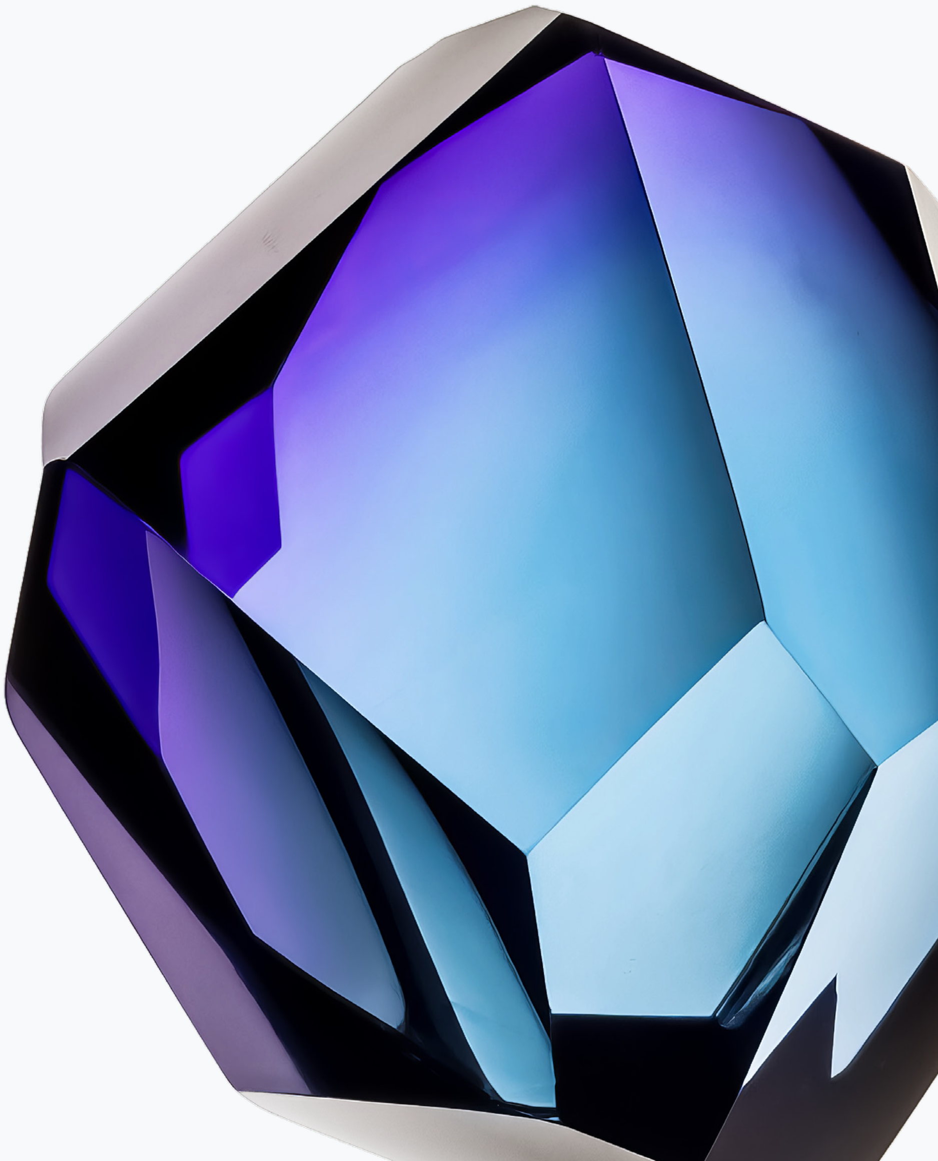
Agents

Authors: Julia Wiesinger, Patrick Marlow
and Vladimir Vuskovic



Acknowledgements

Reviewers and Contributors

Evan Huang

Emily Xue

Olcan Sercinoglu

Sebastian Riedel

Satinder Baveja

Antonio Gulli

Anant Nawalgaria

Curators and Editors

Antonio Gulli

Anant Nawalgaria

Grace Mollison

Technical Writer

Joey Haymaker

Designer

Michael Lanning

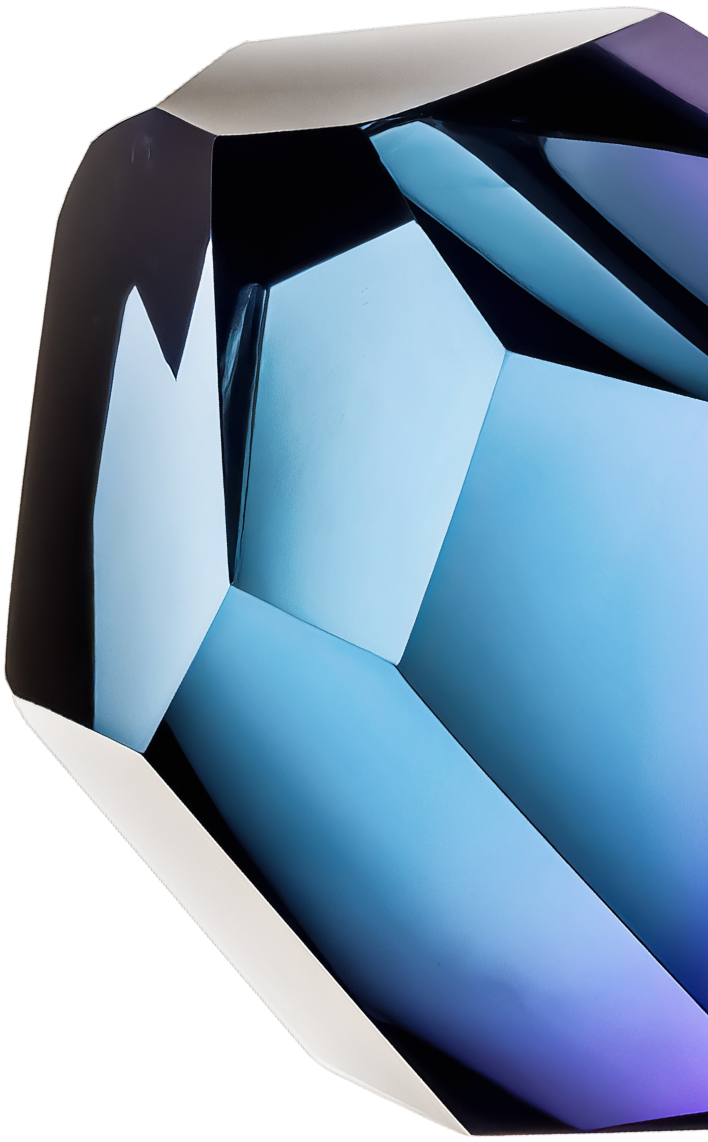
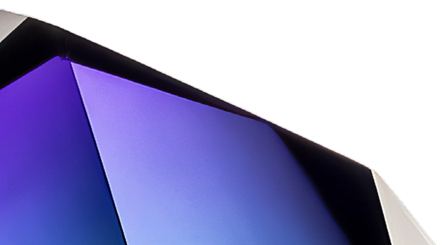


Table of contents



This combination of reasoning, logic, and access to external information that are all connected to a Generative AI model invokes the concept of an agent.

Introduction

Humans are fantastic at messy pattern recognition tasks. However, they often rely on tools - like books, Google Search, or a calculator - to supplement their prior knowledge before arriving at a conclusion. Just like humans, Generative AI models can be trained to use tools to access real-time information or suggest a real-world action. For example, a model can leverage a database retrieval tool to access specific information, like a customer's purchase history, so it can generate tailored shopping recommendations. Alternatively, based on a user's query, a model can make various API calls to send an email response to a colleague or complete a financial transaction on your behalf. To do so, the model must not only have access to a set of external tools, it needs the ability to plan and execute any task in a self-directed fashion. This combination of reasoning, logic, and access to external information that are all connected to a Generative AI model invokes the concept of an agent, or a program that extends beyond the standalone capabilities of a Generative AI model. This whitepaper dives into all these and associated aspects in more detail.

What is an agent?

In its most fundamental form, a Generative AI agent can be defined as an application that attempts to achieve a goal by observing the world and acting upon it using the tools that it has at its disposal. Agents are autonomous and can act independently of human intervention, especially when provided with proper goals or objectives they are meant to achieve. Agents can also be proactive in their approach to reaching their goals. Even in the absence of explicit instruction sets from a human, an agent can reason about what it should do next to achieve its ultimate goal. While the notion of agents in AI is quite general and powerful, this whitepaper focuses on the specific types of agents that Generative AI models are capable of building at the time of publication.

In order to understand the inner workings of an agent, let's first introduce the foundational components that drive the agent's behavior, actions, and decision making. The combination of these components can be described as a cognitive architecture, and there are many such architectures that can be achieved by the mixing and matching of these components. Focusing on the core functionalities, there are three essential components in an agent's cognitive architecture as shown in Figure 1.

