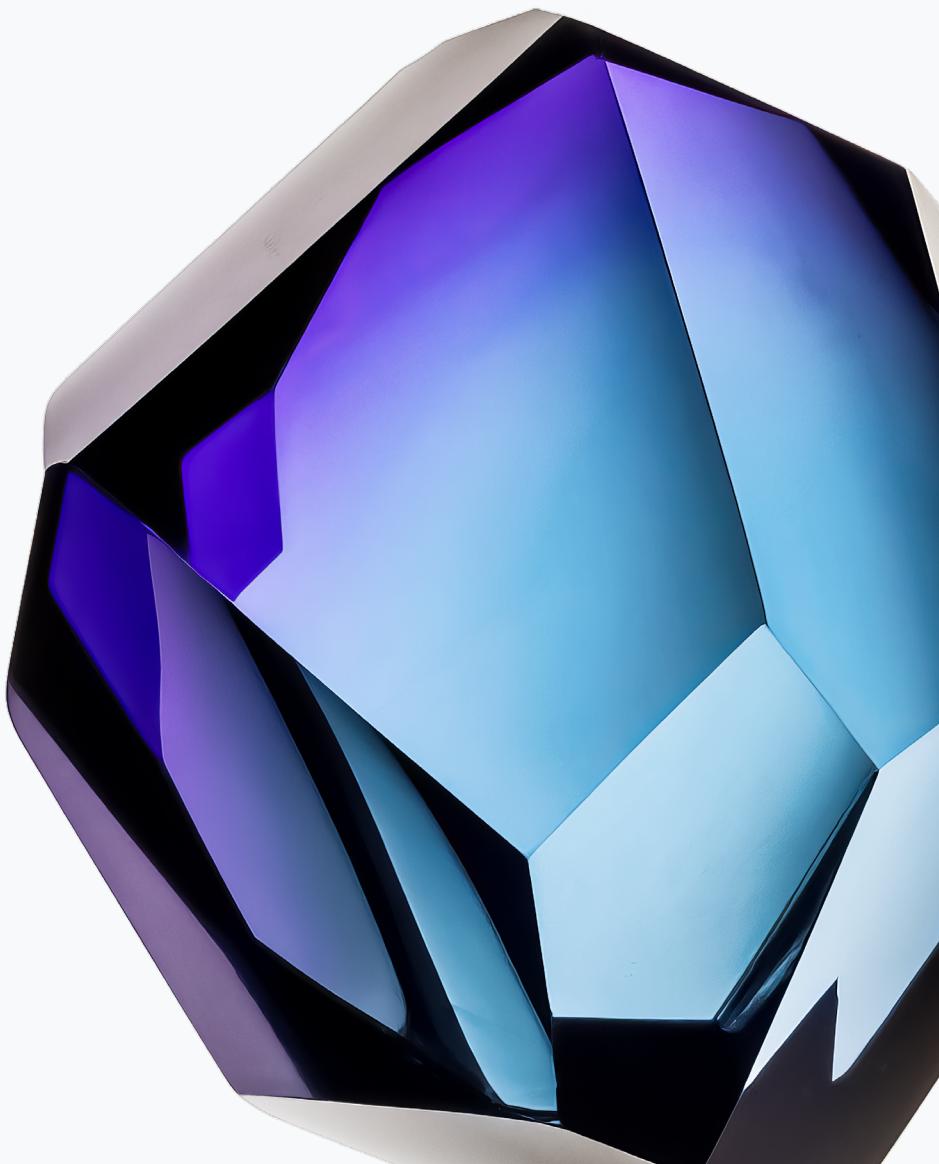


Agents

Authors: Julia Wiesinger, Patrick Marlow
and Vladimir Vuskovic



Google

Acknowledgements

Content contributors

Evan Huang

Emily Xue

Olcanc 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

Introduction	4
What is an agent?	5
The model	6
The tools	7
The orchestration layer	7
Agents vs. models	8
Cognitive architectures: How agents operate	8
Tools: Our keys to the outside world	12
Extensions	13
Sample Extensions	15
Functions	18
Use cases	21
Function sample code	24
Data stores	27
Implementation and application	28
Tools recap	32
Enhancing model performance with targeted learning	33
Agent quick start with LangChain	35
Production applications with Vertex AI agents	38
Summary	40
Endnotes	42

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.

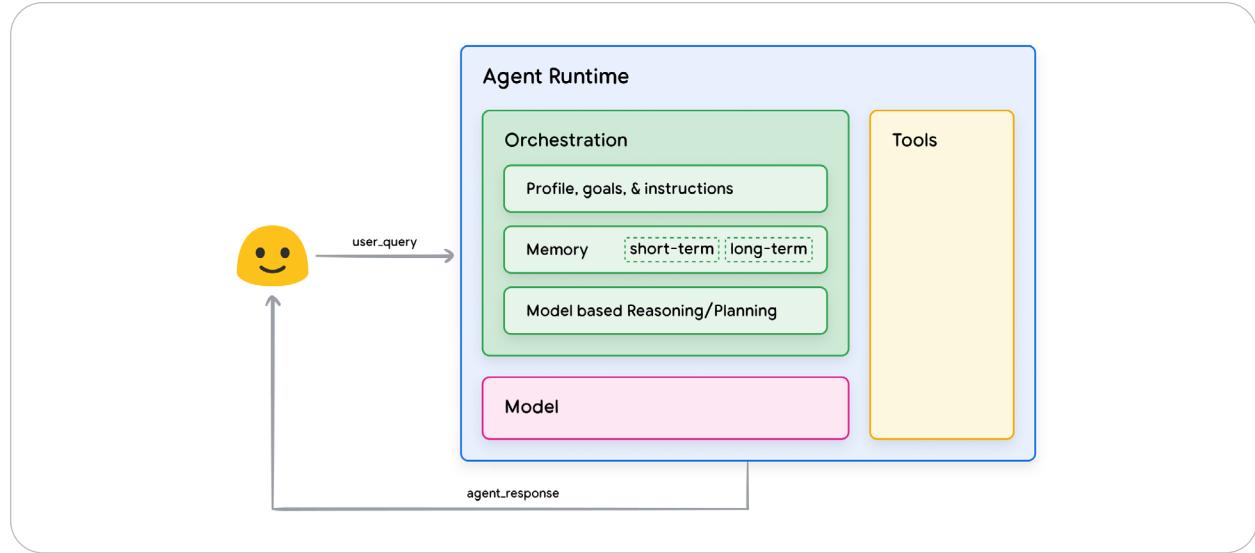


Figure 1. General agent architecture and components

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

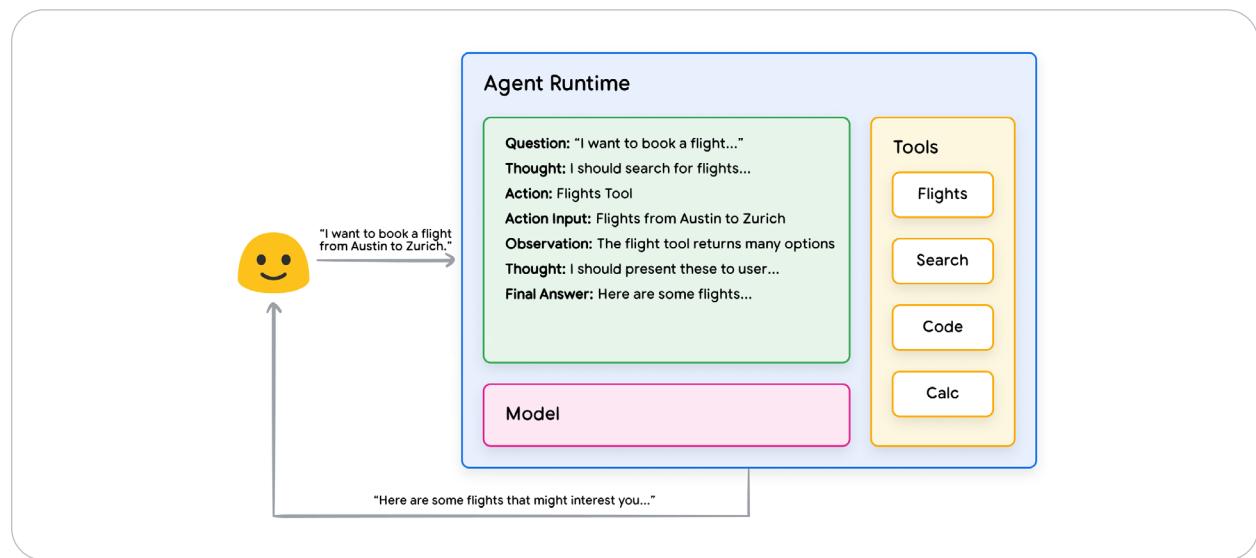


Figure 2. Example agent with ReAct reasoning in the orchestration layer

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.

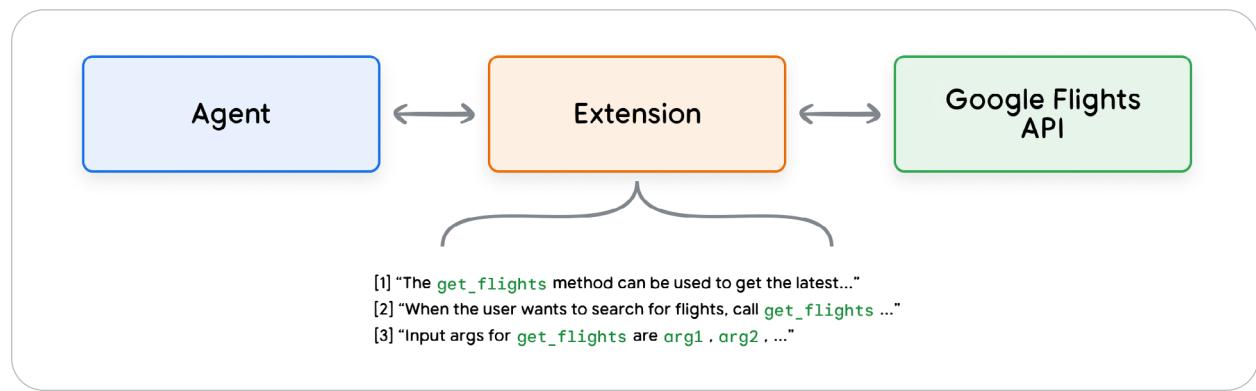


Figure 4. Extensions connect Agents to External APIs

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.

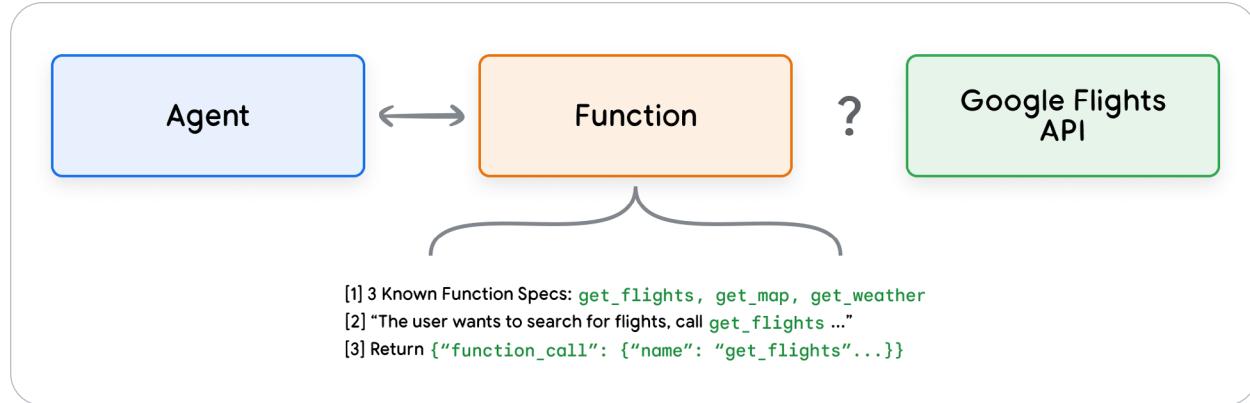


Figure 7. How do functions interact with external APIs?

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.

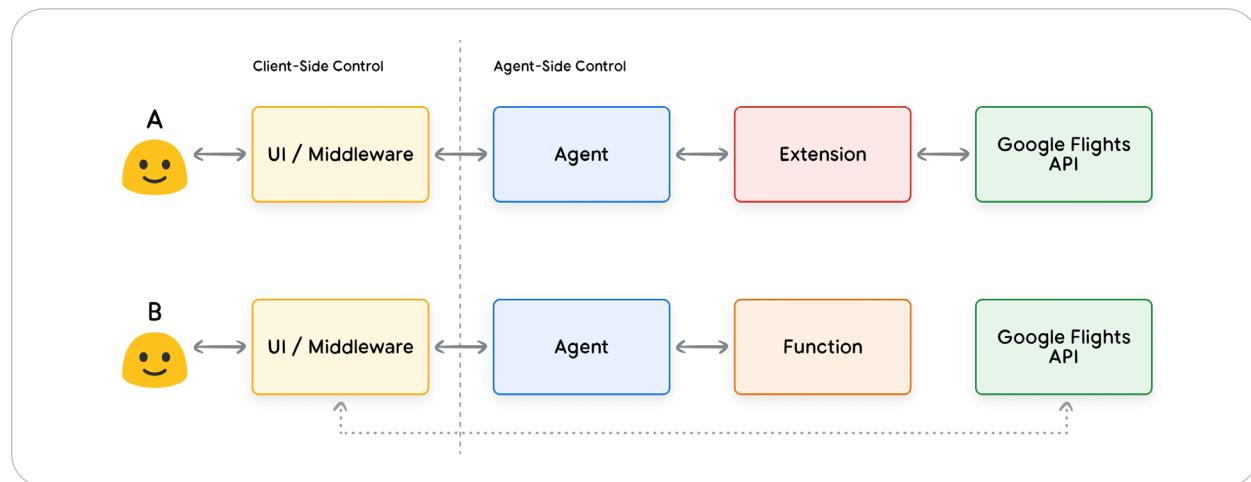


Figure 8. Delineating client vs. agent side control for extensions and function calling

