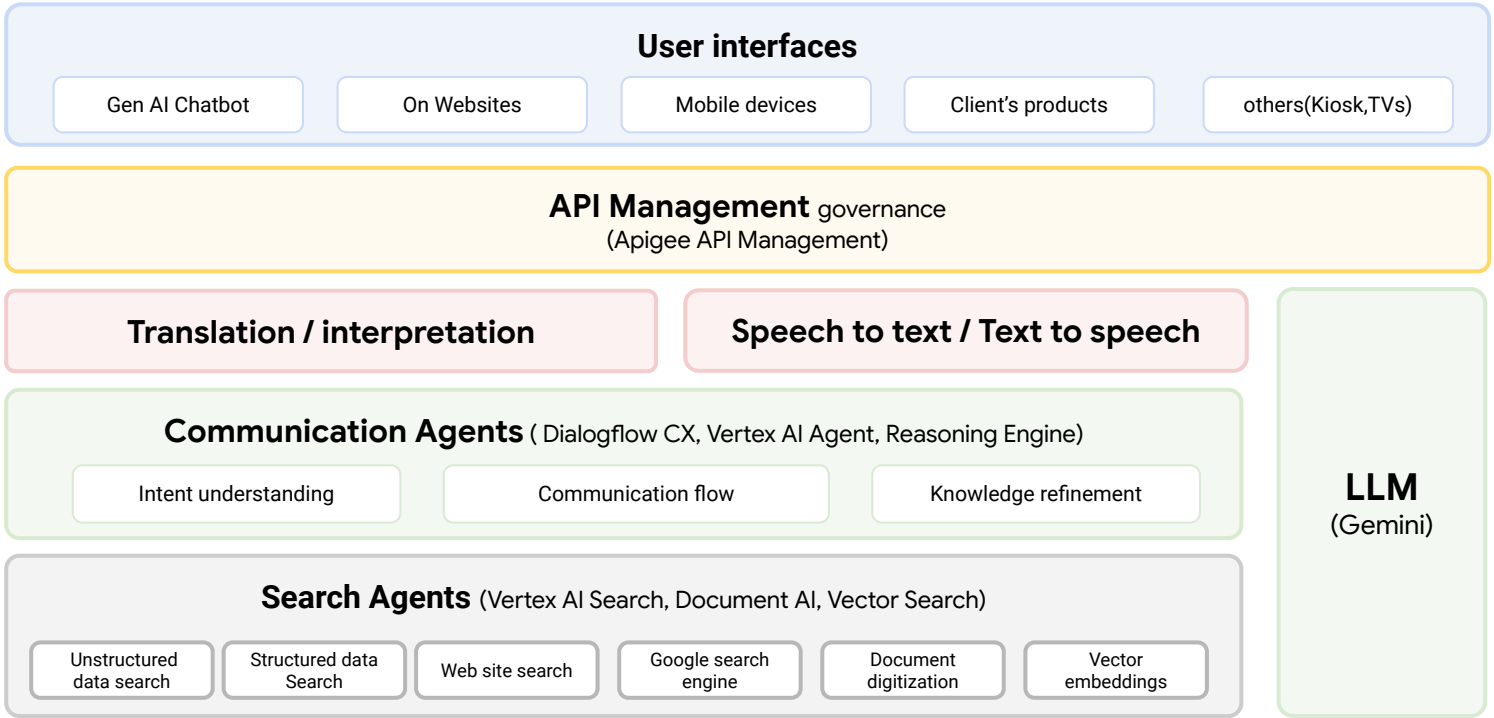
- AI Agent Overview
- AI Agent System Architecture
- Search(RAG)
- AI Agent Framework
- Reasoning Engine
- Function calling
- LlamaIndex on Vertex AI