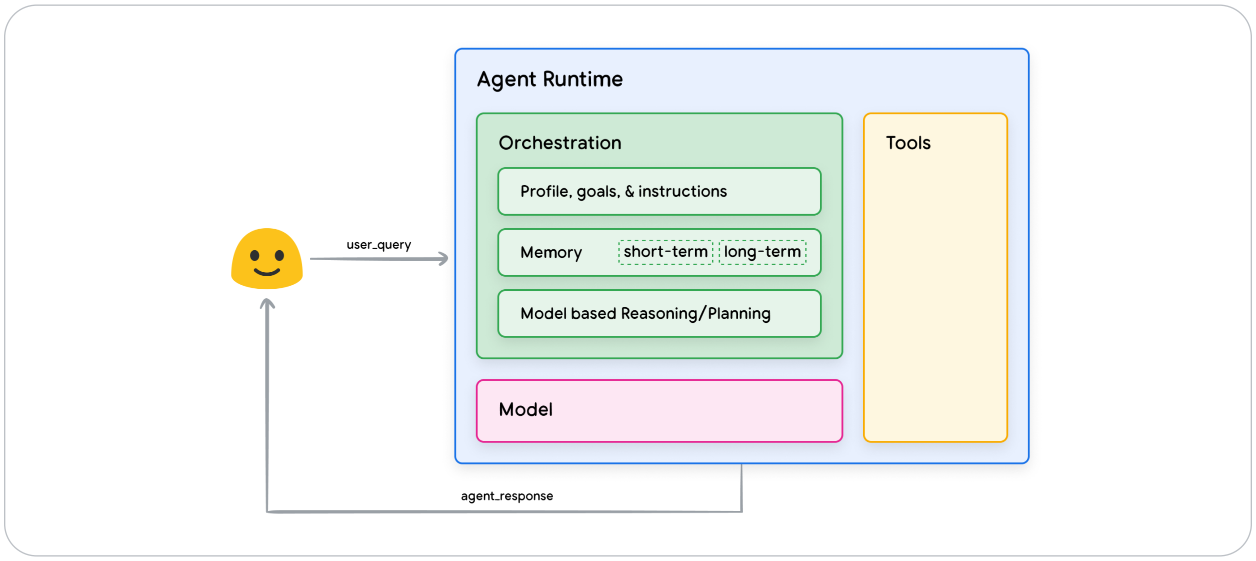


Figure 1. General agent architecture and components

The model

In the scope of an agent, a model refers to the language model (LM) that will be utilized as the centralized decision maker for agent processes. The model used by an agent can be one or multiple LM's of any size (small / large) that are capable of following instruction based reasoning and logic frameworks, like ReAct, Chain-of-Thought, or Tree-of-Thoughts. Models can be general purpose, multimodal or fine-tuned based on the needs of your specific agent architecture. For best production results, you should leverage a model that best fits your desired end application and, ideally, has been trained on data signatures associated with the tools that you plan to use in the cognitive architecture. It's important to note that the model is typically not trained with the specific configuration settings (i.e. tool choices, orchestration/ reasoning setup) of the agent. However, it's possible to further refine the model for the agent's tasks by providing it with examples that showcase the agent's capabilities, including instances of the agent using specific tools or reasoning steps in various contexts.

The tools

Foundational models, despite their impressive text and image generation, remain constrained by their inability to interact with the outside world. *Tools* bridge this gap, empowering agents to interact with external data and services while unlocking a wider range of actions beyond that of the underlying model alone. Tools can take a variety of forms and have varying depths of complexity, but typically align with common web API methods like GET, POST, PATCH, and DELETE. For example, a tool could update customer information in a database or fetch weather data to influence a travel recommendation that the agent is providing to the user. With tools, agents can access and process real-world information. This empowers them to support more specialized systems like retrieval augmented generation (RAG), which significantly extends an agent's capabilities beyond what the foundational model can achieve on its own. We'll discuss tools in more detail below, but the most important thing to understand is that tools bridge the gap between the agent's internal capabilities and the external world, unlocking a broader range of possibilities.

The orchestration layer

The orchestration layer describes a cyclical process that governs how the agent takes in information, performs some internal reasoning, and uses that reasoning to inform its next action or decision. In general, this loop will continue until an agent has reached its goal or a stopping point. The complexity of the orchestration layer can vary greatly depending on the agent and task it's performing. Some loops can be simple calculations with decision rules, while others may contain chained logic, involve additional machine learning algorithms, or implement other probabilistic reasoning techniques. We'll discuss more about the detailed implementation of the agent orchestration layers in the cognitive architecture section.

Agents vs. models

To gain a clearer understanding of the distinction between agents and models, consider the following chart:

Models	Agents
Knowledge is limited to what is available in their training data.	Knowledge is extended through the connection with external systems via tools
Single inference / prediction based on the user query. Unless explicitly implemented for the model, there is no management of session history or continuous context. (i.e. chat history)	Managed session history (i.e. chat history) to allow for multi turn inference / prediction based on user queries and decisions made in the orchestration layer. In this context, a 'turn' is defined as an interaction between the interacting system and the agent. (i.e. 1 incoming event/ query and 1 agent response)
No native tool implementation.	Tools are natively implemented in agent architecture.
No native logic layer implemented. Users can form prompts as simple questions or use reasoning frameworks (CoT, ReAct, etc.) to form complex prompts to guide the model in prediction.	Native cognitive architecture that uses reasoning frameworks like CoT, ReAct, or other pre-built agent frameworks like LangChain.

Cognitive architectures: How agents operate

Imagine a chef in a busy kitchen. Their goal is to create delicious dishes for restaurant patrons which involves some cycle of planning, execution, and adjustment.

- They gather information, like the patron's order and what ingredients are in the pantry and refrigerator.
- They perform some internal reasoning about what dishes and flavor profiles they can create based on the information they have just gathered.
- They take action to create the dish: chopping vegetables, blending spices, searing meat.

At each stage in the process the chef makes adjustments as needed, refining their plan as ingredients are depleted or customer feedback is received, and uses the set of previous outcomes to determine the next plan of action. This cycle of information intake, planning, executing, and adjusting describes a unique cognitive architecture that the chef employs to reach their goal.

Just like the chef, agents can use cognitive architectures to reach their end goals by iteratively processing information, making informed decisions, and refining next actions based on previous outputs. At the core of agent cognitive architectures lies the orchestration layer, responsible for maintaining memory, state, reasoning and planning. It uses the rapidly evolving field of prompt engineering and associated frameworks to guide reasoning and planning, enabling the agent to interact more effectively with its environment and complete tasks. Research in the area of prompt engineering frameworks and task planning for language models is rapidly evolving, yielding a variety of promising approaches. While not an exhaustive list, these are a few of the most popular frameworks and reasoning techniques available at the time of this publication:

- **ReAct**, a prompt engineering framework that provides a thought process strategy for language models to Reason and take action on a user query, with or without in-context examples. ReAct prompting has shown to outperform several SOTA baselines and improve human interoperability and trustworthiness of LLMs.

- **Chain-of-Thought (CoT)**, a prompt engineering framework that enables reasoning capabilities through intermediate steps. There are various sub-techniques of CoT including self-consistency, active-prompt, and multimodal CoT that each have strengths and weaknesses depending on the specific application.
- **Tree-of-thoughts (ToT)**, a prompt engineering framework that is well suited for exploration or strategic lookahead tasks. It generalizes over chain-of-thought prompting and allows the model to explore various thought chains that serve as intermediate steps for general problem solving with language models.

Agents can utilize one of the above reasoning techniques, or many other techniques, to choose the next best action for the given user request. For example, let's consider an agent that is programmed to use the *ReAct* framework to choose the correct actions and tools for the user query. The sequence of events might go something like this:

1. User sends query to the agent
2. Agent begins the ReAct sequence
3. The agent provides a prompt to the model, asking it to generate one of the next ReAct steps and its corresponding output:
   a. **Question:** The input question from the user query, provided with the prompt
   b. **Thought:** The model's thoughts about what it should do next
   c. **Action:** The model's decision on what action to take next
      i. This is where tool choice can occur
      ii. For example, an action could be one of [Flights, Search, Code, None], where the first 3 represent a known tool that the model can choose, and the last represents "no tool choice"

   d. **Action input:** The model's decision on what inputs to provide to the tool (if any)
   e. **Observation:** The result of the action / action input sequence
      i. This thought / action / action input / observation could repeat *N-times* as needed
   f. **Final answer:** The model's final answer to provide to the original user query

4. The ReAct loop concludes and a final answer is provided back to the user

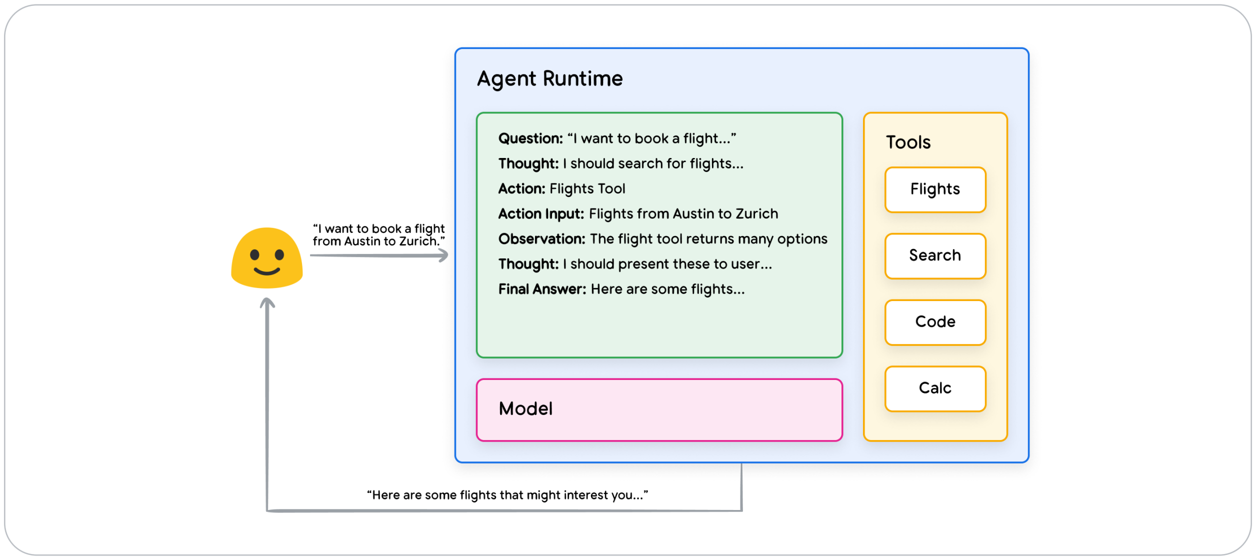

Agent Runtime
"I want to book a flight from Austin to Zurich."
Question: "I want to book a flight..."
Thought: I should search for flights...
Action: Flights Tool
Action Input: Flights from Austin to Zurich
Observation: The flight tool returns many options
Thought: I should present these to user...
Final Answer: Here are some flights...
Model
Tools
Flights
Search
Code
Calc
"Here are some flights that might interest you..."