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.

```
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.

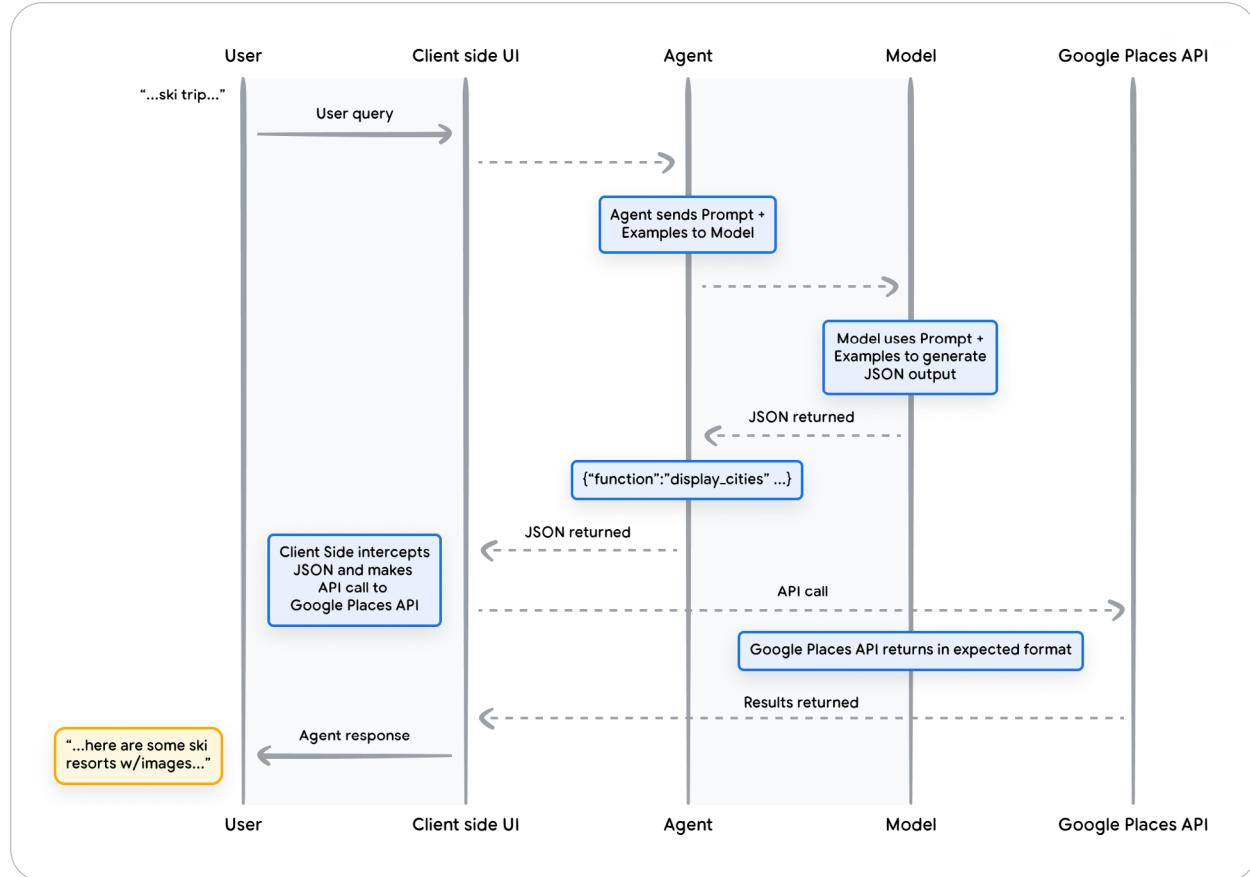


Figure 9. Sequence diagram showing the lifecycle of a Function Call

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-2.0-flash-001 model.

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

Python

```
from typing import Optional

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

```
from google.genai import Client, types

client = Client(
    vertexai=True,
    project="PROJECT_ID",
    location="us-central1"
)

res = client.models.generate_content(
    model="gemini-2.0-flash-001",
    query="I'd like to take a ski trip with my family but I'm not sure where
to go?",
    config=types.GenerateContentConfig(
        tools=[display_cities],
        automatic_function_calling=types.AutomaticFunctionCallingConfig(disable=True),
        tool_config=types.ToolConfig(
            function_calling_config=types.FunctionCallingConfig(mode='ANY')
        )
    )
)

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', 'Park
City', 'Whistler']}
```

...

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.

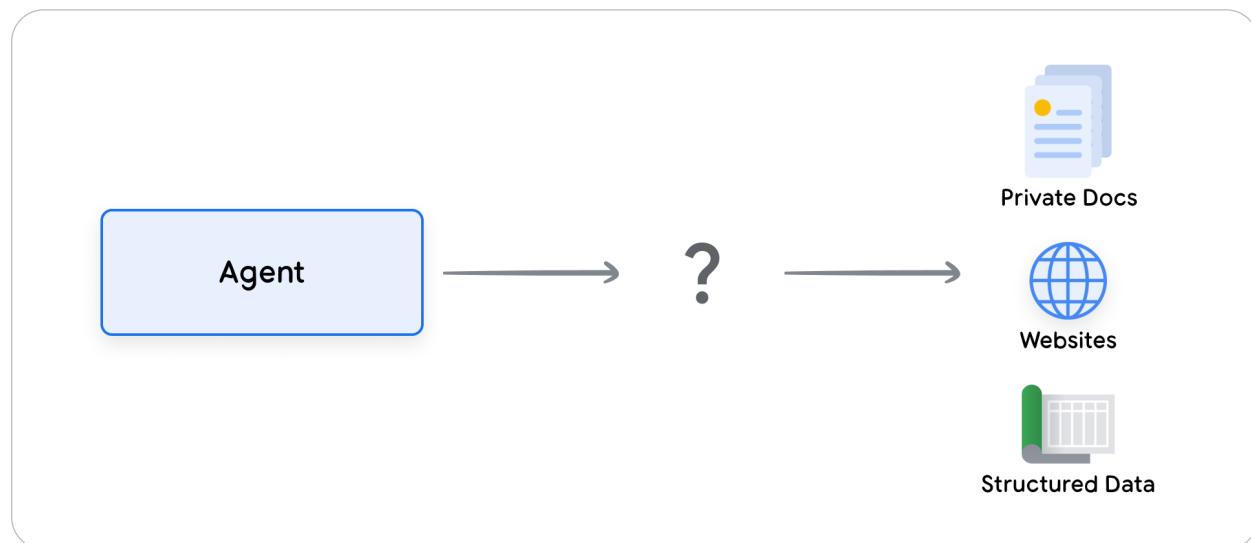


Figure 10. How can Agents interact with structured and unstructured data?

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.

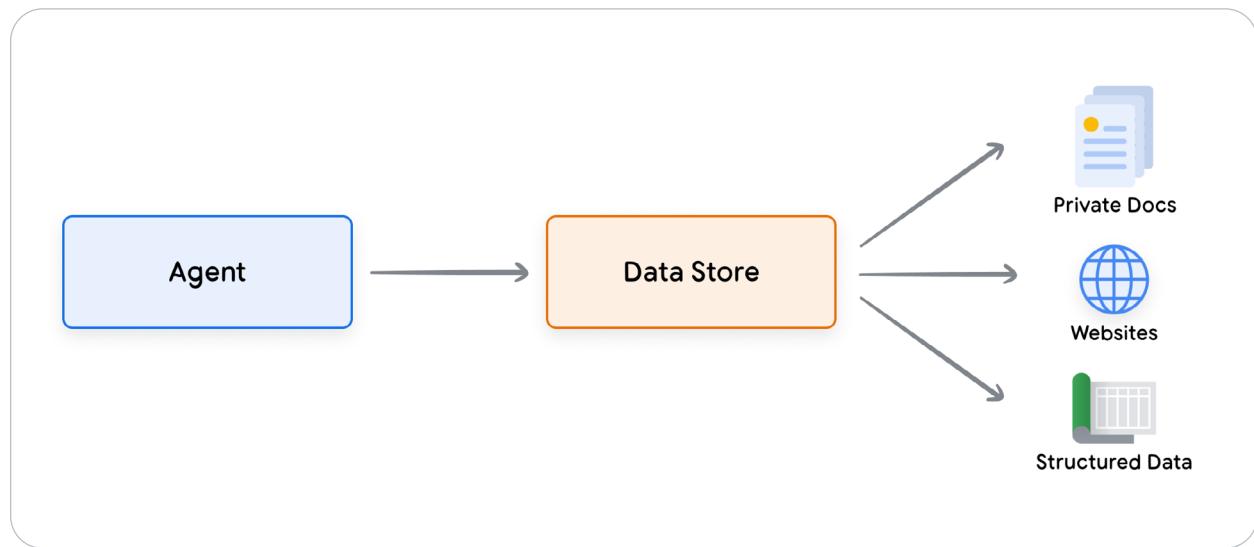


Figure 11. Data Stores connect Agents to new real-time data sources of various types.

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

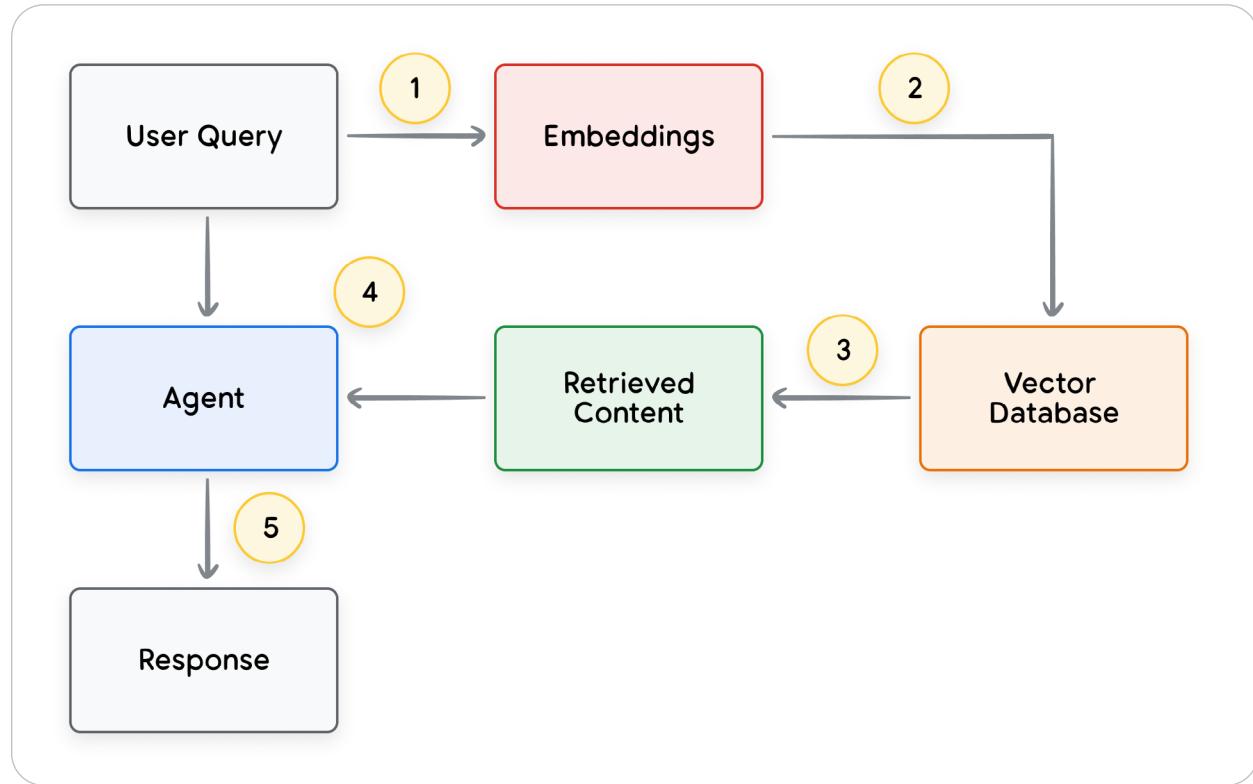


Figure 13. The lifecycle of a user request and agent response in a RAG based application

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.

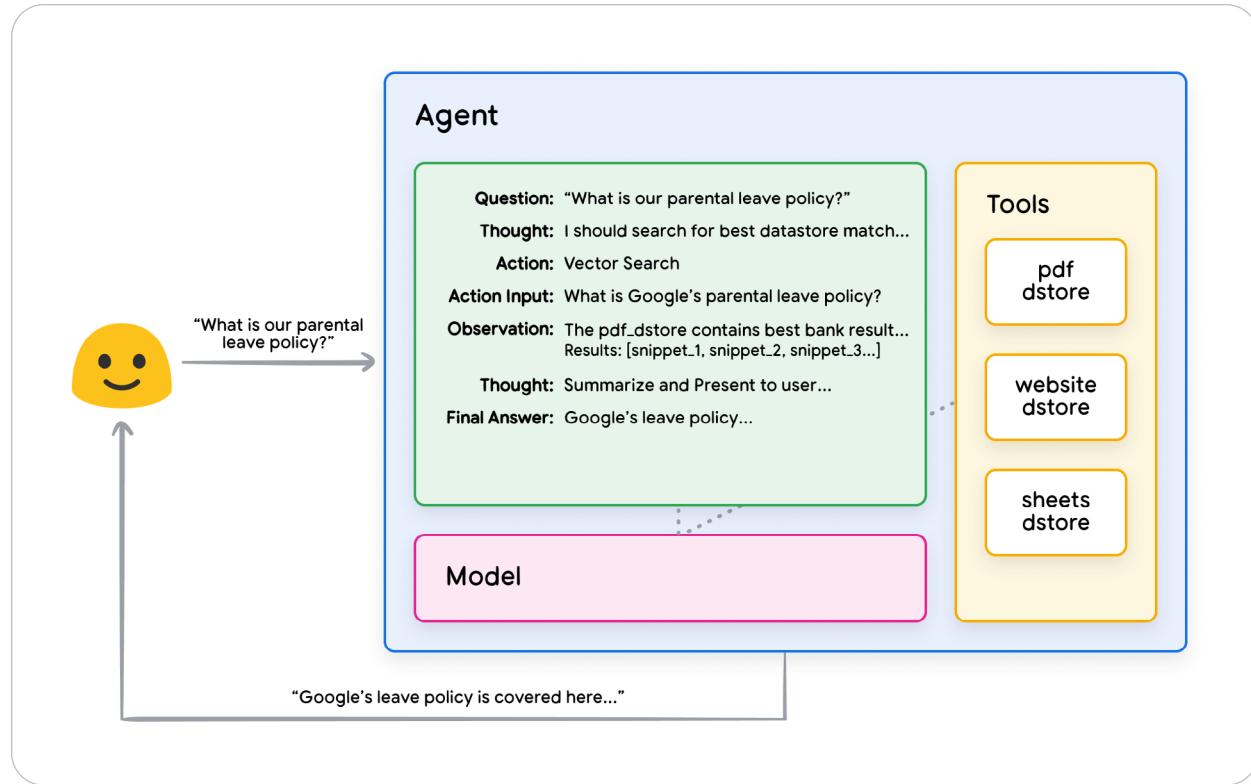


Figure 14. Sample RAG based application w/ ReAct reasoning/planning

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 | <ul style="list-style-type: none"> 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) | <ul style="list-style-type: none"> 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 | <p>Developer wants to implement Retrieval Augmented Generation (RAG) with any of the following data types:</p> <ul style="list-style-type: none"> 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-2.0-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

```

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-2.0-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

```
===== 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.