AI Agent Overview

AI Agents framework for Enterprise scale

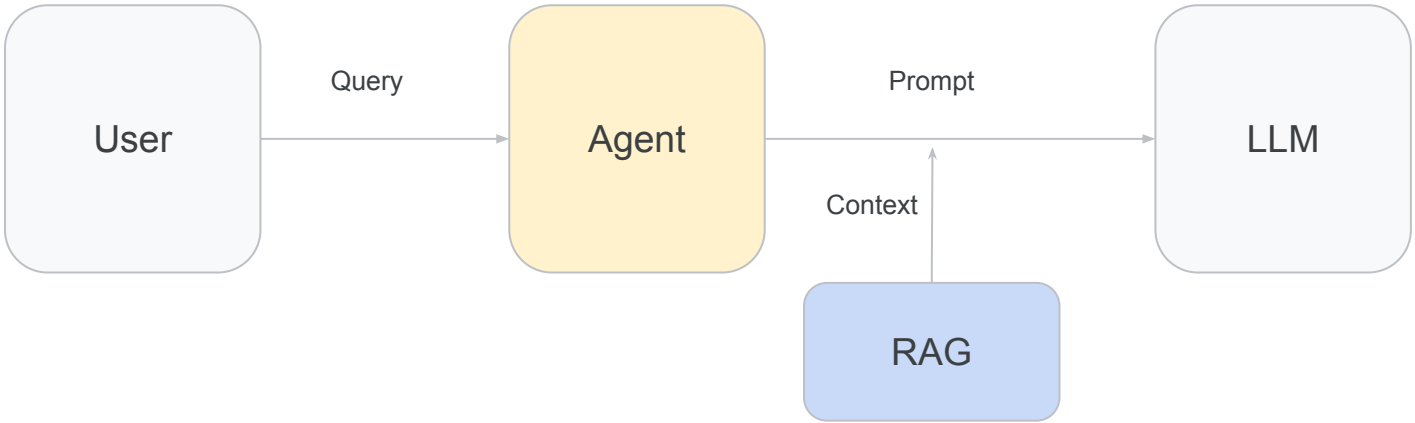


AI Agent Stack

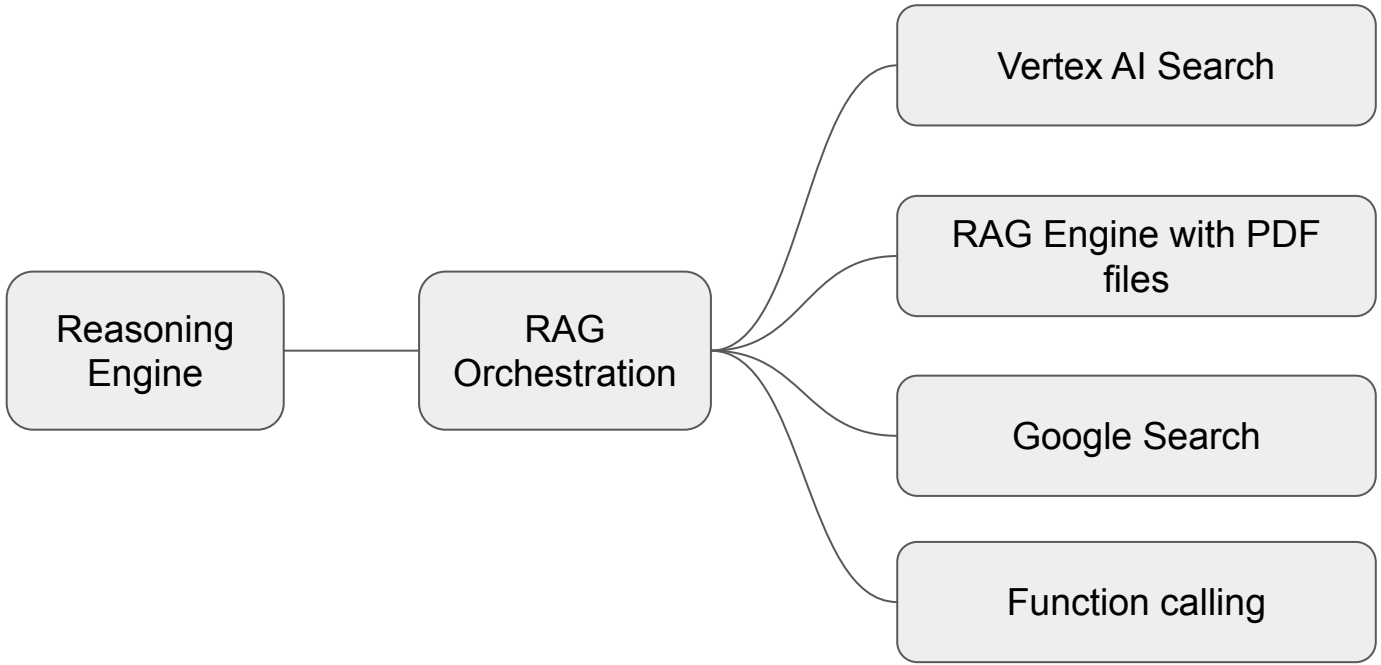


AI Applications → AI Agents (Assistants)

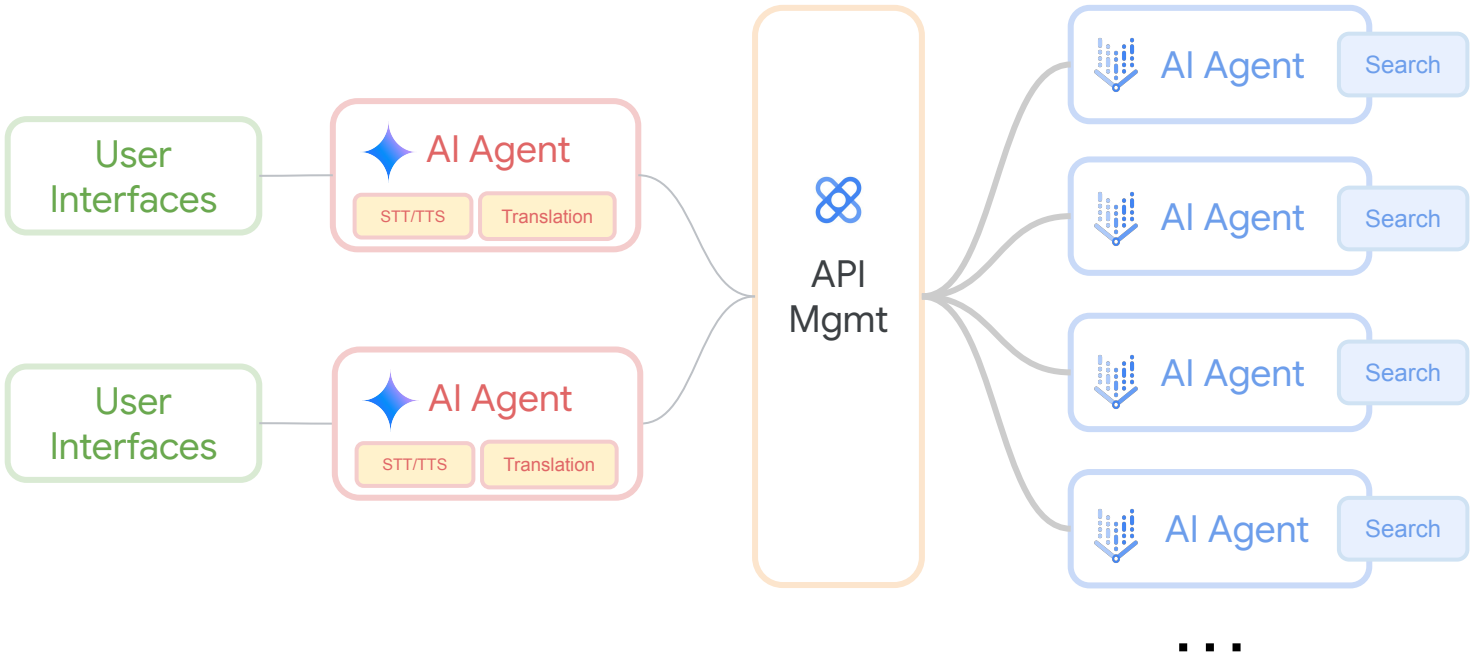
AI applications, also known as AI Agents, are designed to facilitate the delivery of queries and orchestrate the **interaction between RAG and LLMs to generate responses** to user inquiries. These AI Agents play a **crucial role in coordinating call flow and invoking tools capable of executing specific tasks** aligned with the user's query.