Figure 2. Example agent with ReAct reasoning in the orchestration layer

As shown in Figure 2, the model, tools, and agent configuration work together to provide a grounded, concise response back to the user based on the user's original query. While the model could have guessed at an answer (hallucinated) based on its prior knowledge, it instead used a tool (Flights) to search for real-time external information. This additional information was provided to the model, allowing it to make a more informed decision based on real factual data and to summarize this information back to the user.

In summary, the quality of agent responses can be tied directly to the model's ability to reason and act about these various tasks, including the ability to select the right tools, and how well that tools has been defined. Like a chef crafting a dish with fresh ingredients and attentive to customer feedback, agents rely on sound reasoning and reliable information to deliver optimal results. In the next section, we'll dive into the various ways agents connect with fresh data.

Tools: Our keys to the outside world

While language models excel at processing information, they lack the ability to directly perceive and influence the real world. This limits their usefulness in situations requiring interaction with external systems or data. This means that, in a sense, a language model is only as good as what it has learned from its training data. But regardless of how much data we throw at a model, they still lack the fundamental ability to interact with the outside world. So how can we empower our models to have real-time, context-aware interaction with external systems? Functions, Extensions, Data Stores and Plugins are all ways to provide this critical capability to the model.

While they go by many names, tools are what create a link between our foundational models and the outside world. This link to external systems and data allows our agent to perform a wider variety of tasks and do so with more accuracy and reliability. For instance, tools can enable agents to adjust smart home settings, update calendars, fetch user information from a database, or send emails based on a specific set of instructions.

As of the date of this publication, there are three primary tool types that Google models are able to interact with: Extensions, Functions, and Data Stores. By equipping agents with tools, we unlock a vast potential for them to not only understand the world but also act upon it, opening doors to a myriad of new applications and possibilities.

Extensions

The easiest way to understand Extensions is to think of them as bridging the gap between an API and an agent in a standardized way, allowing agents to seamlessly execute APIs regardless of their underlying implementation. Let's say that you've built an agent with a goal of helping users book flights. You know that you want to use the Google Flights API to retrieve flight information, but you're not sure how you're going to get your agent to make calls to this API endpoint.

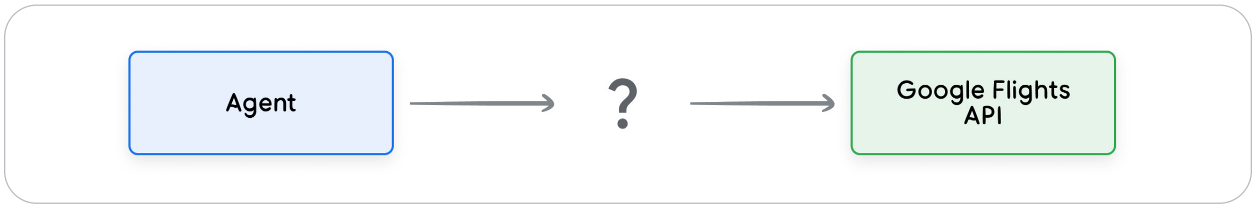


Figure 3. How do Agents interact with External APIs?

One approach could be to implement custom code that would take the incoming user query, parse the query for relevant information, then make the API call. For example, in a flight booking use case a user might state "I want to book a flight *from Austin to Zurich*." In this scenario, our custom code solution would need to extract "Austin" and "Zurich" as relevant entities from the user query before attempting to make the API call. But what happens if the user says "I want to book a flight *to Zurich*" and never provides a departure city? The API call would fail without the required data and more code would need to be implemented in order to catch edge and corner cases like this. This approach is not scalable and could easily break in any scenario that falls outside of the implemented custom code.

A more resilient approach would be to use an Extension. An Extension bridges the gap between an agent and an API by:

1. Teaching the agent how to use the API endpoint using examples.
2. Teaching the agent what arguments or parameters are needed to successfully call the API endpoint.

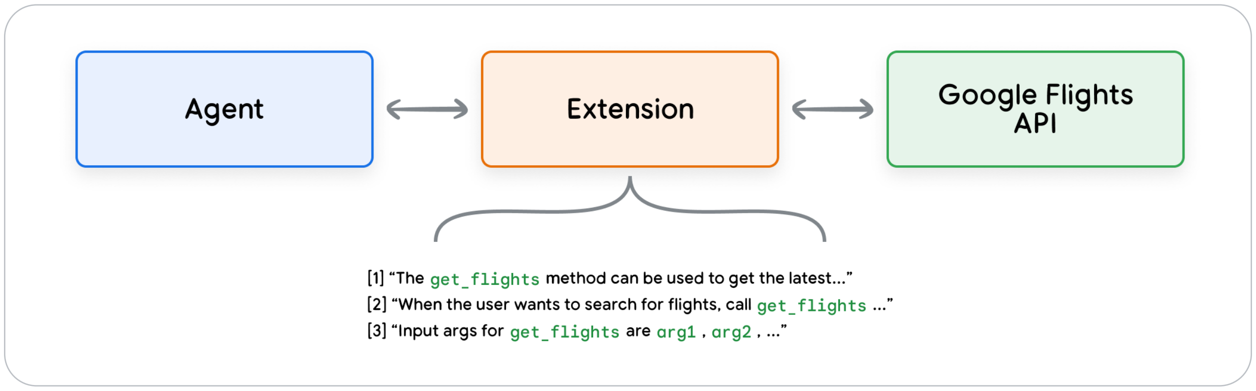

Agent
Extension
Google Flights API
[1] "The get_flights method can be used to get the latest..."
[2] "When the user wants to search for flights, call get_flights ..."
[3] "Input args for get_flights are arg1 , arg2 , ..."

Figure 4. Extensions connect Agents to External APIs

Extensions can be crafted independently of the agent, but should be provided as part of the agent's configuration. The agent uses the model and examples at run time to decide which Extension, if any, would be suitable for solving the user's query. This highlights a key strength of Extensions, their *built-in example types*, that allow the agent to dynamically select the most appropriate Extension for the task.

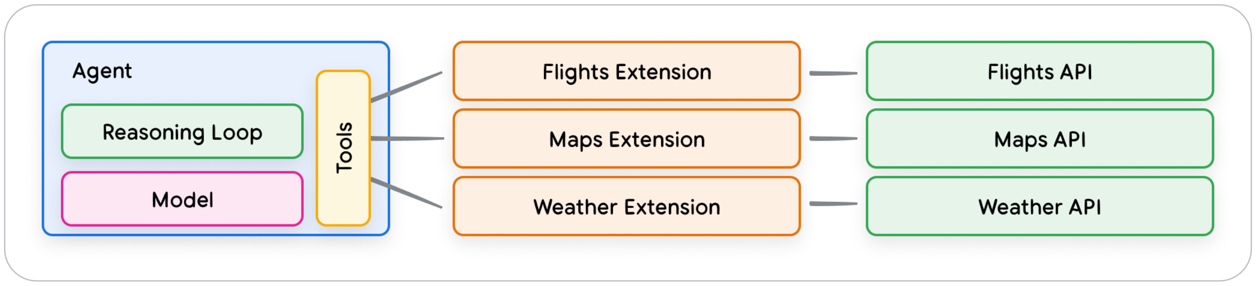


Figure 5. 1-to-many relationship between Agents, Extensions and APIs

Think of this the same way that a software developer decides which API endpoints to use while solving and solutioning for a user's problem. If the user wants to book a flight, the developer might use the Google Flights API. If the user wants to know where the nearest coffee shop is relative to their location, the developer might use the Google Maps API. In this same way, the agent / model stack uses a set of known Extensions to decide which one will be the best fit for the user's query. If you'd like to see Extensions in action, you can try them out on the Gemini application by going to Settings > Extensions and then enabling any you would like to test. For example, you could enable the Google Flights extension then ask Gemini "Show me flights from Austin to Zurich leaving next Friday."