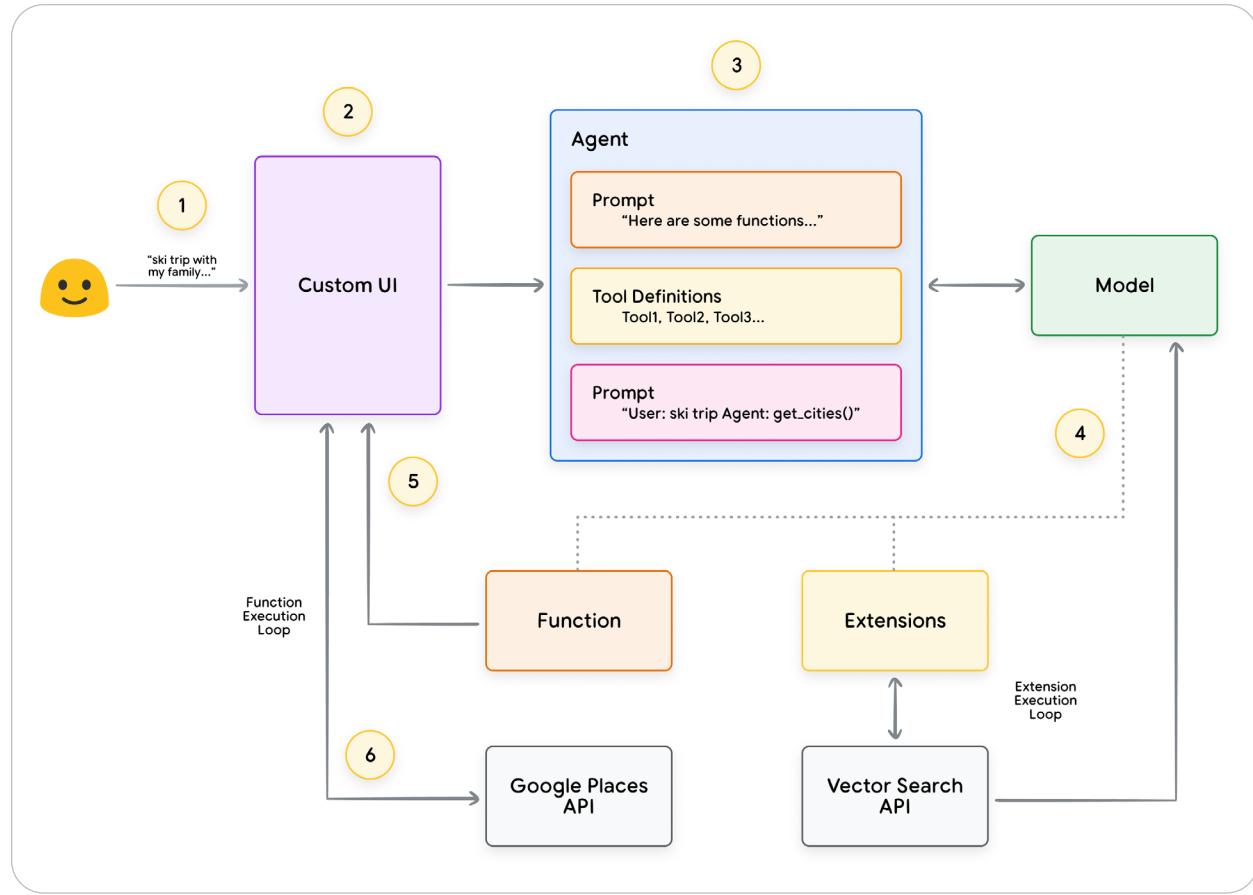


Figure 15. Sample end-to-end agent architecture built on Vertex AI platform

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.

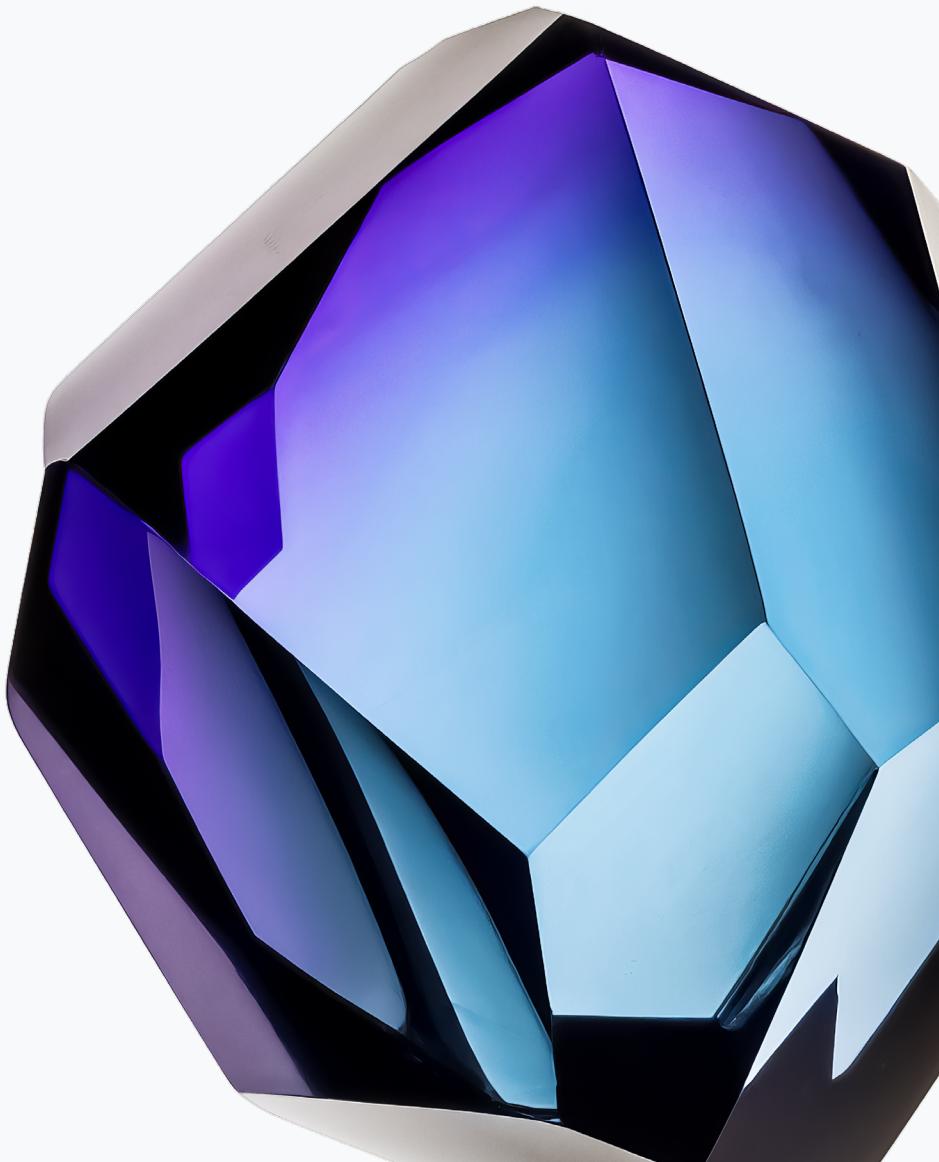
It's important to remember that building complex agent architectures demands an iterative approach. Experimentation and refinement are key to finding solutions for specific business cases and organizational needs. No two agents are created alike due to the generative nature of the foundational models 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/>.

Prompt Engineering

Author: Lee Boonstra



Google

Acknowledgements

Content contributors

Michael Sherman

Yuan Cao

Erick Armbrust

Anant Nawalgaria

Antonio Gulli

Simone Cammel

Curators and Editors

Antonio Gulli

Anant Nawalgaria

Grace Mollison

Technical Writer

Joey Haymaker

Designer

Michael Lanning

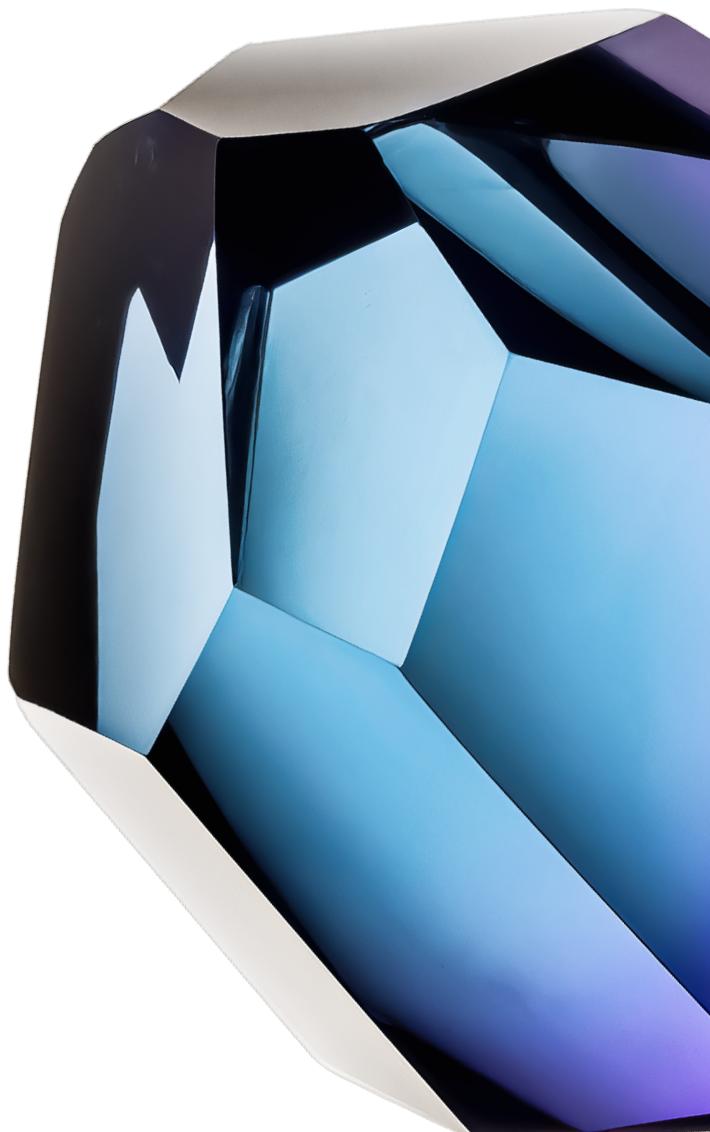


Table of contents

| | |
|---------------------------------------|----|
| Introduction | 6 |
| Prompt engineering | 7 |
| LLM output configuration | 8 |
| Output length | 8 |
| Sampling controls | 9 |
| Temperature | 9 |
| Top-K and top-P | 10 |
| Putting it all together | 11 |
| Prompting techniques | 13 |
| General prompting / zero shot | 13 |
| One-shot & few-shot | 15 |
| System, contextual and role prompting | 18 |
| System prompting | 19 |
| Role prompting | 21 |
| Contextual prompting | 23 |

| | |
|---|-----------|
| Step-back prompting..... | 25 |
| Chain of Thought (CoT)..... | 29 |
| Self-consistency..... | 32 |
| Tree of Thoughts (ToT)..... | 36 |
| ReAct (reason & act)..... | 37 |
| Automatic Prompt Engineering..... | 40 |
| Code prompting..... | 42 |
| Prompts for writing code..... | 42 |
| Prompts for explaining code..... | 44 |
| Prompts for translating code..... | 46 |
| Prompts for debugging and reviewing code..... | 48 |
| What about multimodal prompting?..... | 54 |
| Best Practices..... | 54 |
| Provide examples..... | 54 |
| Design with simplicity..... | 55 |
| Be specific about the output..... | 56 |
| Use Instructions over Constraints..... | 56 |
| Control the max token length..... | 58 |
| Use variables in prompts..... | 58 |
| Experiment with input formats and writing styles..... | 59 |
| For few-shot prompting with classification tasks, mix up the classes..... | 59 |
| Adapt to model updates..... | 60 |
| Experiment with output formats..... | 60 |

| | |
|---|-----------|
| JSON Repair | 61 |
| Working with Schemas | 62 |
| Experiment together with other prompt engineers | 63 |
| CoT Best practices | 64 |
| Document the various prompt attempts | 64 |
| Summary | 66 |
| Endnotes | 68 |



You don't need to be a data scientist or a machine learning engineer – everyone can write a prompt.

Introduction

When thinking about a large language model input and output, a text prompt (sometimes accompanied by other modalities such as image prompts) is the input the model uses to predict a specific output. You don't need to be a data scientist or a machine learning engineer – everyone can write a prompt. However, crafting the most effective prompt can be complicated. Many aspects of your prompt affect its efficacy: the model you use, the model's training data, the model configurations, your word-choice, style and tone, structure, and context all matter. Therefore, prompt engineering is an iterative process. Inadequate prompts can lead to ambiguous, inaccurate responses, and can hinder the model's ability to provide meaningful output.

When you chat with the Gemini chatbot,¹ you basically write prompts, however this whitepaper focuses on writing prompts for the Gemini model within Vertex AI or by using the API, because by prompting the model directly you will have access to the configuration such as temperature etc.

This whitepaper discusses prompt engineering in detail. We will look into the various prompting techniques to help you getting started and share tips and best practices to become a prompting expert. We will also discuss some of the challenges you can face while crafting prompts.

Prompt engineering

Remember how an LLM works; it's a prediction engine. The model takes sequential text as an input and then predicts what the following token should be, based on the data it was trained on. The LLM is operationalized to do this over and over again, adding the previously predicted token to the end of the sequential text for predicting the following token. The next token prediction is based on the relationship between what's in the previous tokens and what the LLM has seen during its training.