Architecture



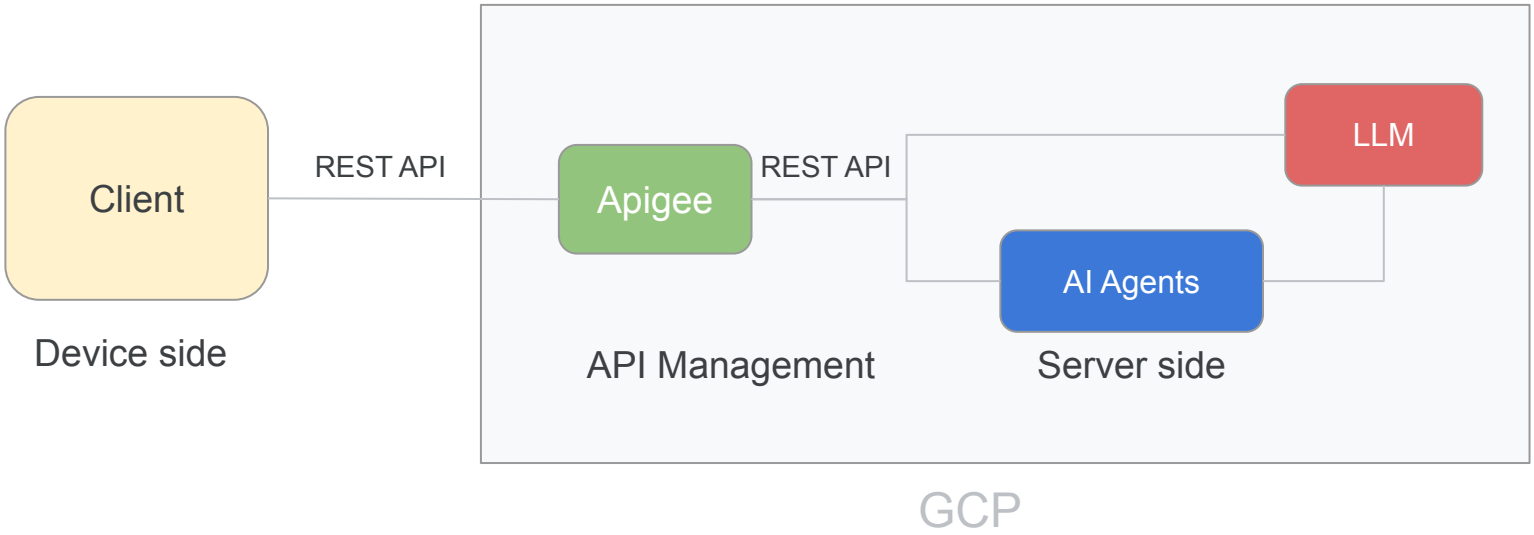
Enterprise AI Agent architecture



AI Agent System Architecture

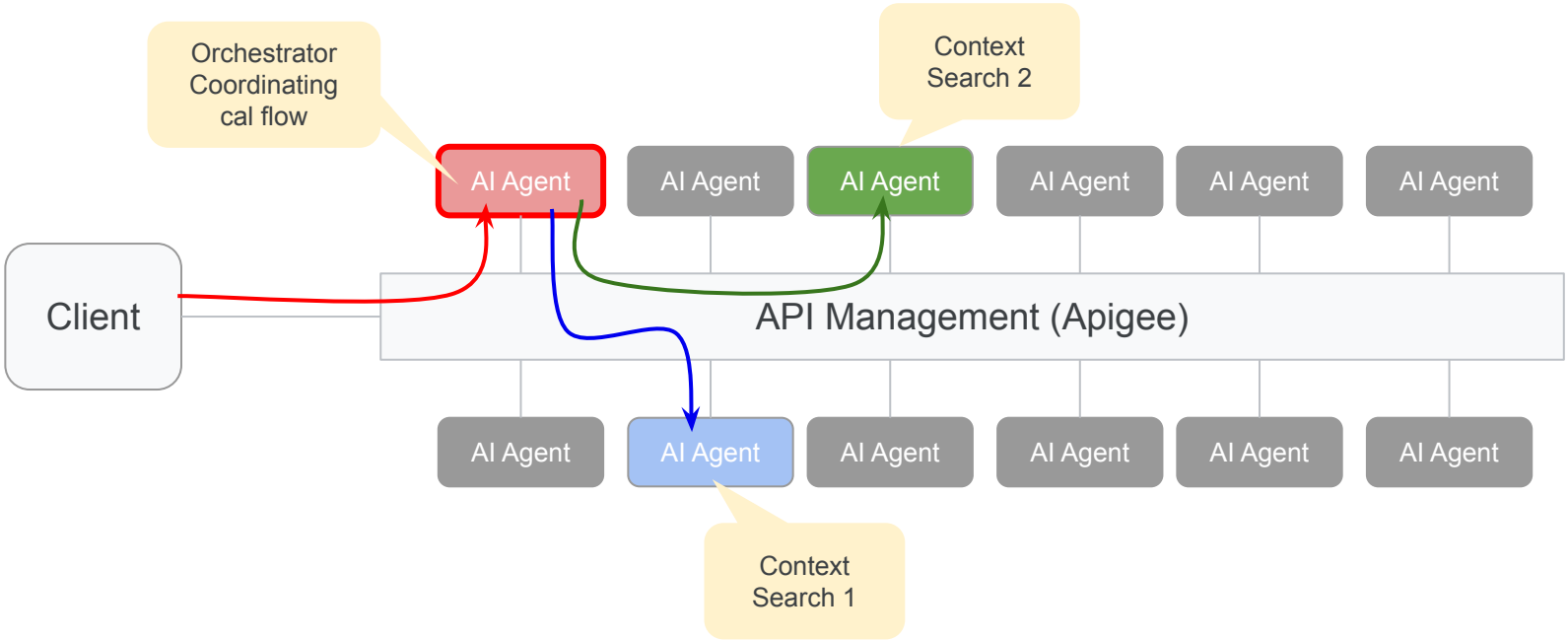
System architecture

Application Programming Interface (API) management (specifically, Apigee) serves as a gateway for receiving requests from various devices and acts as an intermediary to forward those requests to designated AI Agents. Subsequently, the AI Agents process the requests and generate responses, which are then returned to the devices through Apigee responses.