Sample Extensions

To simplify the usage of Extensions, Google provides some out of the box extensions that can be quickly imported into your project and used with minimal configurations. For example, the Code Interpreter extension in Snippet 1 allows you to generate and run Python code from a natural language description.

Python

```
import vertexai
import pprint


PROJECT_ID = "YOUR_PROJECT_ID"
REGION = "us-central1"

vertexai.init(project=PROJECT_ID, location=REGION)


from vertexai.preview.extensions import Extension


extension_code_interpreter = Extension.from_hub("code_interpreter")
CODE_QUERY = """Write a python method to invert a binary tree in O(n) time."""


response = extension_code_interpreter.execute(
  operation_id = "generate_and_execute",
  operation_params = {"query": CODE_QUERY}
  )


print("Generated Code:")
pprint.pprint({response['generated_code']})


# The above snippet will generate the following code.
```

Generated Code:
class TreeNode:
  def __init__(self, val=0, left=None, right=None):
    self.val = val
    self.left = left
    self.right = right
```

Continues next page...
```

Python

```
def invert_binary_tree(root):
    """
    Inverts a binary tree.
    Args:
        root: The root of the binary tree.
    Returns:
        The root of the inverted binary tree.
    """


    if not root:
        return None


    # Swap the left and right children recursively
    root.left, root.right =
invert_binary_tree(root.right), invert_binary_tree(root.left)


    return root


# Example usage:
# Construct a sample binary tree
root = TreeNode(4)
root.left = TreeNode(2)
root.right = TreeNode(7)
root.left.left = TreeNode(1)
root.left.right = TreeNode(3)
root.right.left = TreeNode(6)
root.right.right = TreeNode(9)


# Invert the binary tree
inverted_root = invert_binary_tree(root)
```
```` ``` ````

Snippet 1. Code Interpreter Extension can generate and run Python code
````

To summarize, Extensions provide a way for agents to perceive, interact, and influence the outside world in a myriad of ways. The selection and invocation of these Extensions is guided by the use of Examples, all of which are defined as part of the Extension configuration.

Functions

In the world of software engineering, functions are defined as self-contained modules of code that accomplish a specific task and can be reused as needed. When a software developer is writing a program, they will often create many functions to do various tasks. They will also define the logic for when to call function_a versus function_b, as well as the expected inputs and outputs.

Functions work very similarly in the world of agents, but we can replace the software developer with a model. A model can take a set of known functions and decide when to use each Function and what arguments the Function needs based on its specification. Functions differ from Extensions in a few ways, most notably:

1. A model outputs a Function and its arguments, but doesn't make a live API call.
2. Functions are executed on the *client-side*, while Extensions are executed on the *agent-side*.

Using our Google Flights example again, a simple setup for functions might look like the example in Figure 7.

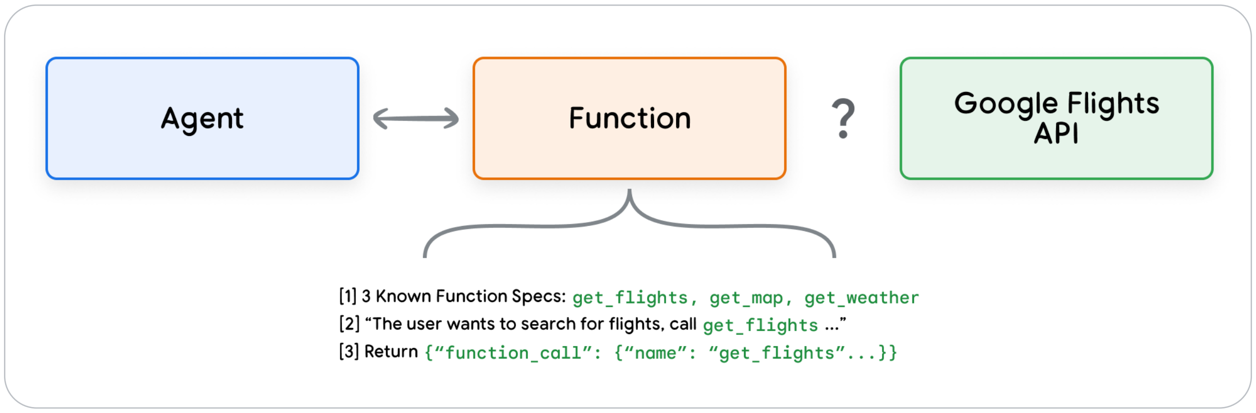

Agent
Function
?
Google Flights API
[1] 3 Known Function Specs: get_flights, get_map, get_weather
[2] "The user wants to search for flights, call get_flights ..."
[3] Return {"function_call": {"name": "get_flights"...}}

Figure 7. How do functions interact with external APIs?

Note that the main difference here is that neither the Function nor the agent interact directly with the Google Flights API. So how does the API call actually happen?

With functions, the logic and execution of calling the actual API endpoint is offloaded away from the agent and back to the client-side application as seen in Figure 8 and Figure 9 below. This offers the developer more granular control over the flow of data in the application. There are many reasons why a Developer might choose to use functions over Extensions, but a few common use cases are:

- API calls need to be made at another layer of the application stack, outside of the direct agent architecture flow (e.g. a middleware system, a front end framework, etc.)
- Security or Authentication restrictions that prevent the agent from calling an API directly (e.g API is not exposed to the internet, or non-accessible by agent infrastructure)
- Timing or order-of-operations constraints that prevent the agent from making API calls in real-time. (i.e. batch operations, human-in-the-loop review, etc.)

- Additional data transformation logic needs to be applied to the API Response that the agent cannot perform. For example, consider an API endpoint that doesn't provide a filtering mechanism for limiting the number of results returned. Using Functions on the client-side provides the developer additional opportunities to make these transformations.
- The developer wants to iterate on agent development without deploying additional infrastructure for the API endpoints (i.e. Function Calling can act like "stubbing" of APIs)

While the difference in internal architecture between the two approaches is subtle as seen in Figure 8, the additional control and decoupled dependency on external infrastructure makes Function Calling an appealing option for the Developer.

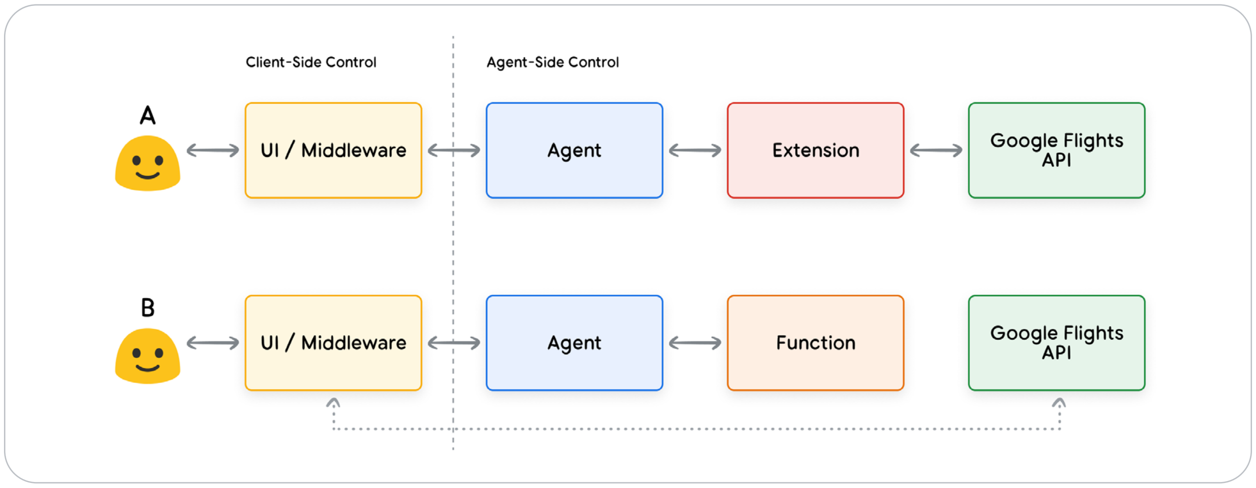


Figure 8. Delineating client vs. agent side control for extensions and function calling

Use cases

A model can be used to invoke functions in order to handle complex, client-side execution flows for the end user, where the agent Developer might not want the language model to manage the API execution (as is the case with Extensions). Let's consider the following example where an agent is being trained as a travel concierge to interact with users that want to book vacation trips. The goal is to get the agent to produce a list of cities that we can use in our middleware application to download images, data, etc. for the user's trip planning. A user might say something like:

I'd like to take a ski trip with my family but I'm not sure where to go.

In a typical prompt to the model, the output might look like the following:

Sure, here's a list of cities that you can consider for family ski trips:

- Crested Butte, Colorado, USA
- Whistler, BC, Canada
- Zermatt, Switzerland

While the above output contains the data that we need (city names), the format isn't ideal for parsing. With Function Calling, we can teach a model to format this output in a structured style (like JSON) that's more convenient for another system to parse. Given the same input prompt from the user, an example JSON output from a Function might look like Snippet 5 instead.

Unset

```
function_call {
  name: "display_cities"
  args: {
    "cities": ["Crested Butte", "Whistler", "Zermatt"],
    "preferences": "skiing"
    }
}
```

Snippet 5. Sample Function Call payload for displaying a list of cities and user preferences

This JSON payload is generated by the model, and then sent to our Client-side server to do whatever we would like to do with it. In this specific case, we'll call the Google Places API to take the cities provided by the model and look up Images, then provide them as formatted rich content back to our User. Consider this sequence diagram in Figure 9 showing the above interaction in step by step detail.