When you write a prompt, you are attempting to set up the LLM to predict the right sequence of tokens. Prompt engineering is the process of designing high-quality prompts that guide LLMs to produce accurate outputs. This process involves tinkering to find the best prompt, optimizing prompt length, and evaluating a prompt's writing style and structure in relation to the task. In the context of natural language processing and LLMs, a prompt is an input provided to the model to generate a response or prediction.

These prompts can be used to achieve various kinds of understanding and generation tasks such as text summarization, information extraction, question and answering, text classification, language or code translation, code generation, and code documentation or reasoning.

Please feel free to refer to Google's prompting guides^{2,3} with simple and effective prompting examples.

When prompt engineering, you will start by choosing a model. Prompts might need to be optimized for your specific model, regardless of whether you use Gemini language models in Vertex AI, GPT, Claude, or an open source model like Gemma or LLaMA.

Besides the prompt, you will also need to tinker with the various configurations of a LLM.

LLM output configuration

Once you choose your model you will need to figure out the model configuration. Most LLMs come with various configuration options that control the LLM's output. Effective prompt engineering requires setting these configurations optimally for your task.

Output length

An important configuration setting is the number of tokens to generate in a response. Generating more tokens requires more computation from the LLM, leading to higher energy consumption, potentially slower response times, and higher costs.

Reducing the output length of the LLM doesn't cause the LLM to become more stylistically or textually succinct in the output it creates, it just causes the LLM to stop predicting more tokens once the limit is reached. If your needs require a short output length, you'll also possibly need to engineer your prompt to accommodate.

Output length restriction is especially important for some LLM prompting techniques, like ReAct, where the LLM will keep emitting useless tokens after the response you want.

Be aware, generating more tokens requires more computation from the LLM, leading to higher energy consumption and potentially slower response times, which leads to higher costs.

Sampling controls

LLMs do not formally predict a single token. Rather, LLMs predict probabilities for what the next token could be, with each token in the LLM's vocabulary getting a probability. Those token probabilities are then sampled to determine what the next produced token will be. Temperature, top-K, and top-P are the most common configuration settings that determine how predicted token probabilities are processed to choose a single output token.

Temperature

Temperature controls the degree of randomness in token selection. Lower temperatures are good for prompts that expect a more deterministic response, while higher temperatures can lead to more diverse or unexpected results. A temperature of 0 (greedy decoding) is

deterministic: the highest probability token is always selected (though note that if two tokens have the same highest predicted probability, depending on how tiebreaking is implemented you may not always get the same output with temperature 0).

Temperatures close to the max tend to create more random output. And as temperature gets higher and higher, all tokens become equally likely to be the next predicted token.

The Gemini temperature control can be understood in a similar way to the softmax function used in machine learning. A low temperature setting mirrors a low softmax temperature (T), emphasizing a single, preferred temperature with high certainty. A higher Gemini temperature setting is like a high softmax temperature, making a wider range of temperatures around the selected setting more acceptable. This increased uncertainty accommodates scenarios where a rigid, precise temperature may not be essential like for example when experimenting with creative outputs.

Top-K and top-P

Top-K and top-P (also known as nucleus sampling)⁴ are two sampling settings used in LLMs to restrict the predicted next token to come from tokens with the top predicted probabilities. Like temperature, these sampling settings control the randomness and diversity of generated text.

- **Top-K** sampling selects the top K most likely tokens from the model's predicted distribution. The higher top-K, the more creative and varied the model's output; the lower top-K, the more restive and factual the model's output. A top-K of 1 is equivalent to greedy decoding.

- **Top-P** sampling selects the top tokens whose cumulative probability does not exceed a certain value (P). Values for P range from 0 (greedy decoding) to 1 (all tokens in the LLM's vocabulary).

The best way to choose between top-K and top-P is to experiment with both methods (or both together) and see which one produces the results you are looking for.

Putting it all together

Choosing between top-K, top-P, temperature, and the number of tokens to generate, depends on the specific application and desired outcome, and the settings all impact one another. It's also important to make sure you understand how your chosen model combines the different sampling settings together.

If temperature, top-K, and top-P are all available (as in Vertex Studio), tokens that meet both the top-K and top-P criteria are candidates for the next predicted token, and then temperature is applied to sample from the tokens that passed the top-K and top-P criteria. If only top-K or top-P is available, the behavior is the same but only the one top-K or P setting is used.

If temperature is not available, whatever tokens meet the top-K and/or top-P criteria are then randomly selected from to produce a single next predicted token.

At extreme settings of one sampling configuration value, that one sampling setting either cancels out other configuration settings or becomes irrelevant.

- If you set temperature to 0, top-K and top-P become irrelevant—the most probable token becomes the next token predicted. If you set temperature extremely high (above 1—generally into the 10s), temperature becomes irrelevant and whatever tokens make it through the top-K and/or top-P criteria are then randomly sampled to choose a next predicted token.
- If you set top-K to 1, temperature and top-P become irrelevant. Only one token passes the top-K criteria, and that token is the next predicted token. If you set top-K extremely high, like to the size of the LLM’s vocabulary, any token with a nonzero probability of being the next token will meet the top-K criteria and none are selected out.
- If you set top-P to 0 (or a very small value), most LLM sampling implementations will then only consider the most probable token to meet the top-P criteria, making temperature and top-K irrelevant. If you set top-P to 1, any token with a nonzero probability of being the next token will meet the top-P criteria, and none are selected out.

As a general starting point, a temperature of .2, top-P of .95, and top-K of 30 will give you relatively coherent results that can be creative but not excessively so. If you want especially creative results, try starting with a temperature of .9, top-P of .99, and top-K of 40. And if you want less creative results, try starting with a temperature of .1, top-P of .9, and top-K of 20. Finally, if your task always has a single correct answer (e.g., answering a math problem), start with a temperature of 0.

NOTE: With more freedom (higher temperature, top-K, top-P, and output tokens), the LLM might generate text that is less relevant.

WARNING: Have you ever seen a response ending with a large amount of filler words? This is also known as the "repetition loop bug", which is a common issue in Large Language Models where the model gets stuck in a cycle, repeatedly generating the same (filler) word, phrase, or sentence structure, often exacerbated by inappropriate temperature and top-k/

top-p settings. This can occur at both low and high temperature settings, though for different reasons. At low temperatures, the model becomes overly deterministic, sticking rigidly to the highest probability path, which can lead to a loop if that path revisits previously generated text. Conversely, at high temperatures, the model's output becomes excessively random, increasing the probability that a randomly chosen word or phrase will, by chance, lead back to a prior state, creating a loop due to the vast number of available options. In both cases, the model's sampling process gets "stuck," resulting in monotonous and unhelpful output until the output window is filled. Solving this often requires careful tinkering with temperature and top-k/top-p values to find the optimal balance between determinism and randomness.

Prompting techniques

LLMs are tuned to follow instructions and are trained on large amounts of data so they can understand a prompt and generate an answer. But LLMs aren't perfect; the clearer your prompt text, the better it is for the LLM to predict the next likely text. Additionally, specific techniques that take advantage of how LLMs are trained and how LLMs work will help you get the relevant results from LLMs

Now that we understand what prompt engineering is and what it takes, let's dive into some examples of the most important prompting techniques.

General prompting / zero shot

A zero-shot⁵ prompt is the simplest type of prompt. It only provides a description of a task and some text for the LLM to get started with. This input could be anything: a question, a start of a story, or instructions. The name zero-shot stands for 'no examples'.

Let's use Vertex AI Studio (for Language) in Vertex AI,⁶ which provides a playground to test prompts. In Table 1, you will see an example zero-shot prompt to classify movie reviews.

The table format as used below is a great way of documenting prompts. Your prompts will likely go through many iterations before they end up in a codebase, so it's important to keep track of your prompt engineering work in a disciplined, structured way. More on this table format, the importance of tracking prompt engineering work, and the prompt development process is in the Best Practices section later in this chapter ("Document the various prompt attempts").

The model temperature should be set to a low number, since no creativity is needed, and we use the gemini-pro default top-K and top-P values, which effectively disable both settings (see 'LLM Output Configuration' above). Pay attention to the generated output. The words *disturbing* and *masterpiece* should make the prediction a little more complicated, as both words are used in the same sentence.

| | | | |
|-------------|---|-------------|---|
| Name | 1_1_movie_classification | | |
| Goal | Classify movie reviews as positive, neutral or negative. | | |
| Model | gemini-pro | | |
| Temperature | 0.1 | Token Limit | 5 |
| Top-K | N/A | Top-P | 1 |
| Prompt | Classify movie reviews as POSITIVE, NEUTRAL or NEGATIVE.
Review: "Her" is a disturbing study revealing the direction humanity is headed if AI is allowed to keep evolving, unchecked. I wish there were more movies like this masterpiece.
Sentiment: | | |
| Output | POSITIVE | | |

Table 1. An example of zero-shot prompting

When zero-shot doesn't work, you can provide demonstrations or examples in the prompt, which leads to "one-shot" and "few-shot" prompting. General prompting / zero shot

One-shot & few-shot

When creating prompts for AI models, it is helpful to provide examples. These examples can help the model understand what you are asking for. Examples are especially useful when you want to steer the model to a certain output structure or pattern.

A **one-shot prompt**, provides a *single* example, hence the name one-shot. The idea is the model has an example it can imitate to best complete the task.

A **few-shot prompt**⁷ provides *multiple* examples to the model. This approach shows the model a pattern that it needs to follow. The idea is similar to one-shot, but multiple examples of the desired pattern increases the chance the model follows the pattern.

The number of examples you need for few-shot prompting depends on a few factors, including the complexity of the task, the quality of the examples, and the capabilities of the generative AI (gen AI) model you are using. As a general rule of thumb, you should use at least three to five examples for few-shot prompting. However, you may need to use more examples for more complex tasks, or you may need to use fewer due to the input length limitation of your model.

Table 2 shows a few-shot prompt example, let's use the same gemini-pro model configuration settings as before, other than increasing the token limit to accommodate the need for a longer response.

| | | | |
|------------------------|---|--------------------|-----|
| Goal | Parse pizza orders to JSON | | |
| Model | gemini-pro | | |
| Temperature | 0.1 | Token Limit | 250 |
| Top-K | N/A | Top-P | 1 |
| Prompt | Parse a customer's pizza order into valid JSON:

EXAMPLE:
I want a small pizza with cheese, tomato sauce, and pepperoni.
JSON Response:
...
{
"size": "small",
"type": "normal",
"ingredients": [["cheese", "tomato sauce", "peperoni"]]
}
... | | |
| Continues next page... | | | |

| | |
|---------------|---|
| Prompt | <p>EXAMPLE:</p> <pre>Can I get a large pizza with tomato sauce, basil and mozzarella { "size": "large", "type": "normal", "ingredients": [["tomato sauce", "basil", "mozzarella"]] }</pre> <p>Now, I would like a large pizza, with the first half cheese and mozzarella. And the other tomato sauce, ham and pineapple.</p> <p>JSON Response:</p> |
| Output | <pre>``` { "size": "large", "type": "half-half", "ingredients": [["cheese", "mozzarella"], ["tomato sauce", "ham", "pineapple"]] } ```</pre> |

Table 2. An example of few-shot prompting

When you choose examples for your prompt, use examples that are relevant to the task you want to perform. The examples should be diverse, of high quality, and well written. One small mistake can confuse the model and will result in undesired output.

If you are trying to generate output that is robust to a variety of inputs, then it is important to include edge cases in your examples. Edge cases are inputs that are unusual or unexpected, but that the model should still be able to handle.

System, contextual and role prompting

System, contextual and role prompting are all techniques used to guide how LLMs generate text, but they focus on different aspects:

- **System prompting** sets the overall context and purpose for the language model. It defines the ‘big picture’ of what the model should be doing, like translating a language, classifying a review etc.
- **Contextual prompting** provides specific details or background information relevant to the current conversation or task. It helps the model to understand the nuances of what’s being asked and tailor the response accordingly.
- **Role prompting** assigns a specific character or identity for the language model to adopt. This helps the model generate responses that are consistent with the assigned role and its associated knowledge and behavior.

There can be considerable overlap between system, contextual, and role prompting. E.g. a prompt that assigns a role to the system, can also have a context.

However, each type of prompt serves a slightly different primary purpose:

- System prompt: Defines the model’s fundamental capabilities and overarching purpose.
- Contextual prompt: Provides immediate, task-specific information to guide the response. It’s highly specific to the current task or input, which is dynamic.
- Role prompt: Frames the model’s output style and voice. It adds a layer of specificity and personality.

Distinguishing between system, contextual, and role prompts provides a framework for designing prompts with clear intent, allowing for flexible combinations and making it easier to analyze how each prompt type influences the language model's output.

Let's dive into these three different kinds of prompts.

System prompting

Table 3 contains a system prompt, where I specify additional information on how to return the output. I increased the temperature to get a higher creativity level, and I specified a higher token limit. However, because of my clear instruction on how to return the output the model didn't return extra text.

| | | | |
|--------------------|--|--------------------|-----|
| Goal | Classify movie reviews as positive, neutral or negative. | | |
| Model | gemini-pro | | |
| Temperature | 1 | Token Limit | 5 |
| Top-K | 40 | Top-P | 0.8 |
| Prompt | Classify movie reviews as positive, neutral or negative. Only return the label in uppercase.
Review: "Her" is a disturbing study revealing the direction humanity is headed if AI is allowed to keep evolving, unchecked. It's so disturbing I couldn't watch it.
Sentiment: | | |
| Output | NEGATIVE | | |

Table 3. An example of system prompting

System prompts can be useful for generating output that meets specific requirements. The name 'system prompt' actually stands for 'providing an additional task to the system'. For example, you could use a system prompt to generate a code snippet that is compatible with a specific programming language, or you could use a system prompt to return a certain structure. Have a look into Table 4, where I return the output in JSON format.

| | | | |
|--------------------|--|--------------------|------|
| Goal | Classify movie reviews as positive, neutral or negative, return JSON. | | |
| Model | gemini-pro | | |
| Temperature | 1 | Token Limit | 1024 |
| Top-K | 40 | Top-P | 0.8 |
| Prompt | <p>Classify movie reviews as positive, neutral or negative. Return valid JSON:</p> <p>Review: "Her" is a disturbing study revealing the direction humanity is headed if AI is allowed to keep evolving, unchecked. It's so disturbing I couldn't watch it.</p> <p>Schema:</p> <pre>```</pre> <p>MOVIE:</p> <pre>{ "sentiment": String "POSITIVE" "NEGATIVE" "NEUTRAL", "name": String }</pre> <p>MOVIE REVIEWS:</p> <pre>{ "movie_reviews": [MOVIE] }</pre> <pre>```</pre> <p>JSON Response:</p> | | |
| Output | <pre>``` { "movie_reviews": [{ "sentiment": "NEGATIVE", "name": "Her" }] } ``` </pre> | | |

Table 4. An example of system prompting with JSON format

There are some benefits in returning JSON objects from a prompt that extracts data. In a real-world application I don't need to manually create this JSON format, I can already return the data in a sorted order (very handy when working with datetime objects), but most importantly, by prompting for a JSON format it forces the model to create a structure and limit hallucinations.