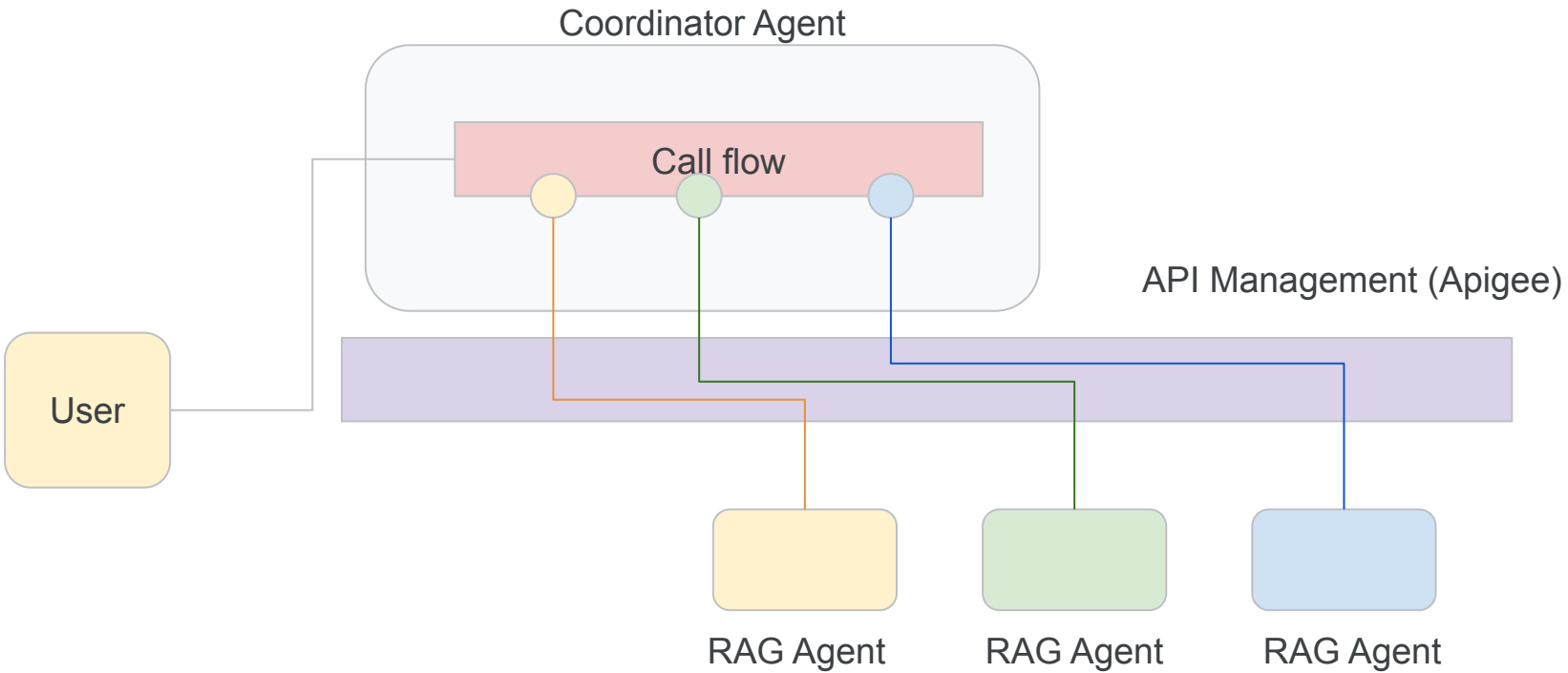
AI Agents Call Flow with Apigee

Within complex AI systems, certain agents assume the role of orchestrators, tasked with coordinating queries and delegating responsibilities to other agents. Other agents facilitate efficient information retrieval and task execution by utilizing API management protocols to communicate with the orchestrators.



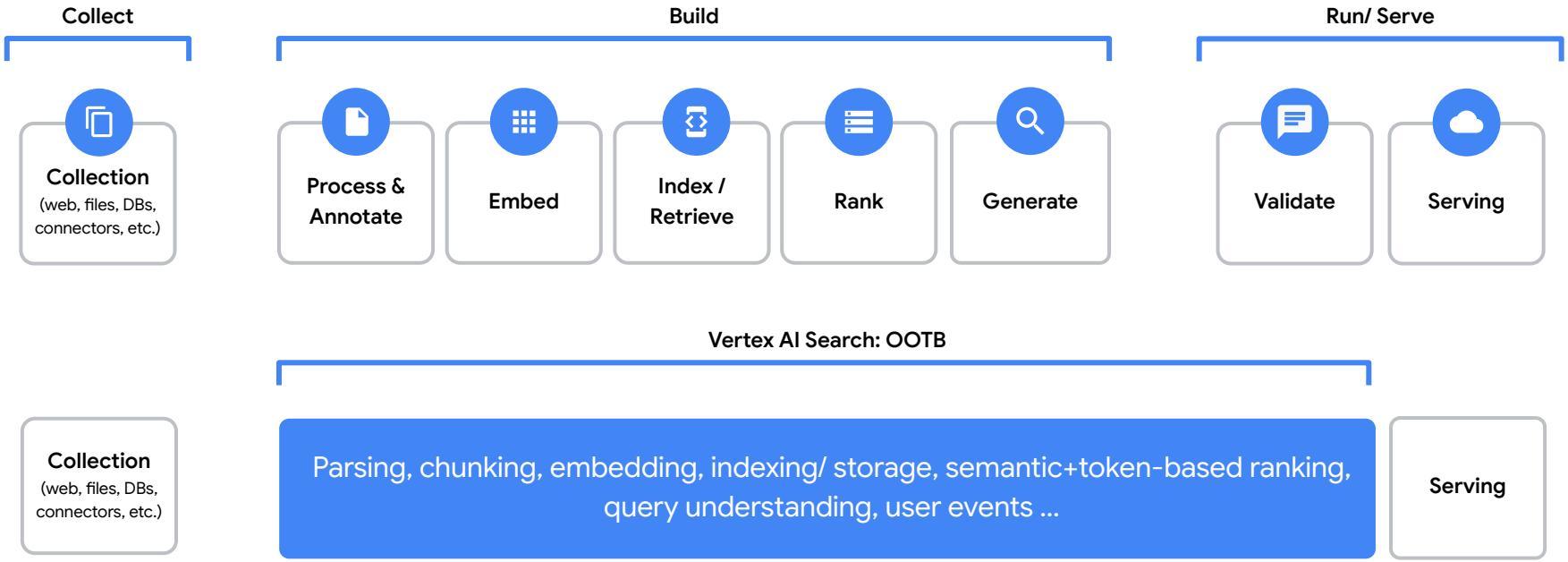
AI Agent System Architecture

Detailed Call Flow



Search(RAG)

With OOTB Vertex AI Search we simplified the process of building RAG / Search applications into a few clicks



Enterprise edition features

Enterprise edition features add functionality at the app level. Enterprise tier features incur additional cost. For more information, see Pricing. Turning on Enterprise edition features is required for the following features:

- Extractive answers.
- Extractive segments.
- Search tuning.

In addition, turning on Enterprise edition features is required for website search. That is, Enterprise edition features are required for any Vertex AI Search app that uses website data (basic and advanced website indexing). This setting can't be turned off for apps that are attached to website data stores.

If you turn off Enterprise edition for an existing app, any features in your app that require Enterprise edition will no longer work.

Advanced LLM features

Advanced LLM features add generative AI functionality at the app level. Advanced LLM features incur additional cost. For more information, see Pricing. Turning on advanced LLM features is required for the following features:

- Search summarization. For website search, this also requires advanced website indexing.
- Search with follow-ups. For website search, this also requires advanced website indexing.
- Search with answers and follow-ups (answer method). For website search, this also requires advanced website indexing.

If you turn off advanced LLM features for an existing app, any functionality in your app that requires advanced LLM features will no longer work.