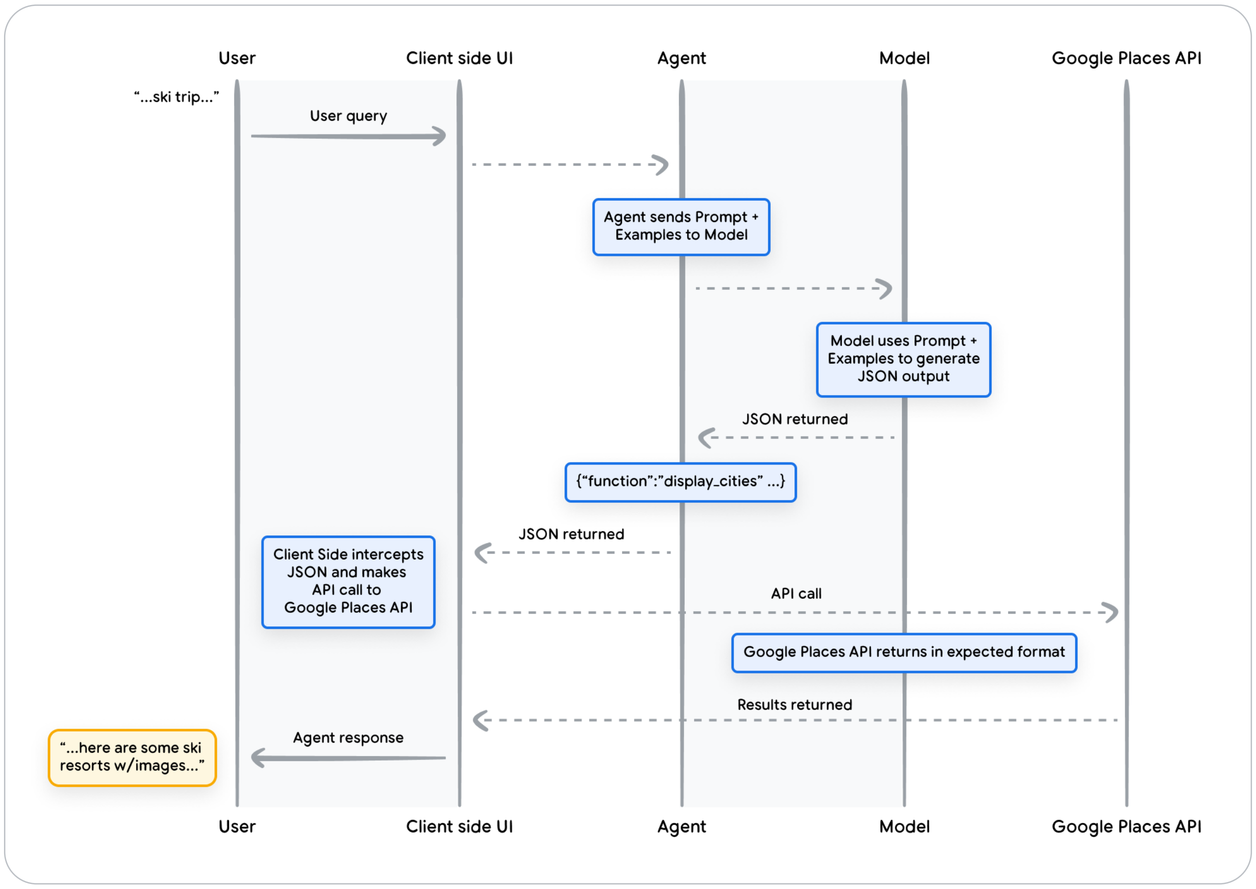

User
Client side UI
Agent
Model
Google Places API
"...ski trip..."
User query
Agent sends Prompt + Examples to Model
Model uses Prompt + Examples to generate JSON output
JSON returned
{"function":"display_cities" ...}
JSON returned
Client Side intercepts JSON and makes API call to Google Places API
API call
Google Places API returns in expected format
Results returned
Agent response
"...here are some ski resorts w/images..."
User
Client side UI
Agent
Model
Google Places API

Figure 9. Sequence diagram showing the lifecycle of a Function Call

The result of the example in Figure 9 is that the model is leveraged to "fill in the blanks" with the parameters required for the Client side UI to make the call to the Google Places API. The Client side UI manages the actual API call using the parameters provided by the model in the returned Function. This is just one use case for Function Calling, but there are many other scenarios to consider like:

- You want a language model to suggest a function that you can use in your code, but you don't want to include credentials in your code. Because function calling doesn't run the function, you don't need to include credentials in your code with the function information.

- You are running asynchronous operations that can take more than a few seconds. These scenarios work well with function calling because it's an asynchronous operation.
- You want to run functions on a device that's different from the system producing the function calls and their arguments.

One key thing to remember about functions is that they are meant to offer the developer much more control over not only the execution of API calls, but also the entire flow of data in the application as a whole. In the example in Figure 9, the developer chose to not return API information back to the agent as it was not pertinent for future actions the agent might take. However, based on the architecture of the application, it may make sense to return the external API call data to the agent in order to influence future reasoning, logic, and action choices. Ultimately, it is up to the application developer to choose what is right for the specific application.

Function sample code

To achieve the above output from our ski vacation scenario, let's build out each of the components to make this work with our gemini-1.5-flash-001 model.

First, we'll define our display_cities function as a simple Python method.

Python

```python
def display_cities(cities: list[str], preferences: Optional[str] = None):
    """Provides a list of cities based on the user's search query and preferences.

    Args:
        preferences (str): The user's preferences for the search, like skiing,
        beach, restaurants, bbq, etc.
        cities (list[str]): The list of cities being recommended to the user.

    Returns:
        list[str]: The list of cities being recommended to the user.
    """

    return cities
```

Snippet 6. Sample python method for a function that will display a list of cities.

Next, we'll instantiate our model, build the Tool, then pass in our user's query and tools to the model. Executing the code below would result in the output as seen at the bottom of the code snippet.

Python

```python
from vertexai.generative_models import GenerativeModel, Tool, FunctionDeclaration

model = GenerativeModel("gemini-1.5-flash-001")

display_cities_function = FunctionDeclaration.from_func(display_cities)
tool = Tool(function_declarations=[display_cities_function])

message = "I'd like to take a ski trip with my family but I'm not sure where
to go."

res = model.generate_content(message, tools=[tool])

print(f"Function Name: {res.candidates[0].content.parts[0].function_call.name}")
print(f"Function Args: {res.candidates[0].content.parts[0].function_call.args}")

> Function Name: display_cities
> Function Args: {'preferences': 'skiing', 'cities': ['Aspen', 'Vail',
'Park City']}
```

Snippet 7. Building a Tool, sending to the model with a user query and allowing the function call to take place

In summary, functions offer a straightforward framework that empowers application developers with fine-grained control over data flow and system execution, while effectively leveraging the agent/model for critical input generation. Developers can selectively choose whether to keep the agent "in the loop" by returning external data, or omit it based on specific application architecture requirements.

Data stores

Imagine a language model as a vast library of books, containing its training data. But unlike a library that continuously acquires new volumes, this one remains static, holding only the knowledge it was initially trained on. This presents a challenge, as real-world knowledge is constantly evolving. Data Stores address this limitation by providing access to more dynamic and up-to-date information, and ensuring a model's responses remain grounded in factuality and relevance.

Consider a common scenario where a developer might need to provide a small amount of additional data to a model, perhaps in the form of spreadsheets or PDFs.

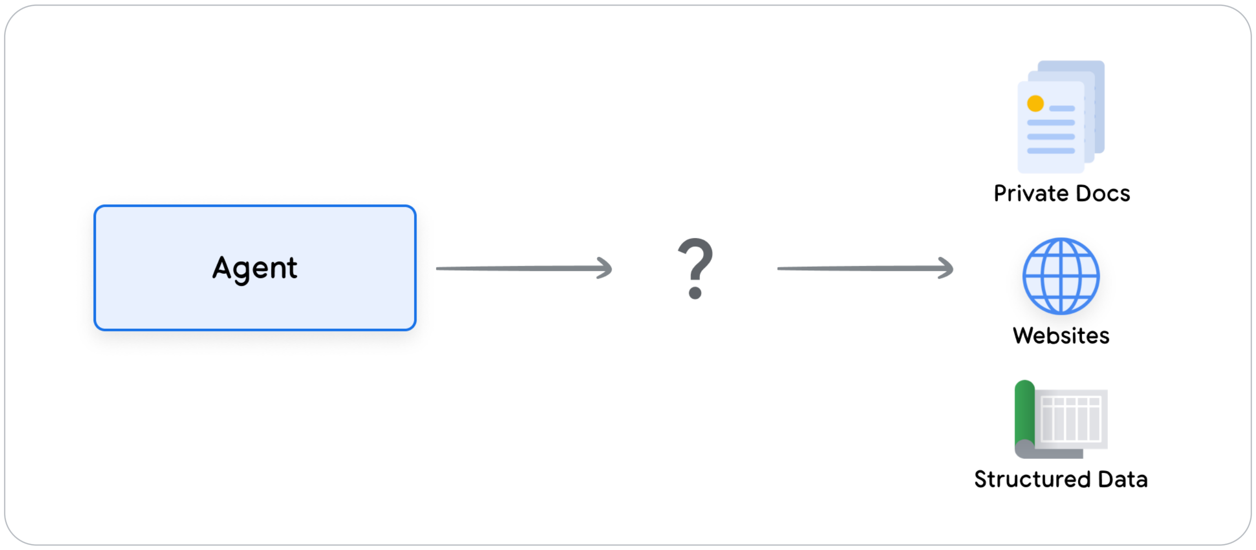


Figure 10. How can Agents interact with structured and unstructured data?