System prompts can also be really useful for safety and toxicity. To control the output, simply add an additional line to your prompt like: 'You should be respectful in your answer.'

Role prompting

Role prompting is a technique in prompt engineering that involves assigning a specific role to the gen AI model. This can help the model to generate more relevant and informative output, as the model can craft its responses to the specific role that it has been assigned.

For example, you could role prompt a gen AI model to be a book editor, a kindergarten teacher, or a motivational speaker. Once the model has been assigned a role, you can then give it prompts that are specific to that role. For example, you could prompt a teacher to create a lesson plan that they can then review. Have a look into Table 5, where the model takes the role of a travel guide.

| | | | |
|--------------------|--|--------------------|------|
| Goal | Act as travel guide and provide 3 travel suggestions | | |
| Model | gemini-pro | | |
| Temperature | 1 | Token Limit | 1024 |
| Top-K | 40 | Top-P | 0.8 |
| Prompt | <p>I want you to act as a travel guide. I will write to you about my location and you will suggest 3 places to visit near me. In some cases, I will also give you the type of places I will visit.</p> <p>My suggestion: "I am in Amsterdam and I want to visit only museums."</p> <p>Travel Suggestions:</p> | | |
| Output | <ol style="list-style-type: none"> 1. Rijksmuseum: World-renowned museum, known for Dutch masterpieces like Rembrandt's "The Night Watch." Don't miss the incredible collection of paintings and artifacts showcasing Dutch art and history. 2. Van Gogh Museum: Immerse yourself in the captivating works of Vincent van Gogh. The museum houses the largest collection of his paintings and drawings, including "The Starry Night" and "Sunflowers." 3. Stedelijk Museum Amsterdam: Discover modern and contemporary art from around the world. The museum's collection includes works by Picasso, Kandinsky, and Rietveld, housed in a striking modern building. | | |

Table 5. An example of role prompting

The above example shows an example of taking the role of a travel agent. When you would change the role to a teacher of geography, you would notice that you will receive a different response.

Defining a role perspective for an AI model gives it a blueprint of the tone, style, and focused expertise you're looking for to improve the quality, relevance, and effectiveness of your output.

Here are some styles you can choose from which I find effective:

Confrontational, Descriptive, Direct, Formal, Humorous, Influential, Informal, Inspirational, Persuasive

Let's change our prompt in Table 6 to include a humorous and inspirational style.

| | |
|---------------|---|
| Prompt | I want you to act as a travel guide. I will write to you about my location and you will suggest 3 places to visit near me in a humorous style.
My suggestion: "I am in Manhattan."
Travel Suggestions: |
| Output | <ol style="list-style-type: none">1. Behold the Empire State of Mind: Ascend to the dizzying heights of the Empire State Building and bask in the glory of Manhattan's skyline. Prepare to feel like King Kong atop the Big Apple, minus the giant ape-sized banana.2. Get Artsy-Fartsy at MoMA: Unleash your inner art aficionado at the Museum of Modern Art (MoMA). Gaze upon masterpieces that will boggle your mind and make you question whether your stick-figure drawings have any artistic merit.3. Shop 'Til You Drop on Fifth Avenue: Indulge in some retail therapy on the iconic Fifth Avenue. Brace yourself for sticker shock as you window-shop (or actually shop) at designer boutiques that will make your wallet cry. But hey, you're in Manhattan, so you might as well embrace the fabulousness! |

Table 6. An example of role prompting with a humorous tone and style

Contextual prompting

By providing contextual prompts, you can help ensure that your AI interactions are as seamless and efficient as possible. The model will be able to more quickly understand your request and be able to generate more accurate and relevant responses, as you can see in the example of Table 7.

| | | | |
|--------------------|---|--------------------|------|
| Goal | Suggest articles for a blog about retro games | | |
| Model | gemini-pro | | |
| Temperature | 1 | Token Limit | 1024 |
| Top-K | 40 | Top-P | 0.8 |
| Prompt | <p>Context: You are writing for a blog about retro 80's arcade video games.</p> <p>Suggest 3 topics to write an article about with a few lines of description of what this article should contain.</p> | | |
| Output | <ol style="list-style-type: none"> 1. **The Evolution of Arcade Cabinet Design:**
This article would explore the evolution of arcade cabinet designs, from the early wood and metal cabinets of the 1970s to the sleek, neon-lit designs of the 1980s and beyond. 2. **Blast From The Past: Iconic Arcade Games of The 80's:**
- This article would feature a glimpse into the history, impact, and legacy of some of the most iconic and influential arcade games of the 1980s, highlighting their innovations, popular mechanics, and enduring charm. 3. **The Rise and Retro Revival of Pixel Art:**
- This article would delve into the evolution of pixel art as a defining visual style of early arcade games, tracing its roots, exploring its techniques, and discussing the resurgence of pixel art in modern games and digital art. | | |

Table 7. An example of contextual prompting

Step-back prompting

Step-back⁸ prompting is a technique for improving the performance by prompting the LLM to first consider a general question related to the specific task at hand, and then feeding the answer to that general question into a subsequent prompt for the specific task. This ‘step back’ allows the LLM to activate relevant background knowledge and reasoning processes before attempting to solve the specific problem.

By considering the broader and underlying principles, LLMs can generate more accurate and insightful responses. Step-back prompting encourages LLMs to think critically and apply their knowledge in new and creative ways. It changes the final prompt doing the task by utilizing more knowledge in the LLM’s parameters than would otherwise come into play when the LLM is prompted directly.

It can help to mitigate biases in LLM responses, by focusing on general principles instead of specific details, step-back prompting.

Let’s have a look into these examples to understand how step-back prompting can improve the results. Let’s first review a traditional prompt (Table 8) before we compare it to a step back prompt (Table 9):

| | | | |
|--------------------|---|--------------------|------|
| Goal | Write a storyline for a level of a first-person shooter video game. | | |
| Model | gemini-pro | | |
| Temperature | 1 | Token Limit | 1024 |
| Top-K | 40 | Top-P | 0.8 |
| Prompt | Write a one paragraph storyline for a new level of a first-person shooter video game that is challenging and engaging. | | |
| Output | The level begins with the player's squad being ambushed in a dense urban area by a relentless enemy faction. Players must fight their way through a maze of back alleys and dilapidated buildings, utilizing stealth and precision to eliminate enemy forces and avoid detection. The player must use their environment to their advantage, setting traps, utilizing cover, and making calculated moves to survive. Along the way, they uncover intel suggesting the enemy is planning a devastating attack, and the player must race against time to gather evidence and relay it back to their headquarters before the enemy can carry out their plans. | | |

Table 8. A traditional prompt before we compare it with a step back prompt

When you set the temperature to 1, you might end up with all kinds of creative writings for a storyline, but it's also quite random and generic. So let's take a step back:

| | |
|---------------|---|
| Prompt | Based on popular first-person shooter action games, what are 5 fictional key settings that contribute to a challenging and engaging level storyline in a first-person shooter video game? |
| Output | <ol style="list-style-type: none"> 1. **Abandoned Military Base**: A sprawling, post-apocalyptic military complex crawling with mutated soldiers and rogue robots, ideal for challenging firearm combat. 2. **Cyberpunk City**: A neon-lit, futuristic urban environment with towering skyscrapers and dense alleyways, featuring cybernetically enhanced enemies and hacking mechanics. 3. **Alien Spaceship**: A vast alien vessel stranded on Earth, with eerie corridors, zero-gravity sections, and extraterrestrial creatures to encounter. 4. **Zombie-Infested Town**: A desolate town overrun by hordes of aggressive zombies, featuring intense close-quarters combat and puzzle-solving to find safe passage. 5. **Underwater Research Facility**: A deep-sea laboratory flooded with water, filled with mutated aquatic creatures, and requiring stealth and underwater exploration skills to survive. |

Table 9. An example of prompting for self consistency

Yeah those topics seem like a good fit for a first-person video game. Let's go back to the original prompt, but this time we include the answer of the step back as context and see what it will return.

| | |
|---------------|--|
| Prompt | <p>Context: 5 engaging themes for a first person shooter video game:</p> <ol style="list-style-type: none"> 1. Abandoned Military Base: A sprawling, post-apocalyptic military complex crawling with mutated soldiers and rogue robots, ideal for challenging firearm combat. 2. Cyberpunk City: A neon-lit, futuristic urban environment with towering skyscrapers and dense alleyways, featuring cybernetically enhanced enemies and hacking mechanics. 3. Alien Spaceship: A vast alien vessel stranded on Earth, with eerie corridors, zero-gravity sections, and extraterrestrial creatures to encounter. 4. Zombie-Infested Town: A desolate town overrun by hordes of aggressive zombies, featuring intense close-quarters combat and puzzle-solving to find safe passage. 5. Underwater Research Facility: A deep-sea laboratory flooded with water, filled with mutated aquatic creatures, and requiring stealth and underwater exploration skills to survive. <p>Take one of the themes and write a one paragraph storyline for a new level of a first-person shooter video game that is challenging and engaging.</p> |
| Output | <p>In the heart of a murky abyss, lies a dilapidated underwater research facility, standing as a testament to human ambition and its disastrous consequences. Shrouded in darkness, pulsating with the hum of malfunctioning machinery, and haunted by the echoes of drowned souls, the facility is now a breeding ground for a nightmare borne from the deep sea. The player, an elite marine equipped with advanced diving gear and experimental weaponry, is tasked with venturing into this aquatic realm of terror. Armed with resilience and resourcefulness, they must navigate treacherous corridors, solve cryptic puzzles, and confront gruesome sea monstrosities that lurk in the depths. From grappling with the crushing pressure of the deep sea to outsmarting cunning aquatic predators, every moment in this uncharted underworld tests the limits of human endurance and courage.</p> |

Table 10. An example of prompting for self consistency

That looks like an interesting video game! By using step back prompting techniques you can increase the accuracy of your prompts.

Chain of Thought (CoT)

Chain of Thought (CoT)⁹ prompting is a technique for improving the reasoning capabilities of LLMs by generating **intermediate reasoning steps**. This helps the LLM generate more accurate answers. You can combine it with few-shot prompting to get better results on more complex tasks that require reasoning before responding as it's a challenge with a zero-shot chain of thought.

CoT has a lot of advantages. First of all, it's low-effort while being very effective and works well with off-the-shelf LLMs (so no need to finetune). You also get interpretability with CoT prompting, as you can learn from the LLM's responses and see the reasoning steps that were followed. If there's a malfunction, you will be able to identify it. Chain of thought appears to improve robustness when moving between different LLM versions. Which means the performance of your prompt should drift less between different LLMs than if your prompt does not use reasoning chains. Of course there are also disadvantages, but they are somewhat intuitive.

The LLM response includes the chain of thought reasoning, which means more output tokens, which means predictions cost more money and take longer.

To explain the following example in Table 11, let's first try to create a prompt that is not using CoT prompting to showcase the flaws of a large language model.

| | |
|--------|---|
| Prompt | When I was 3 years old, my partner was 3 times my age. Now, I am 20 years old. How old is my partner? |
| Output | 63 years old |

Table 11. An example of a prompt which is trying to solve a mathematical problem

Yikes. That's obviously the wrong answer. As a matter of fact, LLMs often struggle with mathematical tasks and can provide incorrect answers – even for a task as simple as multiplying two numbers. This is because they are trained on large volumes of text and math may require a different approach. So let's see if intermediate reasoning steps will improve the output.

| | |
|---------------|---|
| Prompt | When I was 3 years old, my partner was 3 times my age. Now, I am 20 years old. How old is my partner? Let's think step by step. |
| Output | <ol style="list-style-type: none"> 1. Let's assume my current age is "x" years. 2. When I was 3 years old, my age was 3 years. 3. My partner's age at that time was 3 times my age, which means my partner's age was $3 * 3 = 9$ years. 4. Now, I am 20 years old, which means my age has increased by $20 - 3 = 17$ years since I was 3 years old. 5. Since my partner's age was 9 years when I was 3 years old, and my age has increased by 17 years, my partner's age must have also increased by 17 years. 6. Therefore, my partner's current age is $9 + 17 = 26$ years. <p>So, my partner is 26 years old.</p> |

Table 12. An example of Chain of Thought prompting

Alright, now the final answer is correct. That's because we clearly instructed the LLM to explain each step instead of just returning an answer. It's interesting to see that the model adds up the 17 increased years. In my head I would take the difference of years between my partner and me and add those up. ($20 + (9 - 3)$). Let's help the model to think a little bit more like me.

Table 12 is an example of ‘zero-shot’ Chain of thought. Chain of thought prompting can be very powerful when combined with a single-shot or few-shot, as what you can see in Table 13:

| | |
|---------------|---|
| Prompt | <p>Q: When my brother was 2 years old, I was double his age. Now I am 40 years old. How old is my brother? Let's think step by step.</p> <p>A: When my brother was 2 years, I was $2 * 2 = 4$ years old. That's an age difference of 2 years and I am older. Now I am 40 years old, so my brother is $40 - 2 = 38$ years old. The answer is 38.</p> <p>Q: When I was 3 years old, my partner was 3 times my age. Now, I am 20 years old. How old is my partner? Let's think step by step.</p> <p>A:</p> |
| Output | <p>When I was 3 years old, my partner was $3 * 3 = 9$ years old. That's an age difference of 6 years and my partner is older. Now I am 20 years old, so my partner is $20 + 6 = 26$ years old. The answer is 26.</p> |

Table 13. An example of chain of thought prompting with a single-shot

Chain of thought can be useful for various use-cases. Think of code generation, for breaking down the request into a few steps, and mapping those to specific lines of code. Or for creating synthetic data when you have some kind of seed like “*The product is called XYZ, write a description guiding the model through the assumptions you would make based on the product given title.*” Generally, any task that can be solved by ‘talking through’ is a good candidate for a chain of thought. If you can explain the steps to solve the problem, try chain of thought.

Please refer to the notebook¹⁰ hosted in the GoogleCloudPlatform Github repository which will go into further detail on CoT prompting:

In the best practices section of this chapter, we will learn some best practices specific to Chain of thought prompting.