Parse documents

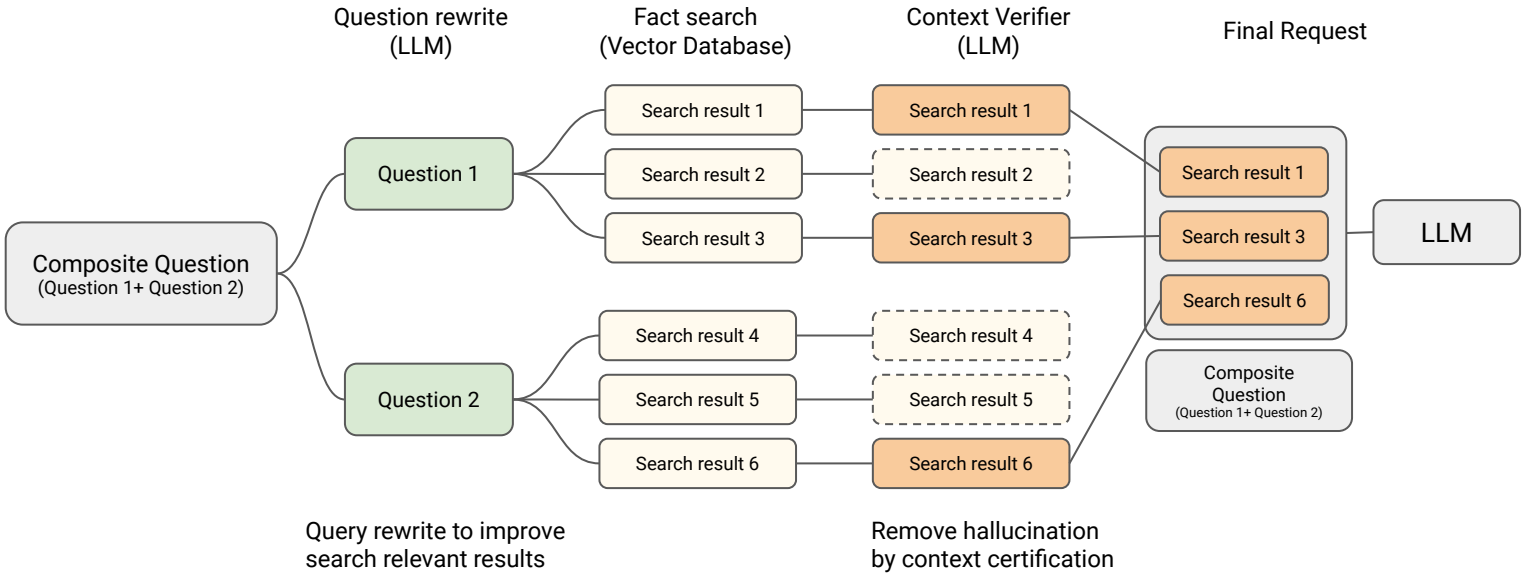
- **Specify parser type.** You can specify the type of parsing to apply depending on file type:
  - **Digital parser.** The digital parser is on by default for all file types unless a different parser type is specified. The digital parser processes ingested documents if no other default parser is specified for the data store or if the specified parser doesn't support the file type of an ingested document.
  - **OCR parsing for PDFs**. If you plan to upload scanned PDFs or PDFs with text inside images, you can turn on the OCR parser to improve PDF indexing. See the OCR parser for PDFs section of this document.
  - **Layout parser.** Turn on the layout parser for HTML, PDF, or DOCX files if you plan to use Vertex AI Search for RAG. See Chunk documents for RAG for information about this parser and how to turn it on.
- **Bring your own parsed document.** (Preview with allowlist) If you've already parsed your unstructured documents, you can import that pre-parsed content into Vertex AI Search. See Bring your own parsed document.

Parser availability comparison

File type	Digital parser	**OCR parser** The OCR processor can parse a maximum of 500 pages per PDF file. For longer PDFs, the OCR processor parses the first 500 pages and the default parser parses the rest.	**Layout parser** The layout parser supports a maximum PDF file size of 40 MB.
HTML	Detects paragraph elements	N/A	Detects paragraph, table, list, title, and heading elements
PDF	Detects paragraph (digital text) elements	Detects paragraph elements	Detects paragraph, table, title, and heading elements
DOCX (Preview)	Detects paragraph elements	N/A	Detects paragraph, table, list, title, heading elements
PPTX (Preview)	Detects paragraph elements	N/A	Detects paragraph, table, list, title, heading elements
TXT	Detects paragraph elements	N/A	Detects paragraph, table, title, heading elements
XLSX (Preview)	Detects paragraph elements	N/A	Detects paragraph, table, title, heading elements

Advance RAG - Architecture in detail

For complex questions, this architecture shows how to understand and expand a query and how to increase search results to retrieve more relevant content, then verify the result to mitigate hallucination.



AI Agent Framework

AI Agent

AI agents can range in complexity from **simple chatbots to advanced AI assistants**. They use machine learning and natural language processing (NLP) to perform tasks.

- An AI agent is a software program that can use artificial intelligence (AI) to perform tasks on behalf of a user or system. AI agents can:
    - **Collect data:**
        - AI agents can interact with their environment and gather data.
    - **Make decisions:**
        - AI agents can use data to make decisions and perform actions to achieve goals.
    - **Learn and adapt:**
        - AI agents can improve their performance over time through self-learning.
    - **Communicate**:
        - AI agents can communicate with other agents or humans when needed.

AI Agent implementation

There are three potential methods to implement an AI Agent, including the use of RAG to extract relevant context from data sources.

Probabilistic	Probabilistic	Deterministic
Function calling	Reasoning engine	Reasoning engine with LangGraph
Source	Source	Source

Function calling vs Reasoning Engine

- If you were to use Gemini and Function Calling on their own without a reasoning layer, you would **need to handle the process of calling functions and APIs** in your application code, and you would need to **implement retries and additional logic** to ensure that your function calling code is resilient to failures and malformed requests.
- When you define your function, it's **important to include comments that fully and clearly describe the function's parameters**, what the function does, and what the function returns. This information is used by the model to determine which function to use.
- Reasoning Engine uses more simplified and streamlined processes compared to the Function calling.

```python
def get_exchange_rate(
    currency_from: str = "USD",
    currency_to: str = "EUR",
    currency_date: str = "latest",
):
    """Retrieves the exchange rate between two currencies on a specified date.

    Uses the Frankfurter API (https://api.frankfurter.app/) to obtain
    exchange rate data.

    Args:
        currency_from: The base currency (3-letter currency code).
            Defaults to "USD" (US Dollar).
        currency_to: The target currency (3-letter currency code).
            Defaults to "EUR" (Euro).
        currency_date: The date for which to retrieve the exchange rate.
            Defaults to "latest" for the most recent exchange rate data.
            Can be specified in YYYY-MM-DD format for historical rates.

    Returns:
        dict: A dictionary containing the exchange rate information.
            Example: {"amount": 1.0, "base": "USD", "date": "2023-11-24",
                "rates": {"EUR": 0.95534}}
    """
    import requests
    response = requests.get(
        f"https://api.frankfurter.app/{currency_date}",
        params={"from": currency_from, "to": currency_to},
    )
    return response.json()
```