Data Stores allow developers to provide additional data in its original format to an agent, eliminating the need for time-consuming data transformations, model retraining, or fine-tuning. The Data Store converts the incoming document into a set of *vector database embeddings* that the agent can use to extract the information it needs to supplement its next action or response to the user.

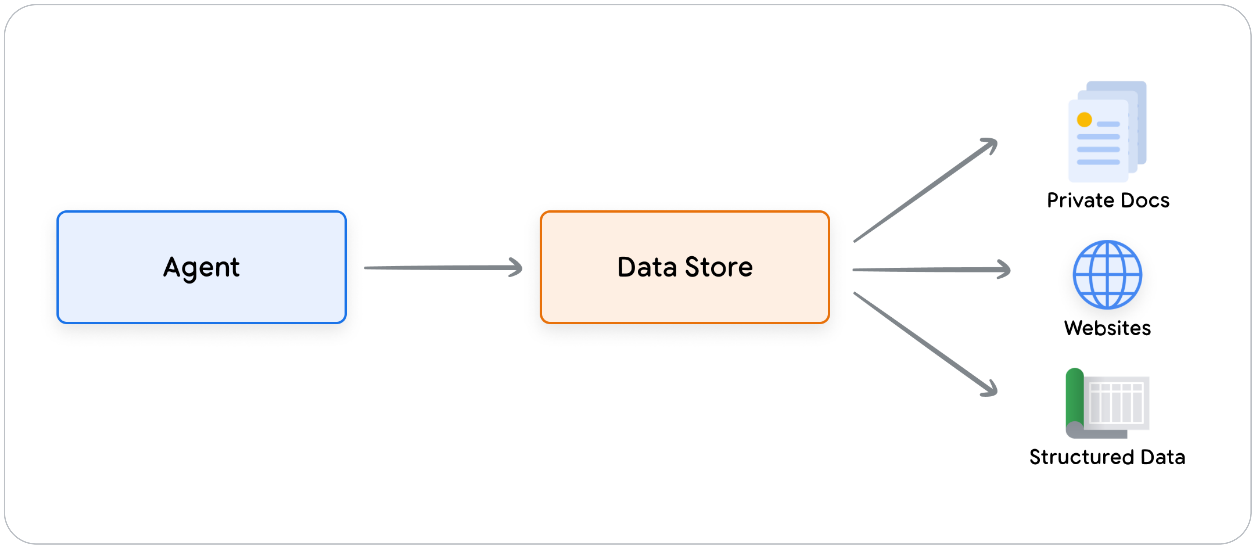


Figure 11. Data Stores connect Agents to new real-time data sources of various types.

Implementation and application

In the context of Generative AI agents, Data Stores are typically implemented as a *vector database* that the developer wants the agent to have access to at runtime. While we won't cover vector databases in depth here, the key point to understand is that they store data in the form of vector embeddings, a type of high-dimensional vector or mathematical representation of the data provided. One of the most prolific examples of Data Store usage with language models in recent times has been the implementation of Retrieval Augmented

Generation (RAG) based applications. These applications seek to extend the breadth and depth of a model's knowledge beyond the foundational training data by giving the model access to data in various formats like:

- Website content
- Structured Data in formats like PDF, Word Docs, CSV, Spreadsheets, etc.
- Unstructured Data in formats like HTML, PDF, TXT, etc.

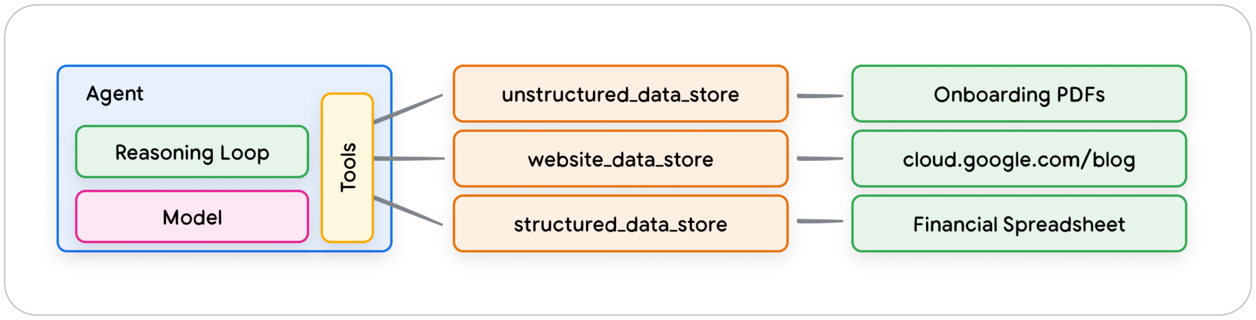


Figure 12. 1-to-many relationship between agents and data stores, which can represent various types of pre-indexed data

The underlying process for each user request and agent response loop is generally modeled as seen in Figure 13.

1. A user query is sent to an embedding model to generate embeddings for the query
2. The query embeddings are then matched against the contents of the vector database using a matching algorithm like SCaNN
3. The matched content is retrieved from the vector database in text format and sent back to the agent
4. The agent receives both the user query and retrieved content, then formulates a response or action

5. A final response is sent to the user

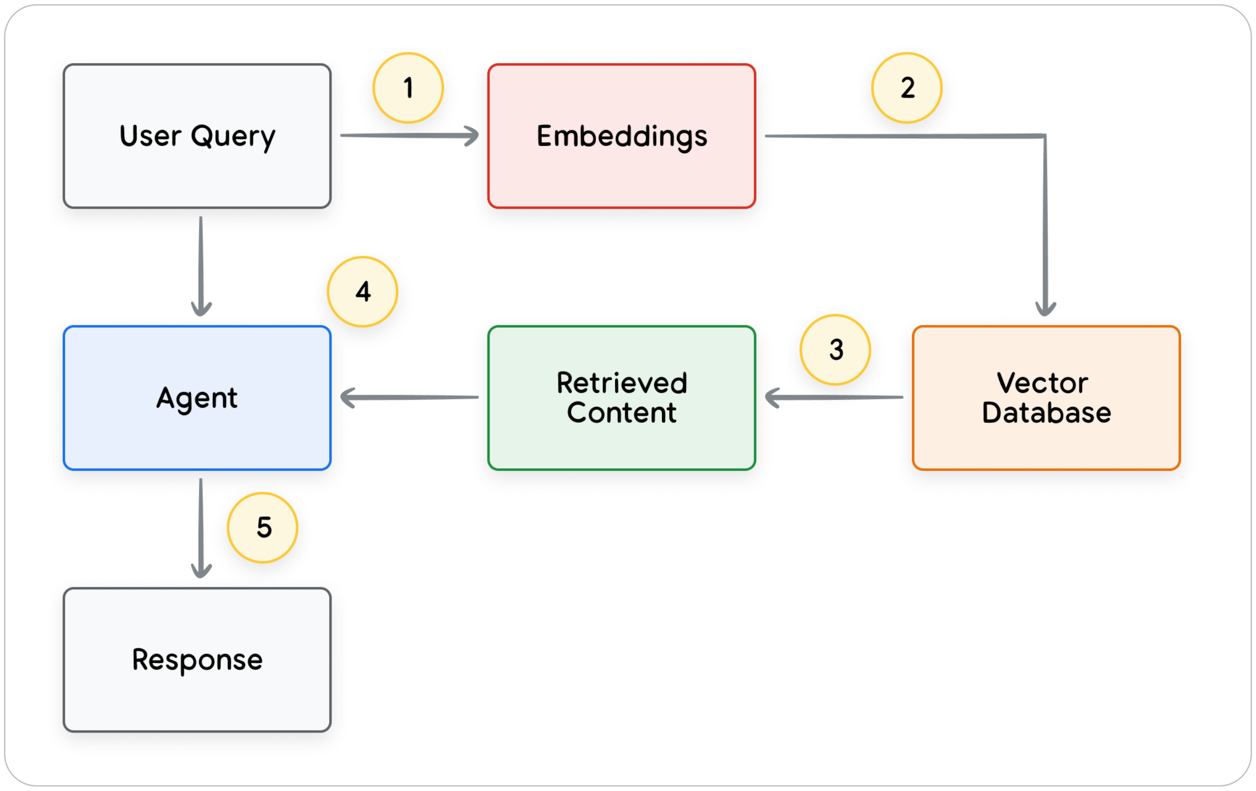


Figure 13. The lifecycle of a user request and agent response in a RAG based application

The end result is an application that allows the agent to match a user's query to a known data store through vector search, retrieve the original content, and provide it to the orchestration layer and model for further processing. The next action might be to provide a final answer to the user, or perform an additional vector search to further refine the results.

A sample interaction with an agent that implements *RAG with ReAct reasoning/planning* can be seen in Figure 14.