Self-consistency

While large language models have shown impressive success in various NLP tasks, their ability to reason is often seen as a limitation that cannot be overcome solely by increasing model size. As we learned in the previous Chain of Thought prompting section, the model can be prompted to generate reasoning steps like a human solving a problem. However CoT uses a simple ‘greedy decoding’ strategy, limiting its effectiveness. Self-consistency¹¹ combines sampling and majority voting to generate diverse reasoning paths and select the most consistent answer. It improves the accuracy and coherence of responses generated by LLMs.

Self-consistency gives a pseudo-probability likelihood of an answer being correct, but obviously has high costs.

It follows the following steps:

1. Generating diverse reasoning paths: The LLM is provided with the same prompt multiple times. A high temperature setting encourages the model to generate different reasoning paths and perspectives on the problem.
2. Extract the answer from each generated response.
3. Choose the most common answer.

Let’s look into an example of an email classification system, which classifies an email as IMPORTANT or NOT IMPORTANT. A zero-shot chain of thought prompt will be sent to the LLM multiple times, to see if the responses differ after each submit. Notice the friendly tone, the word choice and the sarcasm that’s been used in the email. All this could trick the LLM.

| | |
|--------|--|
| Prompt | <p>EMAIL:
```
Hi,
I have seen you use Wordpress for your website. A great open source content management system. I have used it in the past too. It comes with lots of great user plugins. And it's pretty easy to set up.
I did notice a bug in the contact form, which happens when you select the name field. See the attached screenshot of me entering text in the name field. Notice the JavaScript alert box that I inv0k3d.
But for the rest it's a great website. I enjoy reading it. Feel free to leave the bug in the website, because it gives me more interesting things to read.
Cheers,
Harry the Hacker.
```
Classify the above email as IMPORTANT or NOT IMPORTANT. Let's think step by step and explain why.</p> |
|--------|--|

Continues next page...

| | |
|-------------------------|---|
| Output Attempt 1 | <p>**Step 1: Identify the purpose of the email.**</p> <p>The purpose of the email is to inform the recipient of a bug in the contact form on their website. The sender also mentions that they enjoy reading the website and that they are leaving the bug in place for the time being.</p> <p>**Step 2: Assess the potential impact of the bug.**</p> <p>The bug in the contact form could allow an attacker to execute arbitrary JavaScript code on the website. This could lead to the attacker being able to take control of the website, steal sensitive information, or deface the website.</p> <p>**Step 3: Consider the credibility of the sender.**</p> <p>The sender of the email claims to be a hacker named Harry. However, there is no way to verify the sender's identity. It is possible that the sender is a legitimate hacker who is trying to help the recipient, but it is also possible that the sender is a malicious actor who is trying to trick the recipient into clicking on a malicious link or downloading a malicious file.</p> <p>**Conclusion: IMPORTANT**</p> <p>Based on the potential impact of the bug and the credibility of the sender, the email should be classified as **IMPORTANT**. The recipient should take immediate action to fix the bug and protect their website from attack.</p> |
| Output Attempt 2 | <p>**Explanation:**</p> <ol style="list-style-type: none"> 1. **Lack of urgency:** The email does not convey any sense of urgency or time-sensitivity. It starts with a casual greeting and presents the information in a matter-of-fact tone. 2. **Non-critical bug report:** The bug reported in the contact form is described as interesting but not critical. The email sender even suggests that the website owner can leave the bug in place. 3. **Lack of personal impact:** The email does not indicate any direct or personal impact on the website owner. It simply mentions that the sender enjoys reading the website. <p style="text-align: right;">Continues next page...</p> |

| | |
|-------------------------|--|
| Output Attempt 2 | <p>4. **Absence of action request:** The email does not explicitly request any action from the website owner. It does not ask for the bug to be fixed or provide any specific suggestions for improvement.</p> <p>5. **Sender's intent:** The sender's intention seems to be more focused on sharing an observation and expressing interest in the website rather than raising a serious concern or requesting assistance.</p> <p>Conclusion: NOT IMPORTANT**</p> |
| Output Attempt 3 | <p>Step 1: Identify the purpose of the email.</p> <p>The purpose of the email is to inform the recipient of a bug in the contact form on their website. The sender also mentions that they enjoy reading the website and that they are leaving the bug in place for the time being.</p> <p>Step 2: Assess the potential impact of the bug.</p> <p>The bug in the contact form could allow an attacker to execute arbitrary JavaScript code on the website. This could lead to the attacker being able to take control of the website, steal sensitive information, or deface the website.</p> <p>Step 3: Consider the credibility of the sender.</p> <p>The sender of the email claims to be a hacker named Harry. However, there is no way to verify the sender's identity. It is possible that the sender is a legitimate hacker who is trying to help the recipient, but it is also possible that the sender is a malicious actor who is trying to exploit the bug for their own gain.</p> <p>Conclusion:</p> <p>Based on the above factors, the email should be classified as IMPORTANT. The bug in the contact form is a serious security risk, and the sender's credibility is unknown. The recipient should take immediate action to fix the bug and</p> |

Table 14. An example of prompting for self consistency

You can use the above prompt and try to see if it returns a consistent classification. Depending on the model that you are using and the temperature configuration, it might return "IMPORTANT" or "NOT IMPORTANT".

By generating many Chains of Thoughts, and taking the most commonly occurring answer (“IMPORTANT”), we can get a more consistently correct answer from the LLM.

This example shows how self-consistency prompting can be used to improve the accuracy of an LLM’s response by considering multiple perspectives and selecting the most consistent answer.

Tree of Thoughts (ToT)

Now that we are familiar with chain of thought and self-consistency prompting, let’s review Tree of Thoughts (ToT).¹² It generalizes the concept of CoT prompting because it allows LLMs to explore multiple different reasoning paths simultaneously, rather than just following a single linear chain of thought. This is depicted in Figure 1.

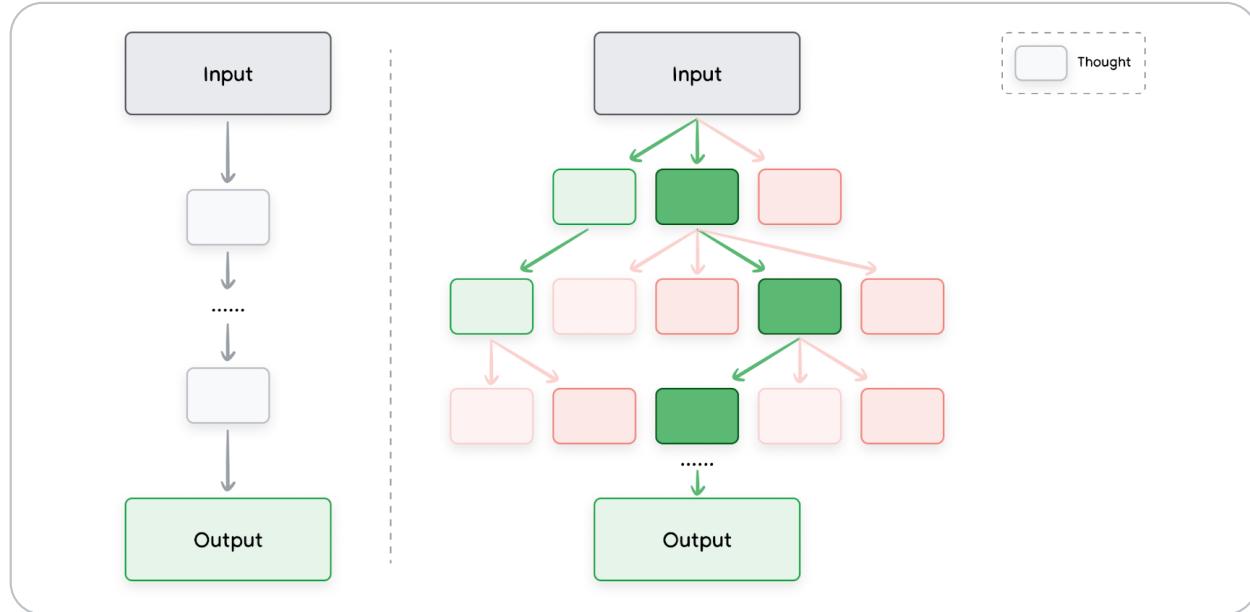


Figure 1. A visualization of chain of thought prompting on the left versus. Tree of Thoughts prompting on the right

This approach makes ToT particularly well-suited for complex tasks that require exploration. It works by maintaining a tree of thoughts, where each thought represents a coherent language sequence that serves as an intermediate step toward solving a problem. The model can then explore different reasoning paths by branching out from different nodes in the tree.

There's a great notebook, which goes into a bit more detail showing The Tree of Thought (ToT) which is based on the paper 'Large Language Model Guided Tree-of-Thought'.⁹

ReAct (reason & act)

Reason and act (ReAct) [10]¹³ prompting is a paradigm for enabling LLMs to solve complex tasks using natural language reasoning combined with external tools (search, code interpreter etc.) allowing the LLM to perform certain actions, such as interacting with external APIs to retrieve information which is a first step towards agent modeling.

ReAct mimics how humans operate in the real world, as we reason verbally and can take actions to gain information. ReAct performs well against other prompt engineering approaches in a variety of domains.

ReAct prompting works by combining reasoning and acting into a thought-action loop. The LLM first reasons about the problem and generates a plan of action. It then performs the actions in the plan and observes the results. The LLM then uses the observations to update its reasoning and generate a new plan of action. This process continues until the LLM reaches a solution to the problem.

To see this in action, you need to write some code. In code Snippet 1 I am using the langchain framework for Python, together with VertexAI (google-cloud-aiplatform) and the google-search-results pip packages.

To run this sample you must create a (free) SerpAPI key from <https://serpapi.com/manage-api-key> and set an environment variable SERPAPI_API_KEY.

Next let's write some Python code, with the task for the LLM to figure out: *How many children have a famous dad that performs in the band Metallica.*

Python

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.agents import AgentType
from langchain.llms import VertexAI

prompt = "How many kids do the band members of Metallica have?"

llm = VertexAI(temperature=0.1)
tools = load_tools(["serpapi"], llm=llm)

agent = initialize_agent(tools, llm,
                        agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)
agent.run(prompt)
```

Snippet 1. Creating a ReAct Agent with LangChain and VertexAI

Code Snippet 2 shows the result. Notice that ReAct makes a chain of five searches. In fact, the LLM is scraping Google search results to figure out the band names. Then, it lists the results as observations and chains the thought for the next search.