Types of AI Agent

Vertex AI Agent

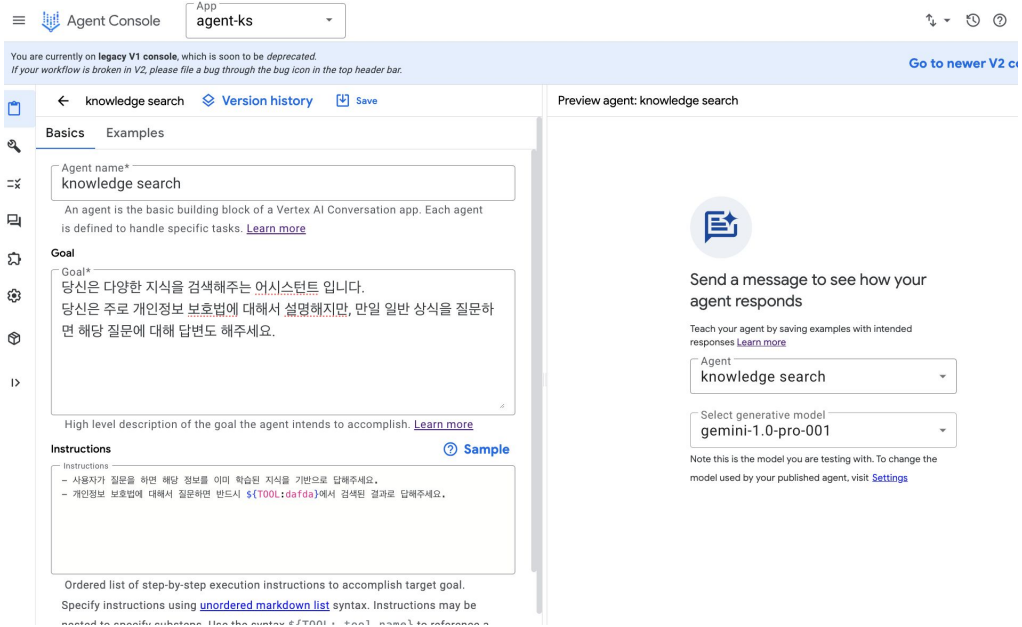
Agent API
(Private preview)

Reasoning engine
(LangChain on Vertex AI)

	Vertex AI Agent	Reasoning engine(Function calling, LangGraph)
Functionality	Designed to interact with users **through natural language**, providing responses to queries or initiating **conversations**	Primarily used to process information, **apply logic rules, and make decisions** based on the given data and user-defined tools
Focusing areas	Defining conversation flows, user intents, and response generation	Defining the logic, tools, and data sources that the agent can access to make informed decisions.
Implementation	**UI based implementation** process by using natural language. built using various components like Dialogflow,	Implemented using frameworks like **LangChain on Vertex AI,** allowing developers to define complex reasoning logic with custom functions and external data access.
Development	No code, less code	Full code

Vertex AI Agent

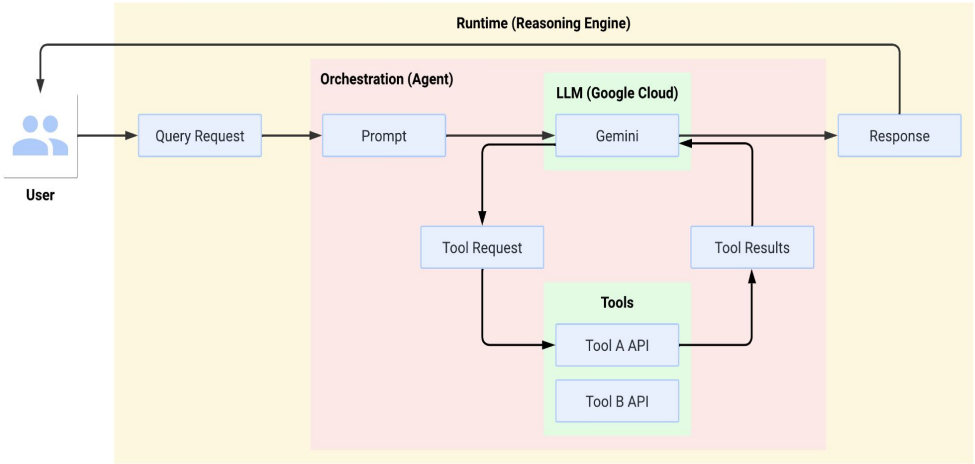
Vertex AI Agents is a new natural language understanding **platform built on large language models (LLMs)**. It makes it **easy to design and integrate a conversational user interface** into your mobile app, web application, device, bot, interactive voice response system.



- **Natural language understanding**: Includes out-of-the-box capabilities for understanding synonyms, correcting spellings, and auto-suggesting searches
- **Generative AI:** Includes generative AI-powered summarization and conversational search for unstructured documents
- **No-code conversational AI agents**: Allows users to design, deploy, and manage agents using natural language

Reasoning engine

Reasoning Engine, also known as LangChain on Vertex AI, is **a managed service that helps users build and deploy agent reasoning frameworks**. Reasoning Engine can help streamline the development and deployment of intelligent applications in enterprise environments.



- **Flexibility**
  - Users can choose how much reasoning to delegate to the large language model (LLM) and how much to handle with custom code.
- **Integration**
  - Reasoning Engine integrates with the Python SDK for the Gemini model in Vertex AI.
- **Compatibility**
  - Reasoning Engine is compatible with LangChain, LlamaIndex, or other Python frameworks.
- **Deployment**
  - Reasoning Engine provides a managed runtime that includes security, privacy, observability, and scalability.
- **Scalability**
  - Users can productionize and scale their application with an API call.

Reasoning Engine

Reasoning Engine

To build effective Generative AI applications, it is **key to enable LLMs to interact with external systems**. This makes **models data-aware and agentic**, meaning they can **understand, reason, and use data to take action in a meaningful way**. The external systems could be public data corpus, private knowledge repositories, databases, applications, APIs, or access to the public internet via Google Search.

Here are a few patterns where LLMs can be augmented with other systems:

- Convert natural language to SQL, executing the SQL on database, analyze and present the results
- Calling an external webhook or API based on the user query
- Synthesize outputs from multiple models, or chain the models in a specific order

Use cases of Reasoning Engine

- Build generative AI applications by connecting to **public APIs**
- Build generative AI applications by connecting to **databases**
- Build generative AI applications with **OSS frameworks**
- Debugging and optimizing generative AI applications