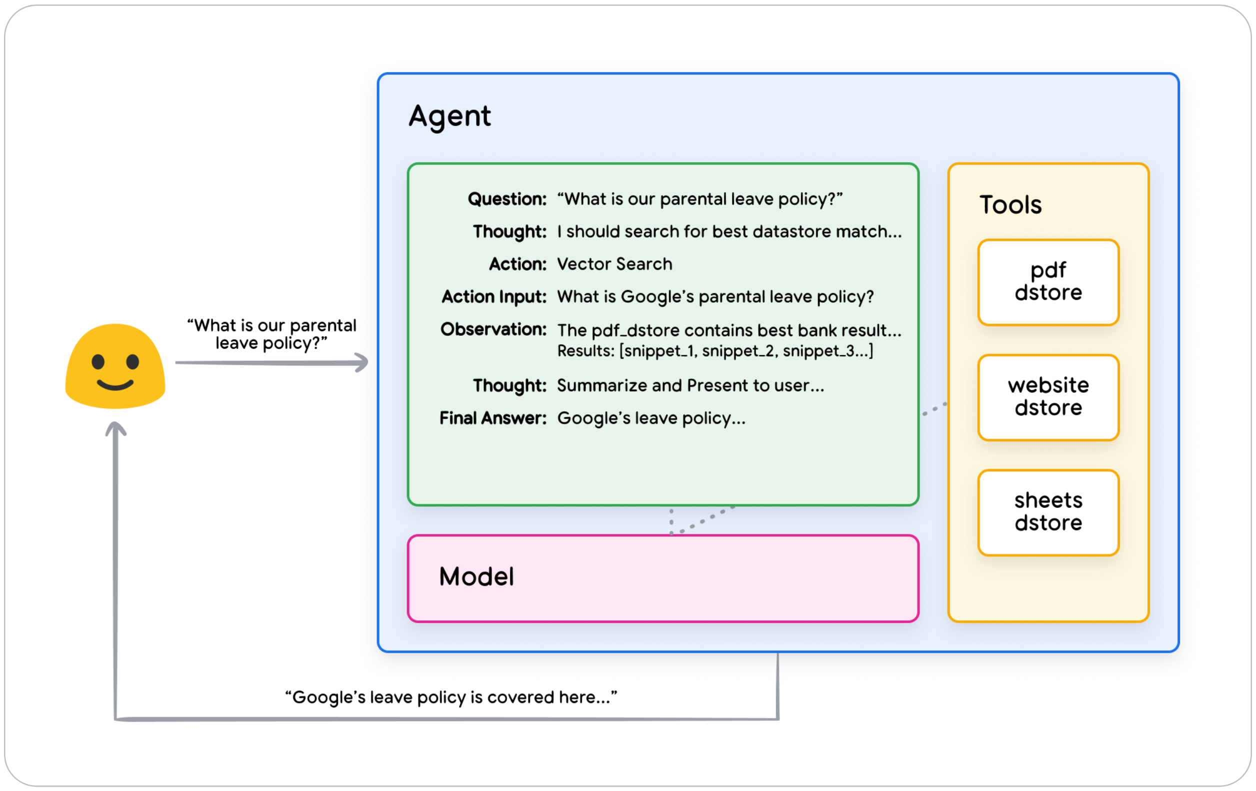

Agent
Question: "What is our parental leave policy?"
Thought: I should search for best datastore match...
Action: Vector Search
Action Input: What is Google's parental leave policy?
Observation: The pdf_dstore contains best bank result... Results: [snippet_1, snippet_2, snippet_3...]
Thought: Summarize and Present to user...
Final Answer: Google's leave policy...
Model
Tools
pdf dstore
website dstore
sheets dstore
"What is our parental leave policy?"
"Google's leave policy is covered here..."

Figure 14. Sample RAG based application w/ ReAct reasoning/planning

Tools recap

To summarize, extensions, functions and data stores make up a few different tool types available for agents to use at runtime. Each has their own purpose and they can be used together or independently at the discretion of the agent developer.

	Extensions	Function Calling	Data Stores
Execution	Agent-Side Execution	Client-Side Execution	Agent-Side Execution
Use Case	• Developer wants agent to control interactions with the API endpoints • Useful when leveraging native pre-built Extensions (i.e. Vertex Search, Code Interpreter, etc.) • Multi-hop planning and API calling (i.e. the next agent action depends on the outputs of the previous action / API call)	• Security or Authentication restrictions prevent the agent from calling an API directly • Timing constraints or order-of-operations constraints that prevent the agent from making API calls in real-time. (i.e. batch operations, human-in-the-loop review, etc.) • API that is not exposed to the internet, or non-accessible by Google systems	Developer wants to implement Retrieval Augmented Generation (RAG) with any of the following data types: • Website Content from pre-indexed domains and URLs • Structured Data in formats like PDF, Word Docs, CSV, Spreadsheets, etc. • Relational / Non-Relational Databases • Unstructured Data in formats like HTML, PDF, TXT, etc.

Enhancing model performance with targeted learning

A crucial aspect of using models effectively is their ability to choose the right tools when generating output, especially when using tools at scale in production. While general training helps models develop this skill, real-world scenarios often require knowledge beyond the training data. Imagine this as the difference between basic cooking skills and mastering a specific cuisine. Both require foundational cooking knowledge, but the latter demands targeted learning for more nuanced results.

To help the model gain access to this type of specific knowledge, several approaches exist:

- **In-context learning:** This method provides a generalized model with a prompt, tools, and few-shot examples at inference time which allows it to learn ‘on the fly' how and when to use those tools for a specific task. The ReAct framework is an example of this approach in natural language.
- **Retrieval-based in-context learning:** This technique dynamically populates the model prompt with the most relevant information, tools, and associated examples by retrieving them from external memory. An example of this would be the ‘Example Store’ in Vertex AI extensions or the data stores RAG based architecture mentioned previously.
- **Fine-tuning based learning:** This method involves training a model using a larger dataset of specific examples prior to inference. This helps the model understand when and how to apply certain tools prior to receiving any user queries.

To provide additional insights on each of the targeted learning approaches, let’s revisit our cooking analogy.

- Imagine a chef has received a specific recipe (the prompt), a few key ingredients (relevant tools) and some example dishes (few-shot examples) from a customer. Based on this limited information and the chef's general knowledge of cooking, they will need to figure out how to prepare the dish 'on the fly' that most closely aligns with the recipe and the customer's preferences. This is *in-context learning*.
- Now let's imagine our chef in a kitchen that has a well-stocked pantry (external data stores) filled with various ingredients and cookbooks (examples and tools). The chef is now able to dynamically choose ingredients and cookbooks from the pantry and better align to the customer's recipe and preferences. This allows the chef to create a more informed and refined dish leveraging both *existing and new knowledge*. This is *retrieval-based in-context learning*.
- Finally, let's imagine that we sent our chef back to school to learn a new cuisine or set of cuisines (pre-training on a larger dataset of specific examples). This allows the chef to approach future unseen customer recipes with deeper understanding. This approach is perfect if we want the chef to excel in specific cuisines (knowledge domains). This is *fine-tuning based learning*.

Each of these approaches offers unique advantages and disadvantages in terms of speed, cost, and latency. However, by combining these techniques in an agent framework, we can leverage the various strengths and minimize their weaknesses, allowing for a more robust and adaptable solution.

Agent quick start with LangChain

In order to provide a real-world executable example of an agent in action, we'll build a quick prototype with the LangChain and LangGraph libraries. These popular open source libraries allow users to build customer agents by "chaining" together sequences of logic, reasoning, and tool calls to answer a user's query. We'll use our `gemini-1.5-flash-001` model and some simple tools to answer a multi-stage query from the user as seen in Snippet 8.

The tools we are using are the SerpAPI (for Google Search) and the Google Places API. After executing our program in Snippet 8, you can see the sample output in Snippet 9.

Python

```python
from langgraph.prebuilt import create_react_agent
from langchain_core.tools import tool
from langchain_community.utilities import SerpAPIWrapper
from langchain_community.tools import GooglePlacesTool

os.environ["SERPAPI_API_KEY"] = "XXXXX"
os.environ["GPLACES_API_KEY"] = "XXXXX"

@tool
def search(query: str):
   """Use the SerpAPI to run a Google Search."""
   search = SerpAPIWrapper()
   return search.run(query)

@tool
def places(query: str):
   """Use the Google Places API to run a Google Places Query."""
   places = GooglePlacesTool()
   return places.run(query)

model = ChatVertexAI(model="gemini-1.5-flash-001")
tools = [search, places]

query = "Who did the Texas Longhorns play in football last week? What is the
address of the other team's stadium?"

agent = create_react_agent(model, tools)
input = {"messages": [("human", query)]}

for s in agent.stream(input, stream_mode="values"):
   message = s["messages"][-1]
   if isinstance(message, tuple):
      print(message)
   else:
      message.pretty_print()
```

Snippet 8. Sample LangChain and LangGraph based agent with tools

Unset

```
=============================== Human Message ================================
Who did the Texas Longhorns play in football last week? What is the address
of the other team's stadium?
================================= Ai Message =================================
Tool Calls: search
Args:
   query: Texas Longhorns football schedule
================================ Tool Message ================================
Name: search
{...Results: "NCAA Division I Football, Georgia, Date..."}
================================= Ai Message =================================
The Texas Longhorns played the Georgia Bulldogs last week.
Tool Calls: places
Args:
   query: Georgia Bulldogs stadium
================================ Tool Message ================================
Name: places

{...Sanford Stadium Address: 100 Sanford...}
================================= Ai Message =================================
The address of the Georgia Bulldogs stadium is 100 Sanford Dr, Athens, GA
30602, USA.
```

Snippet 9. Output from our program in Snippet 8

While this is a fairly simple agent example, it demonstrates the foundational components of Model, Orchestration, and tools all working together to achieve a specific goal. In the final section, we'll explore how these components come together in Google-scale managed products like Vertex AI agents and Generative Playbooks.