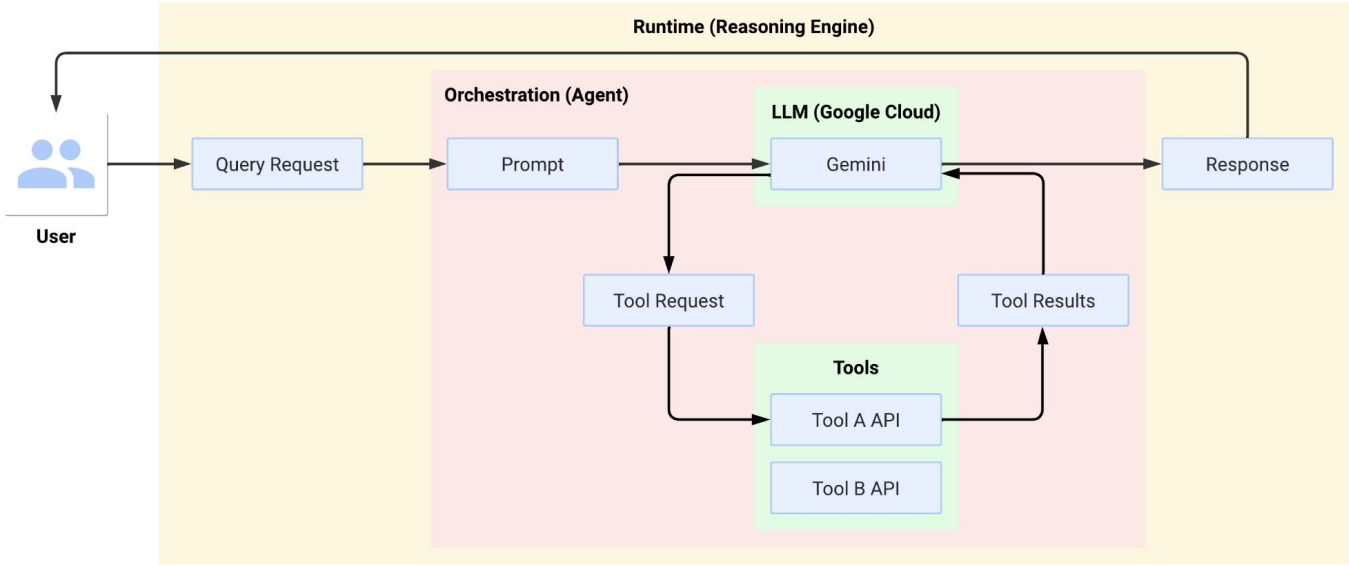
System components

Building and deploying a custom generative AI application using OSS LangChain and Vertex AI consists of four components:

- **LLM**
  - You can choose to define a set of tools that communicates with external APIs and provide them to the model. While processing a query, the model delegates certain tasks to the tools. This implies one or more model calls to foundation or fine-tuned models.
- **Tool**
  - You can choose to define a set of tools that communicates with external APIs (for example, a database) and provide them to the model. While processing a query, the model can delegate certain tasks to the tools.
- **Orchestration framework**
  - LangChain on Vertex AI lets you use the LangChain orchestration framework in Vertex AI. Use LangChain to decide how deterministic your application should be.
- **Managed runtime**
  - LangChain on Vertex AI lets you deploy your application to a Reasoning Engine managed runtime. This runtime is a Vertex AI service that has all the benefits of Vertex AI integration: security, privacy, observability, and scalability.

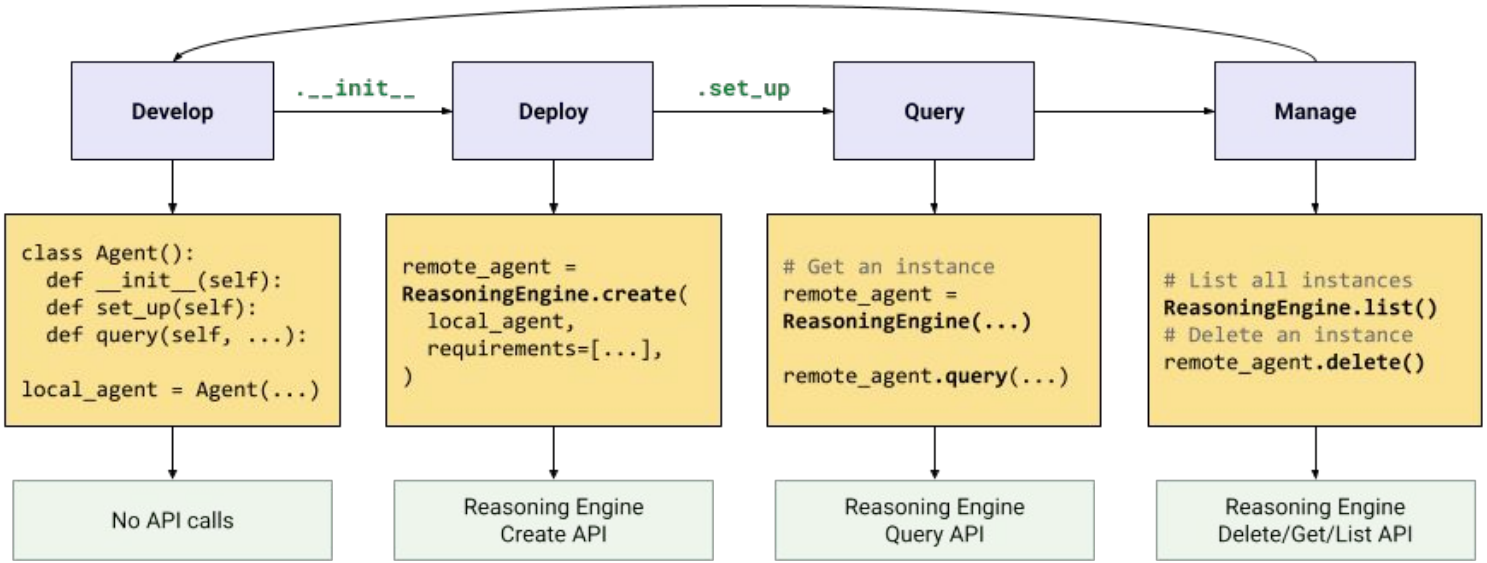
System flow at runtime

When the user makes a query, the defined agent formats it into a prompt for the LLM. The LLM processes the prompt and determines whether it wants to use any of the tools.



Create and deploy a generative AI application

The workflow for building a generative AI applications is:



Multiple tools

Add more tools to conduct different tasks, but don’t make the function description clear and accurate to find a function corresponding to a query.

```
agent = reasoning_engines.LangchainAgent(
    model=model,
    tools=[
        get_exchange_rate,          # Optional (Python function)
        grounded_search_tool,       # Optional (Grounding Tool)
        movie_search_tool,          # Optional (Langchain Tool)
        generate_and_execute_code, # Optional (Vertex Extension)
    ],
)

agent.query(input="When is the next total solar eclipse in US?")
```

Forced function calling

Using these Function Calling modes, you can configure the model to behave in one of the following ways:

- **AUTO mode** : Allow the model to choose whether to predict a function call or natural language response
- **ANY mode** : Force the model to predict a function call on one function or a set of functions
- **NONE mode** : Disable function calling and return a natural language response as if no functions or tools were defined

```
from vertexai.preview.generative_models import ToolConfig

agent = reasoning_engines.LangchainAgent(
    model="gemini-1.5-pro",
    tools=[search_arxiv, get_exchange_rate],
    model_tool_kwargs={
        "tool_config": {  # Specify the tool configuration here.
            "function_calling_config": {
                "mode": ToolConfig.FunctionCallingConfig.Mode.ANY,
                "allowed_function_names": ["search_arxiv", "get_exchange_rate"],
            },
        },
    },
)

agent.query(
    input="Explain the Schrodinger equation in a few sentences",
)
```