Production applications with Vertex AI agents

While this whitepaper explored the core components of agents, building production-grade applications requires integrating them with additional tools like user interfaces, evaluation frameworks, and continuous improvement mechanisms. Google's *Vertex AI* platform simplifies this process by offering a fully managed environment with all the fundamental elements covered earlier. Using a *natural language interface*, developers can rapidly define crucial elements of their agents - goals, task instructions, tools, sub-agents for task delegation, and examples - to easily construct the desired system behavior. In addition, the platform comes with a set of development tools that allow for testing, evaluation, measuring agent performance, debugging, and improving the overall quality of developed agents. This allows developers to focus on building and refining their agents while the complexities of infrastructure, deployment and maintenance are managed by the platform itself.

In Figure 15 we've provided a sample architecture of an agent that was built on the Vertex AI platform using various features such as Vertex Agent Builder, Vertex Extensions, Vertex Function Calling and Vertex Example Store to name a few. The architecture includes many of the various components necessary for a production ready application.

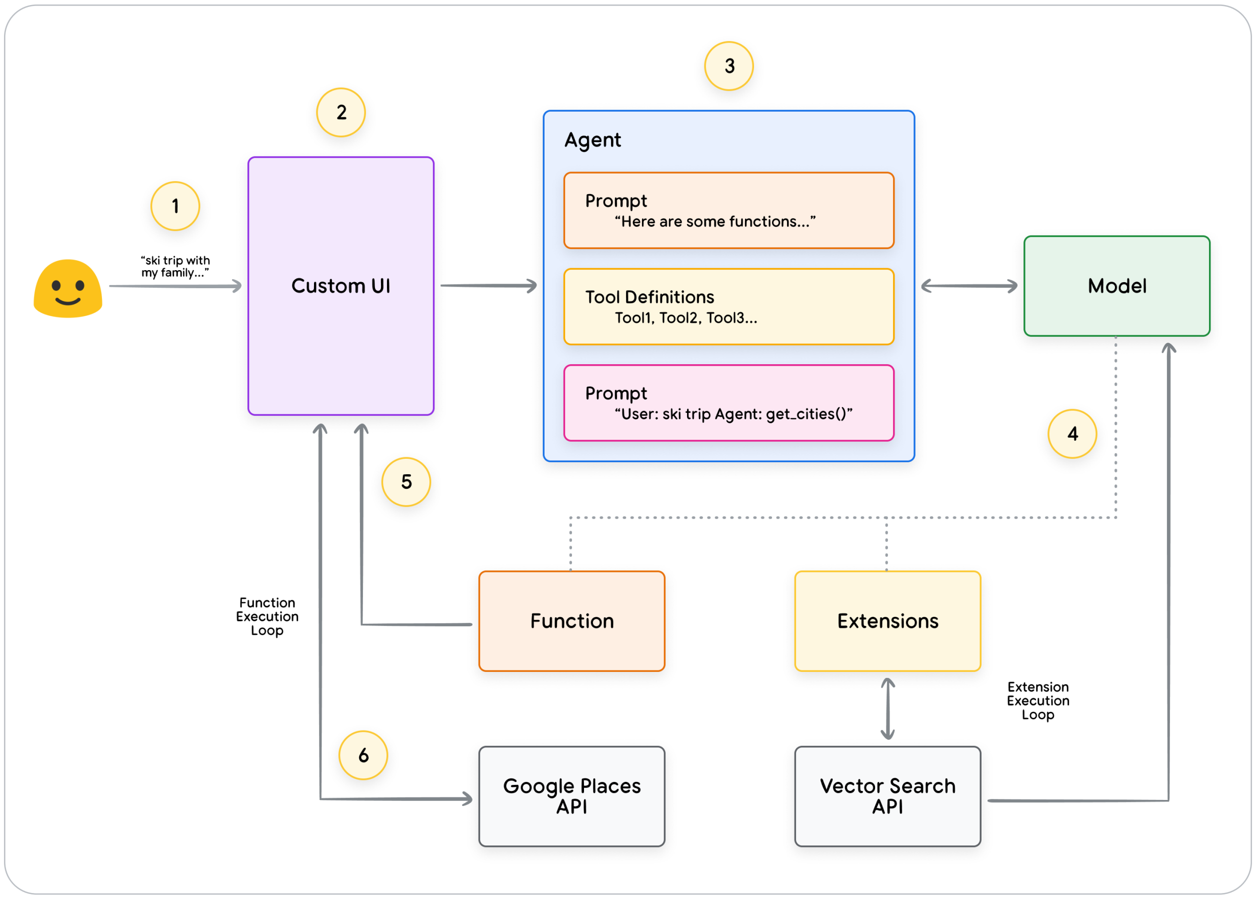


Figure 15. Sample end-to-end agent architecture built on Vertex AI platform

You can try a sample of this prebuilt agent architecture from our official documentation.

Summary

In this whitepaper we've discussed the foundational building blocks of Generative AI agents, their compositions, and effective ways to implement them in the form of cognitive architectures. Some key takeaways from this whitepaper include:

1. Agents extend the capabilities of language models by leveraging tools to access real-time information, suggest real-world actions, and plan and execute complex tasks autonomously. agents can leverage one or more language models to decide when and how to transition through states and use external tools to complete any number of complex tasks that would be difficult or impossible for the model to complete on its own.

2. At the heart of an agent's operation is the orchestration layer, a cognitive architecture that structures reasoning, planning, decision-making and guides its actions. Various reasoning techniques such as ReAct, Chain-of-Thought, and Tree-of-Thoughts, provide a framework for the orchestration layer to take in information, perform internal reasoning, and generate informed decisions or responses.

3. Tools, such as Extensions, Functions, and Data Stores, serve as the keys to the outside world for agents, allowing them to interact with external systems and access knowledge beyond their training data. Extensions provide a bridge between agents and external APIs, enabling the execution of API calls and retrieval of real-time information. functions provide a more nuanced control for the developer through the division of labor, allowing agents to generate Function parameters which can be executed client-side. Data Stores provide agents with access to structured or unstructured data, enabling data-driven applications.

The future of agents holds exciting advancements and we've only begun to scratch the surface of what is possible. As tools become more sophisticated and reasoning capabilities are enhanced, agents will be empowered to solve increasingly complex problems. Furthermore, the strategic approach of 'agent chaining' will continue to gain momentum. By combining specialized agents - each excelling in a particular domain or task - we can create a ‘mixture of agent experts’ approach, capable of delivering exceptional results across various industries and problem areas.

It’s important to remember that building complex agent architectures demands an iterative approach. Experimentation and refinement are key to finding solutions for specific business cases and organizational needs. No two agents are created alike due to the generative nature of the foundational models that underpin their architecture. However, by harnessing the strengths of each of these foundational components, we can create impactful applications that extend the capabilities of language models and drive real-world value.

Endnotes

1. Shafran, I., Cao, Y. et al., 2022, 'ReAct: Synergizing Reasoning and Acting in Language Models'. Available at: https://arxiv.org/abs/2210.03629

2. Wei, J., Wang, X. et al., 2023, 'Chain-of-Thought Prompting Elicits Reasoning in Large Language Models'. Available at: https://arxiv.org/pdf/2201.11903.pdf.

3. Wang, X. et al., 2022, 'Self-Consistency Improves Chain of Thought Reasoning in Language Models'. Available at: https://arxiv.org/abs/2203.11171.

4. Diao, S. et al., 2023, 'Active Prompting with Chain-of-Thought for Large Language Models'. Available at: https://arxiv.org/pdf/2302.12246.pdf.

5. Zhang, H. et al., 2023, 'Multimodal Chain-of-Thought Reasoning in Language Models'. Available at: https://arxiv.org/abs/2302.00923.

6. Yao, S. et al., 2023, 'Tree of Thoughts: Deliberate Problem Solving with Large Language Models'. Available at: https://arxiv.org/abs/2305.10601.

7. Long, X., 2023, 'Large Language Model Guided Tree-of-Thought'. Available at: https://arxiv.org/abs/2305.08291.

8. Google. 'Google Gemini Application'. Available at: http://gemini.google.com.

9. Swagger. 'OpenAPI Specification'. Available at: https://swagger.io/specification/.

10. Xie, M., 2022, 'How does in-context learning work? A framework for understanding the differences from traditional supervised learning'. Available at: https://ai.stanford.edu/blog/understanding-incontext/.

11. Google Research. 'ScaNN (Scalable Nearest Neighbors)'. Available at: https://github.com/google-research/google-research/tree/master/scann.

12. LangChain. 'LangChain'. Available at: https://python.langchain.com/v0.2/docs/introduction/.