Store chat history

To track chat messages and append them to a database, define a get_session_history function and pass it in when you create the agent. This function should take in a session_id and return a BaseChatMessageHistory object.

```
def get_session_history(session_id: str):
    from langchain_google_firestore import FirestoreChatMessageHistory
    from google.cloud import firestore

    client = firestore.Client(project="PROJECT_ID")
    return FirestoreChatMessageHistory(
        client=client,
        session_id=session_id,
        collection="TABLE_NAME",
        encode_message=False,
    )
```

```
agent = reasoning_engines.LangchainAgent(
    model=model,
    chat_history=get_session_history,  # <- new
)
```

```
agent.query(
    input="What is the exchange rate from US dollars to Swedish currency?",
    config={"configurable": {"session_id": "SESSION_ID"}},
)
```

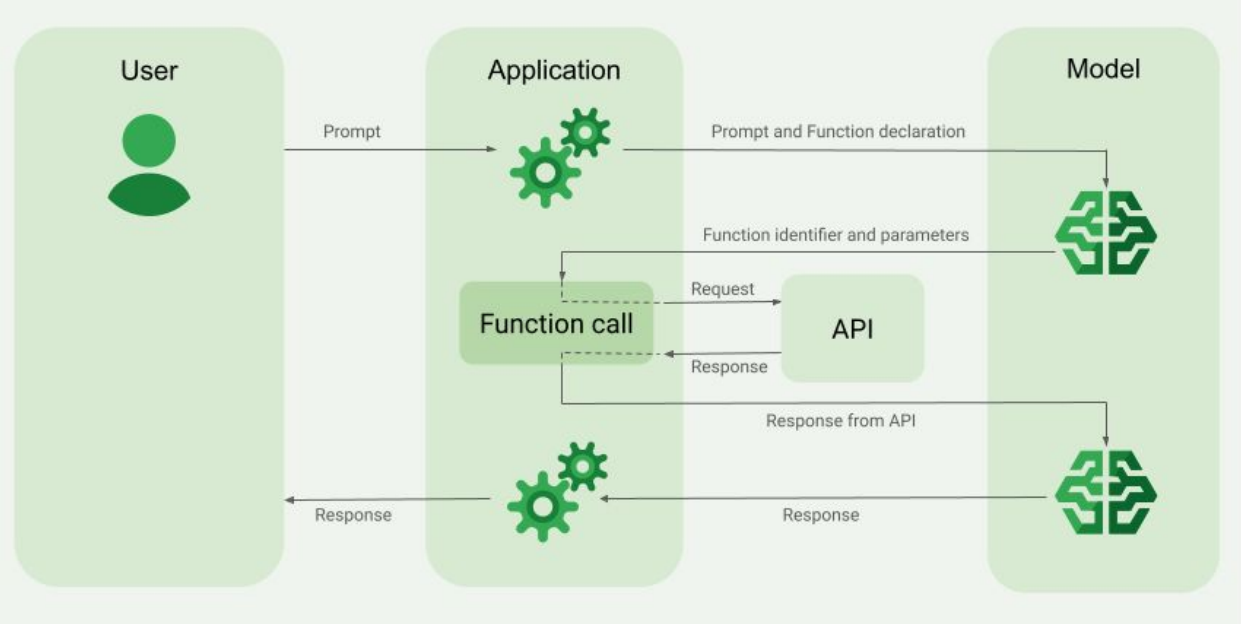
Function calling

Function calling overview

- Large Language Models (LLMs) are powerful at solving many types of problems. However, they are constrained by the following limitations:
  - They are frozen after training, leading to stale knowledge.
  - They can't query or modify external data.
- Function calling can address these shortcomings. **Function calling is sometimes referred to as tool use because it allows the model to use external tools such as APIs and functions**.
- While processing a prompt, the model can choose to delegate certain data processing tasks to the functions that you identify.
- The model does not call the functions directly. Instead, **the model provides structured data output that includes the function to call and parameter values to use**.
- You can then provide the API output back to the model, allowing it to complete its response to the prompt.

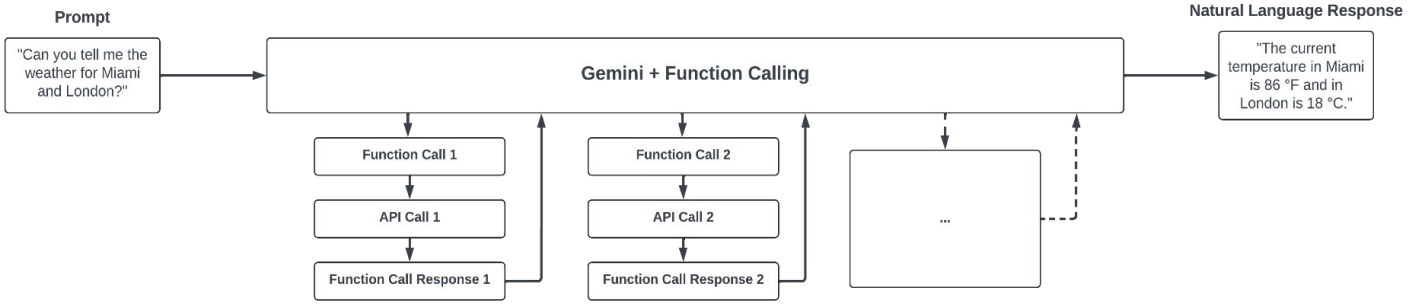
Detailed Call Flow

Use cases of function calling : [Link]

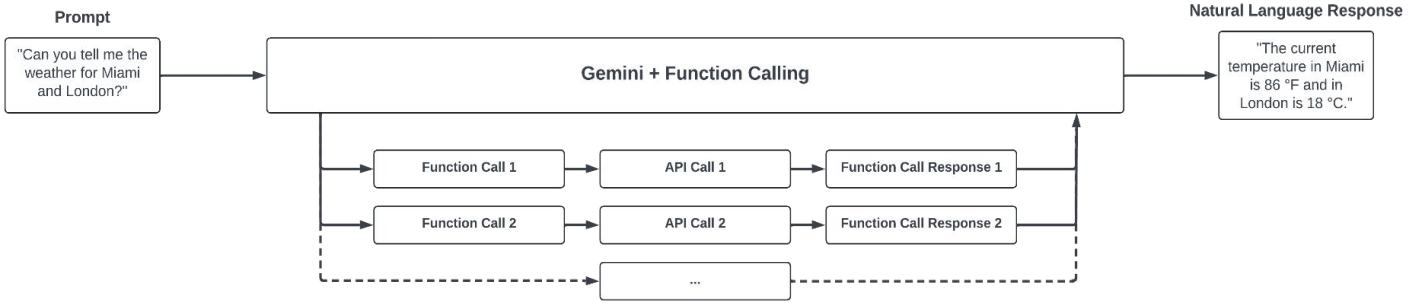


Parallel function calling

Without Parallel Function Calling



With Parallel Function Calling



Thank you!