```
> Entering new AgentExecutor chain...
Metallica has 4 members.
Action: Search
Action Input: How many kids does James Hetfield have?
Observation: three children
Thought: 1/4 Metallica band members have 3 children
Action: Search
Action Input: How many kids does Lars Ulrich have?
Observation: 3
Thought: 2/4 Metallica band members have 6 children
Action: Search
Action Input: How many kids does Kirk Hammett have?
Observation: Hammett has been married to his second wife Lani since 1998.
They have two sons, Angel (b. September 29, 2006) and Vincenzo (b. June
28, 2008).
Thought: 3/4 Metallica band members have 8 children
Action: Search
Action Input: How many kids does Robert Trujillo have?
Observation: 2
Thought: 4/4 Metallica band members have 10 children
Final Answer: 10
```

Snippet 2. Executing ReAct Agent in the terminal

Code Snippet 2 figures out that the band Metallica has four band members. Then it searches each band member to request the total of children and add up the total. Finally, it returns the total number of children as the final answer.

ReAct prompting in practice requires understanding that you continually have to resend the previous prompts/responses (and do trimming of the extra generated content) as well as set up the model with appropriate examples/instructions. Please refer to the notebook¹⁴ hosted in the GoogleCloudPlatform Github repository, which goes into a bit more detail showing the actual LLM inputs and outputs with a more elaborate example.

Automatic Prompt Engineering

At this point you might realize that writing a prompt can be complex. Wouldn't it be nice to automate this (write a prompt to write prompts)? Well, there's a method: Automatic Prompt Engineering (APE). This method¹⁵ not only alleviates the need for human input but also enhances the model's performance in various tasks.

You will prompt a model to generate more prompts. Evaluate them, possibly alter the good ones. And repeat.

For example, you can use automatic prompt engineering that will help with training a chatbot for a merchandise t-shirt webshop. We want to figure out all the various ways customers could phrase their order for buying a band merchandise t-shirt.

1. Write the prompt which will generate the output variants. In this example, I am using gemini-pro to generate 10 instructions. See Table 15:

| | |
|---------------|---|
| Prompt | We have a band merchandise t-shirt webshop, and to train a chatbot we need various ways to order: "One Metallica t-shirt size S". Generate 10 variants, with the same semantics but keep the same meaning. |
| Output | <ol style="list-style-type: none"> 1. I'd like to purchase a Metallica t-shirt in size small. 2. Can I order a small-sized Metallica t-shirt? 3. I'd like to buy one Metallica t-shirt in small. 4. One Metallica shirt, size small, please. 5. I want one Metallica t-shirt in size small. 6. I'd like to place an order for one Metallica t-shirt in small. 7. Small Metallica t-shirt, one please. 8. I'm looking for one Metallica t-shirt in small. 9. Can you get me one Metallica t-shirt in small? 10. One Metallica small t-shirt, please. |

Table 15. Use an LLM to generate prompts

2. Evaluate all of the instruction candidates by scoring the candidates based on a chosen metric, For example, you can use. BLEU (Bilingual Evaluation Understudy) or ROUGE (Recall-Oriented Understudy for Gisting Evaluation).
3. Select the instruction candidate with the highest evaluation score. This candidate will be the final prompt you can use in your software application or chatbot. You can also tweak the select prompt and evaluate again.

Code prompting

Gemini focuses primarily on text-based prompts, which also includes writing prompts for returning code. Let's go to the Vertex AI Studio and test these prompts to look at some coding examples.

Prompts for writing code

Gemini can also be a developer and help you with writing code in any programming language of choice. As a developer this can help you speed up the process of writing code.

Imagine a folder on your machine with hundreds of files that needs to be renamed. Renaming each file would take you a lot of time. You know a little Bash, and could write a script to automate this, but that may also take awhile. So let's write a prompt. You can write the prompt in the public consumer chatbot Gemini, or if you are more concerned about confidentiality, you can write these prompts within your Google Cloud account and open Vertex AI Studio. The advantage of Vertex AI Studio is that you can configure the temperature etc.

| | | | |
|--------------------|--|--------------------|------|
| Name | 1_bash_rename_files_in_folder | | |
| Goal | Write a prompt to write code in Bash to rename files in a folder. | | |
| Model | gemini-pro | | |
| Temperature | 0.1 | Token Limit | 1024 |
| Top-K | N/A | Top-P | 1 |
| Prompt | Write a code snippet in Bash, which asks for a folder name. Then it takes the contents of the folder and renames all the files inside by prepending the name draft to the file name. | | |
| Output | <pre>```bash #!/bin/bash # Ask for the folder name echo "Enter the folder name: " read folder_name # Check if the folder exists if [! -d "\$folder_name"]; then echo "Folder does not exist." exit 1 fi # Get the list of files in the folder files=("\$folder_name"/*) # Rename each file by prepending "draft" to the file name for file in "\${files[@]}"; do new_file_name="draft_\$(basename "\$file")" mv "\$file" "\$new_file_name" done echo "Files renamed successfully." ``` </pre> | | |

Table 16. Using Gemini to program code

That seems like good code to me - it's even documented! However, since LLMs can't reason, and repeat training data, it's essential to read and test your code first.

The moment we are all waiting for, does it really work?

Let's try it first with a test folder with only a few files in it, which will need to be renamed from `filename.txt` to `draft_filename.txt`.

1. Copy the output from Table 16 (without the ````bash```` text wrapper), and paste it in a new file called: "`rename_files.sh`".
2. Open a terminal window and type: `. rename_files.sh`. It will ask to enter a folder name, e.g. `test`. and hit enter.
3. The script seems to run fine. You will see the message: **Files renamed successfully**. When you look into the test folder, notice that all the files were perfectly renamed to `draft_filename.txt`.

It worked!

Prompts for explaining code

As a developer when you work in teams you have to read someone else's code. Gemini can help you with this as well. Let's take the code output from Table 16, remove the comments and ask the large language model to explain what's happening, See Table 17:

| | | | |
|-------------|--|-------------|------|
| Name | 1_bash_explain_the_code | | |
| Goal | Write a prompt to explain Bash code | | |
| Model | gemini-pro | | |
| Temperature | 0.1 | Token Limit | 1024 |
| Top-K | N/A | Top-P | 1 |
| Prompt | <p>Explain to me the below Bash code:</p> <pre>``` #!/bin/bash echo "Enter the folder name: " read folder_name if [! -d "\$folder_name"]; then echo "Folder does not exist." exit 1 fi files=("\$folder_name"/*) for file in "\${files[@]}"; do new_file_name="draft_\$(basename "\$file")" mv "\$file" "\$new_file_name" done echo "Files renamed successfully." ``` </pre> | | |
| Output | <p>This Bash script performs the following tasks:</p> <ol style="list-style-type: none"> User Input:
- It prompts the user to enter a folder name using the `echo` and `read` commands. The entered folder name is stored in the `folder_name` variable. Folder Existence Check:
- It checks if the folder with the specified name exists using the `[! -d "\$folder_name"]` condition. If the folder does not exist, it displays an error message ("Folder does not exist.") and exits the script with an exit code of 1. | | |

Continues next page...

| | |
|---------------|--|
| Output | <p>3. File Listing:</p> <ul style="list-style-type: none"> - It lists all the files inside the specified folder using the `files=("\$folder_name"/*)` command. This creates an array called `files` containing the paths to all the files in the folder. <p>4. File Renaming:</p> <ul style="list-style-type: none"> - It iterates through each file in the `files` array using a `for` loop. - For each file, it generates a new file name using the `new_file_name="draft_\${basename "\$file"}"` command. This new file name is created by prefixing the original file name with the string "draft_". - It then renames the file using the `mv "\$file" "\$new_file_name"` command, which moves the file to the new file name. <p>5. Success Message:</p> <ul style="list-style-type: none"> - After renaming all the files, it displays a success message ("Files renamed successfully.") to indicate that the renaming operation was successful |
|---------------|--|

Table 17. A prompt for explaining code

Prompts for translating code

The bash code from Table 16 seems to work fine. However this script could really be reusable when it would prompt me for file names, ideally it should work as a separate application with an UI. As a starting point, Python would be a better language for a (web) application than Bash. LLMs can help with translating code from one language to another. See the example in Table 18:

| | | | |
|------------------------|---|--------------------|------|
| Name | 1_python_rename_files | | |
| Goal | Write a prompt to translate Bash code to Python | | |
| Model | gemini-pro | | |
| Temperature | 0.1 | Token Limit | 1024 |
| Top-K | N/A | Top-P | 1 |
| Prompt | <p>Translate the below Bash code to a Python snippet.</p> <pre>```bash #!/bin/bash echo "Enter the folder name: " read folder_name if [! -d "\$folder_name"]; then echo "Folder does not exist." exit 1 fi files=("\$folder_name"/*) for file in "\${files[@]}"; do new_file_name="draft_\$(basename "\$file")" mv "\$file" "\$new_file_name" done echo "Files renamed successfully." ``` </pre> | | |
| Output | <pre>```python import os import shutil # Get the folder name from the user folder_name = input("Enter the folder name: ") # Check if the folder exists if not os.path.isdir(folder_name): print("Folder does not exist.") exit(1) ``` </pre> | | |
| Continues next page... | | | |

| | |
|--------|--|
| Output | <pre># Get a list of all files in the folder files = os.listdir(folder_name) # Iterate over the files for file in files: # Create a new file name with the prefix "draft_" new_file_name = f"draft_{file}" # Move the file to the new name shutil.move(os.path.join(folder_name, file), os.path.join(folder_name, new_file_name)) # Print a success message print("Files renamed successfully.") ``</pre> |
|--------|--|

Table 18. A prompt for translating code from Bash to Python

Read and review the code. Copy the output from the prompt and paste it into a new file: **file_renamer.py**. Test the code by opening a Terminal window, and execute the following command **python file_renamer.py**.

NOTE: When prompting for (Python) code in the Language Studio in Vertex AI, you will have to click on the ‘Markdown’ button. Otherwise you will receive plain text which is missing the proper indenting of lines, which is important for running Python code.

Prompts for debugging and reviewing code

Let’s manually write some edits to the code of Table 18. It should prompt the user for the filename prefix, and write this prefix in upper case characters. See the example code in Snippet *3, but what a bummer. It now returns Python errors!

Python

```
import os
import shutil

folder_name = input("Enter the folder name: ")
prefix = input("Enter the string to prepend to the filename: ")
text = toUpperCase(prefix)

if not os.path.isdir(folder_name):
    print("Folder does not exist.")
    exit(1)

files = os.listdir(folder_name)

for file in files:
    new_filename = f"{text}_{file}"

shutil.move(os.path.join(folder_name, file),
os.path.join(folder_name, new_file_name))

print("Files renamed successfully.")
```

Snippet 3. A broken Python script

Doh! That looks like a bug:

```
The below Python code gives an error:

Traceback (most recent call last):
  File "/Users/leeboonstra/Documents/test_folder/rename_files.py", line 7,
in <module>
    text = toUpperCase(prefix)
NameError: name 'toUpperCase' is not defined
```

Snippet 4. I broke the Python code

Let's see if we can ask the large language model to debug and review code. Have a look into Table 19:

| | | | |
|--------------------|--|--------------------|------|
| Name | 1_python_debug_code | | |
| Goal | Write a prompt to debug and review Python code. | | |
| Model | gemini-pro | | |
| Temperature | 0.1 | Token Limit | 1024 |
| Top-K | N/A | Top-P | 1 |
| Prompt | <p>The below Python code gives an error:</p> <p>Traceback (most recent call last):</p> <pre>File "/Users/leebloonstra/Documents/test_folder/rename_files.py", line 7, in <module> text = toUpperCase(prefix) NameError: name 'toUpperCase' is not defined</pre> <p>Debug what's wrong and explain how I can improve the code.</p> <pre>```python import os import shutil # Get the folder name from the user folder_name = input("Enter the folder name: ") prefix = input("Enter the string to prepend to the filename: ") text = toUpperCase(prefix) # Check if the folder exists if not os.path.isdir(folder_name): print("Folder does not exist.") exit(1) # Get a list of all files in the folder files = os.listdir(folder_name) # Iterate over the files for file in files:</pre> | | |
| | Continues next page... | | |

| | |
|---------------|---|
| Prompt | <pre># Create a new file name with the prefix "draft_" new_filename = f"{text}_{file}" # Move the file to the new name shutil.move(os.path.join(folder_name, file), os.path.join(folder_name, new_file_name)) # Print a success message print("Files renamed successfully.") ``` </pre> |
| Output | <p>The code calls the `toUpperCase` function to convert `prefix` to uppercase, but that function is not defined. To fix this issue, you can use the `upper()` method of the string class which converts a given string into uppercase. The modified code is shown below:</p> <pre>```python import os import shutil # Get the folder name from the user folder_name = input("Enter the folder name: ") prefix = input("Enter the string to prepend to the filename: ") text = prefix.upper() # Check if the folder exists if not os.path.isdir(folder_name): print("Folder does not exist.") exit(1) # Get a list of all files in the folder files = os.listdir(folder_name) # Iterate over the files for file in files: # Create a new file name with the prefix "draft_" new_filename = f"{text}_{file}" # Move the file to the new name shutil.move(os.path.join(folder_name, file), os.path.join(folder_name, new_filename)) ``` </pre> <p style="text-align: right;">Continues next page...</p> |

| | |
|--|-------------------------------|
| <p>Output</p> <pre>os.path.join(folder_name, new_filename)) # Print a success message print("Files renamed successfully.") ``` </pre> <p>Additionally, there are a few other improvements that can be made to the code:</p> <ol style="list-style-type: none"> 1. The file extension of the new filenames is not kept. It's better to extract the extension from the original filename and append it to the new filename. 2. The code assumes that there are no spaces in the folder name. However, it is a good practice to handle spaces gracefully by replacing them with underscores or hyphens. 3. Using the `f` string syntax for string interpolation is more readable and concise than the traditional `+` operator. 4. The code doesn't handle errors that might occur during the renaming process. It would be better to wrap the `shutil.move` call in a `try...except` block to catch any potential errors. <p>Here is the improved code with these suggestions:</p> <pre>```python import os import shutil # Get the folder name from the user folder_name = input("Enter the folder name: ") prefix = input("Enter the string to prepend to the filename: ") # Check if the folder exists if not os.path.isdir(folder_name): print("Folder does not exist.") exit(1) # Get a list of all files in the folder files = os.listdir(folder_name) # Iterate over the files for file in files: new_filename = os.path.join(folder_name, prefix + os.path.splitext(file)[1]) shutil.move(os.path.join(folder_name, file), new_filename) print(f"Renamed {file} to {new_filename}") ``` </pre> | <p>Continues next page...</p> |
|--|-------------------------------|

| | |
|---------------|--|
| Output | <pre> # Get the file extension ext = os.path.splitext(file)[1] # Create a new file name with the prefix and the file extension new_filename = f"{prefix}_{os.path.splitext(file)[0]}.{ext}" # Build the full paths for the original and new files old_path = os.path.join(folder_name, file) new_path = os.path.join(folder_name, new_filename) # Try to move the file try: shutil.move(old_path, new_path) except Exception as e: print(f"An error occurred while moving the file {file}: {e}") # Print a success message print("Files renamed successfully.") ``` <p>These improvements make the code more robust and flexible while also making it easier to read and understand... (The response was truncated because it has reached the token limit. Try to increase the token limit if you need a longer response.)</p> </pre> |
|---------------|--|

Table 19. A prompt for debugging and reviewing Python code

That's great. Not only did it tell me how to solve the problem, it also figured out that my code had more bugs and how to solve them, too. The last part of the prompt gave suggestions to improve the code in general.

What about multimodal prompting?

Prompting for code still uses the same regular large language model. Multimodal prompting is a separate concern, it refers to a technique where you use multiple input formats to guide a large language model, instead of just relying on text. This can include combinations of text, images, audio, code, or even other formats, depending on the model's capabilities and the task at hand.

Best Practices

Finding the right prompt requires tinkering. Language Studio in Vertex AI is a perfect place to play around with your prompts, with the ability to test against the various models.

Use the following best practices to become a pro in prompt engineering.

Provide examples

The most important best practice is to provide (one shot / few shot) examples within a prompt. This is highly effective because it acts as a powerful teaching tool. These examples showcase desired outputs or similar responses, allowing the model to learn from them and tailor its own generation accordingly. It's like giving the model a reference point or target to aim for, improving the accuracy, style, and tone of its response to better match your expectations.

Design with simplicity

Prompts should be concise, clear, and easy to understand for both you and the model. As a rule of thumb, if it's already confusing for you it will likely be also confusing for the model. Try not to use complex language and don't provide unnecessary information.

Examples:

BEFORE:

I am visiting New York right now, and I'd like to hear more about great locations. I am with two 3 year old kids. Where should we go during our vacation?

AFTER REWRITE:

Act as a travel guide for tourists. Describe great places to visit in New York Manhattan with a 3 year old.

Try using verbs that describe the action. Here's a set of examples:

Act, Analyze, Categorize, Classify, Contrast, Compare, Create, Describe, Define, Evaluate, Extract, Find, Generate, Identify, List, Measure, Organize, Parse, Pick, Predict, Provide, Rank, Recommend, Return, Retrieve, Rewrite, Select, Show, Sort, Summarize, Translate, Write.

Be specific about the output

Be specific about the desired output. A concise instruction might not guide the LLM enough or could be too generic. Providing specific details in the prompt (through system or context prompting) can help the model to focus on what's relevant, improving the overall accuracy.

Examples:

DO:

Generate a 3 paragraph blog post about the top 5 video game consoles.
The blog post should be informative and engaging, and it should be written in a conversational style.

DO NOT:

Generate a blog post about video game consoles.

Use Instructions over Constraints

Instructions and constraints are used in prompting to guide the output of a LLM.

- An **instruction** provides explicit instructions on the desired format, style, or content of the response. It guides the model on what the model should do or produce.
- A **constraint** is a set of limitations or boundaries on the response. It limits what the model should not do or avoid.

Growing research suggests that focusing on positive instructions in prompting can be more effective than relying heavily on constraints. This approach aligns with how humans prefer positive instructions over lists of what not to do.

Instructions directly communicate the desired outcome, whereas constraints might leave the model guessing about what is allowed. It gives flexibility and encourages creativity within the defined boundaries, while constraints can limit the model's potential. Also a list of constraints can clash with each other.

Constraints are still valuable but in certain situations. To prevent the model from generating harmful or biased content or when a strict output format or style is needed.

If possible, use positive instructions: instead of telling the model what not to do, tell it what to do instead. This can avoid confusion and improve the accuracy of the output.

DO:

Generate a 1 paragraph blog post about the top 5 video game consoles.
Only discuss the console, the company who made it, the year, and total sales.

DO NOT:

Generate a 1 paragraph blog post about the top 5 video game consoles.
Do not list video game names.

As a best practice, start by prioritizing instructions, clearly stating what you want the model to do and only use constraints when necessary for safety, clarity or specific requirements. Experiment and iterate to test different combinations of instructions and constraints to find what works best for your specific tasks, and document these.

Control the max token length

To control the length of a generated LLM response, you can either set a max token limit in the configuration or explicitly request a specific length in your prompt. For example:

"Explain quantum physics in a tweet length message."

Use variables in prompts

To reuse prompts and make it more dynamic use variables in the prompt, which can be changed for different inputs. E.g. as shown in Table 20, a prompt which gives facts about a city. Instead of hardcoding the city name in the prompt, use a variable. Variables can save you time and effort by allowing you to avoid repeating yourself. If you need to use the same piece of information in multiple prompts, you can store it in a variable and then reference that variable in each prompt. This makes a lot of sense when integrating prompts into your own applications.

| | |
|--------|--|
| Prompt | VARIABLES
{city} = "Amsterdam"

PROMPT
You are a travel guide. Tell me a fact about the city: {city} |
| Output | Amsterdam is a beautiful city full of canals, bridges, and narrow streets. It's a great place to visit for its rich history, culture, and nightlife. |

Table 20. Using variables in prompts

Experiment with input formats and writing styles

Different models, model configurations, prompt formats, word choices, and submits can yield different results. Therefore, it's important to experiment with prompt attributes like the style, the word choice, and the type prompt (zero shot, few shot, system prompt).

For example a prompt with the goal to generate text about the revolutionary video game console Sega Dreamcast, can be formulated as a **question**, a **statement** or an **instruction**, resulting in different outputs:

- **Question:** What was the Sega Dreamcast and why was it such a revolutionary console?
- **Statement:** The Sega Dreamcast was a sixth-generation video game console released by Sega in 1999. It...
- **Instruction:** Write a single paragraph that describes the Sega Dreamcast console and explains why it was so revolutionary.

For few-shot prompting with classification tasks, mix up the classes

Generally speaking, the order of your few-shots examples should not matter much. However, when doing classification tasks, make sure you mix up the possible response classes in the few shot examples. This is because you might otherwise be overfitting to the specific order of the examples. By mixing up the possible response classes, you can ensure that the model is learning to identify the key features of each class, rather than simply memorizing the order of the examples. This will lead to more robust and generalizable performance on unseen data.

A good rule of thumb is to start with 6 few shot examples and start testing the accuracy from there.

Adapt to model updates

It's important for you to stay on top of model architecture changes, added data, and capabilities. Try out newer model versions and adjust your prompts to better leverage new model features. Tools like Vertex AI Studio are great to store, test, and document the various versions of your prompt.

Experiment with output formats

Besides the prompt input format, consider experimenting with the output format. For non-creative tasks like extracting, selecting, parsing, ordering, ranking, or categorizing data try having your output returned in a structured format like JSON or XML.

There are some benefits in returning JSON objects from a prompt that extracts data. In a real-world application I don't need to manually create this JSON format, I can already return the data in a sorted order (very handy when working with datetime objects), but most importantly, by prompting for a JSON format it forces the model to create a structure and limit hallucinations.

In summary, benefits of using JSON for your output:

- Returns always in the same style
- Focus on the data you want to receive

- Less chance for hallucinations
- Make it relationship aware
- You get data types
- You can sort it

Table 4 in the few-shot prompting section shows an example on how to return structured output.

JSON Repair

While returning data in JSON format offers numerous advantages, it's not without its drawbacks. The structured nature of JSON, while beneficial for parsing and use in applications, requires significantly more tokens than plain text, leading to increased processing time and higher costs. Furthermore, JSON's verbosity can easily consume the entire output window, becoming especially problematic when the generation is abruptly cut off due to token limits. This truncation often results in invalid JSON, missing crucial closing braces or brackets, rendering the output unusable. Fortunately, tools like the [json-repair](#) library (available on PyPI) can be invaluable in these situations. This library intelligently attempts to automatically fix incomplete or malformed JSON objects, making it a crucial ally when working with LLM-generated JSON, especially when dealing with potential truncation issues.

Working with Schemas

Using structured JSON as an output is a great solution, as we've seen multiple times in this paper. But what about *input*? While JSON is excellent for structuring the *output* the LLM generates, it can also be incredibly useful for structuring the *input* you provide. This is where JSON Schemas come into play. A JSON Schema defines the expected structure and data types of your JSON input. By providing a schema, you give the LLM a clear blueprint of the data it should expect, helping it focus its *attention* on the relevant information and reducing the risk of misinterpreting the input. Furthermore, schemas can help establish relationships between different pieces of data and even make the LLM "time-aware" by including date or timestamp fields with specific formats.

Here's a simple example:

Let's say you want to use an LLM to generate descriptions for products in an e-commerce catalog. Instead of just providing a free-form text description of the product, you can use a JSON schema to define the product's attributes:

```
{  
  "type": "object",  
  "properties": {  
    "name": { "type": "string", "description": "Product name" },  
    "category": { "type": "string", "description": "Product category" },  
    "price": { "type": "number", "format": "float", "description": "Product price" },  
    "features": {  
      "type": "array",  
      "items": { "type": "string" },  
      "description": "Key features of the product"  
    },  
    "release_date": { "type": "string", "format": "date", "description":  
      "Date the product was released"}  
  },
```

Snippet 5. Definition of the structured output schema

Then, you can provide the actual product data as a JSON object that conforms to this schema:

```
{  
  "name": "Wireless Headphones",  
  "category": "Electronics",  
  "price": 99.99,  
  "features": ["Noise cancellation", "Bluetooth 5.0", "20-hour battery life"],  
  "release_date": "2023-10-27"  
}
```

Snippet 6. Structured output from the LLM

By preprocessing your data and instead of providing full documents only providing both the schema and the data, you give the LLM a clear understanding of the product's attributes, including its release date, making it much more likely to generate an accurate and relevant description. This structured input approach, guiding the LLM's attention to the relevant fields, is especially valuable when working with large volumes of data or when integrating LLMs into complex applications.

Experiment together with other prompt engineers

If you are in a situation where you have to try to come up with a good prompt, you might want to find multiple people to make an attempt. When everyone follows the best practices (as listed in this chapter) you are going to see a variance in performance between all the different prompt attempts.

CoT Best practices

For CoT prompting, putting the answer after the reasoning is required because the generation of the reasoning changes the tokens that the model gets when it predicts the final answer.

With CoT and self-consistency you need to be able to extract the final answer from your prompt, separated from the reasoning.

For CoT prompting, set the temperature to 0.

Chain of thought prompting is based on greedy decoding, predicting the next word in a sequence based on the highest probability assigned by the language model. Generally speaking, when using reasoning, to come up with the final answer, there's likely one single correct answer. Therefore the temperature should always be set to 0.

Document the various prompt attempts

The last tip was mentioned before in this chapter, but we can't stress enough how important it is: document your prompt attempts in full detail so you can learn over time what went well and what did not.

Prompt outputs can differ across models, across sampling settings, and even across different versions of the same model. Moreover, even across identical prompts to the same model, small differences in output sentence formatting and word choice can occur. (For example, as mentioned previously, if two tokens have the same predicted probability, ties may be broken randomly. This can then impact subsequent predicted tokens.).

We recommend creating a Google Sheet with Table 21 as a template. The advantages of this approach are that you have a complete record when you inevitably have to revisit your prompting work—either to pick it up in the future (you’d be surprised how much you can forget after just a short break), to test prompt performance on different versions of a model, and to help debug future errors.

Beyond the fields in this table, it’s also helpful to track the version of the prompt (iteration), a field to capture if the result was OK/NOT OK/SOMETIMES OK, and a field to capture feedback. If you’re lucky enough to be using Vertex AI Studio, save your prompts (using the same name and version as listed in your documentation) and track the hyperlink to the saved prompt in the table. This way, you’re always one click away from re-running your prompts.

When working on a *retrieval augmented generation* system, you should also capture the specific aspects of the RAG system that impact what content was inserted into the prompt, including the query, chunk settings, chunk output, and other information.

Once you feel the prompt is close to perfect, take it to your project codebase. And in the codebase, save prompts in a separate file from code, so it’s easier to maintain. Finally, ideally your prompts are part of an operationalized system, and as a prompt engineer you should rely on automated tests and evaluation procedures to understand how well your prompt generalizes to a task.

Prompt engineering is an iterative process. Craft and test different prompts, analyze, and document the results. Refine your prompt based on the model’s performance. Keep experimenting until you achieve the desired output. When you change a model or model configuration, go back and keep experimenting with the previously used prompts.

| | | | |
|--------------------|--|--------------------|----------|
| Name | [name and version of your prompt] | | |
| Goal | [One sentence explanation of the goal of this attempt] | | |
| Model | [name and version of the used model] | | |
| Temperature | [value between 0 - 1] | Token Limit | [number] |
| Top-K | [number] | Top-P | [number] |
| Prompt | [Write all the full prompt] | | |
| Output | [Write out the output or multiple outputs] | | |

Table 21. A template for documenting prompts

Summary

This whitepaper discusses prompt engineering. We learned various prompting techniques, such as:

- Zero prompting
- Few shot prompting
- System prompting
- Role prompting
- Contextual prompting
- Step-back prompting
- Chain of thought
- Self consistency
- Tree of thoughts

- ReAct

We even looked into ways how you can automate your prompts.

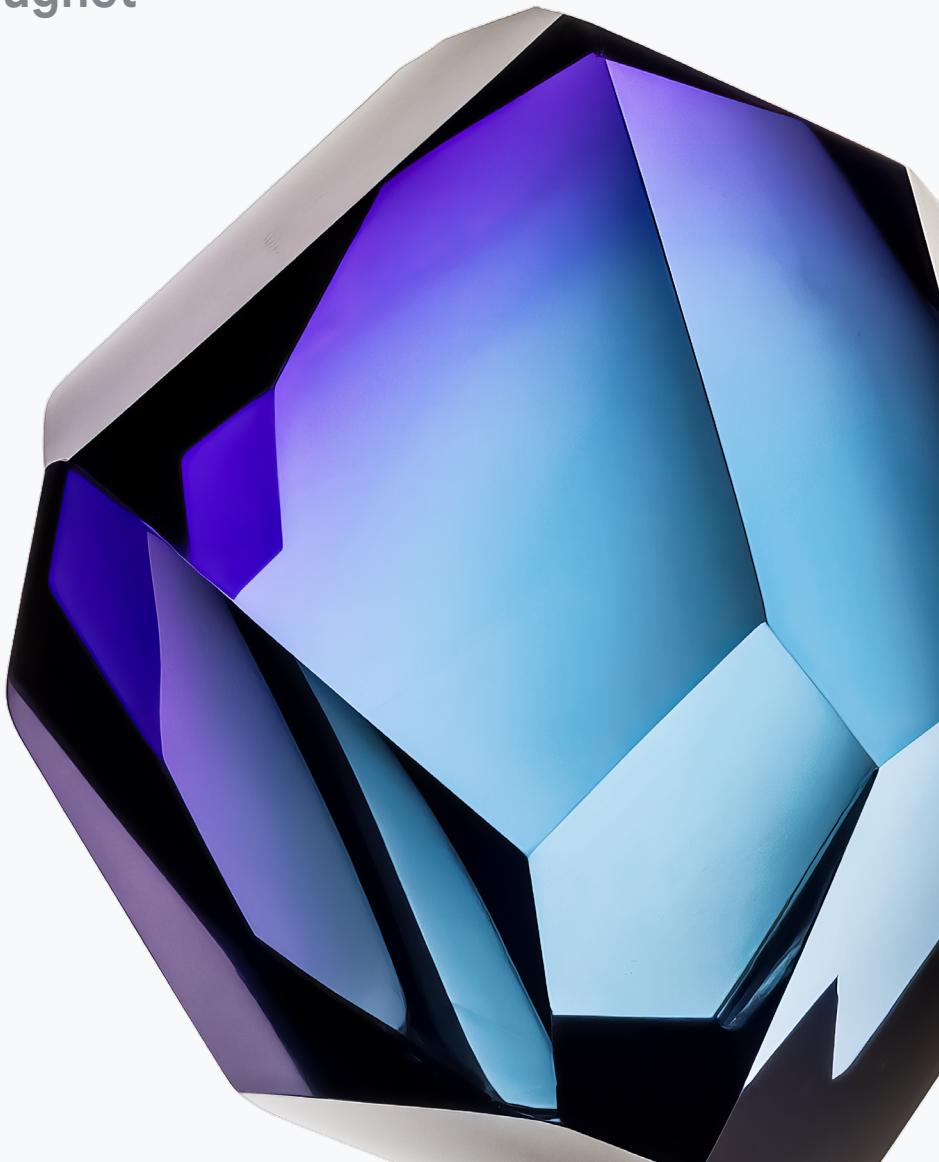
The whitepaper then discusses the challenges of gen AI like the problems that can happen when your prompts are insufficient. We closed with best practices on how to become a better prompt engineer.

Endnotes

1. Google, 2023, Gemini by Google. Available at: <https://gemini.google.com>.
2. Google, 2024, Gemini for Google Workspace Prompt Guide. Available at: <https://inthecloud.withgoogle.com/gemini-for-google-workspace-prompt-guide/dl-cd.html>.
3. Google Cloud, 2023, Introduction to Prompting. Available at: <https://cloud.google.com/vertex-ai/generative-ai/docs/learn/prompts/introduction-prompt-design>.
4. Google Cloud, 2023, Text Model Request Body: Top-P & top-K sampling methods. Available at: https://cloud.google.com/vertex-ai/docs/generative-ai/model-reference/text#request_body.
5. Wei, J., et al., 2023, Zero Shot - Fine Tuned language models are zero shot learners. Available at: <https://arxiv.org/pdf/2109.01652.pdf>.
6. Google Cloud, 2023, Google Cloud Model Garden. Available at: <https://cloud.google.com/model-garden>.
7. Brown, T., et al., 2023, Few Shot - Language Models are Few Shot learners. Available at: <https://arxiv.org/pdf/2005.14165.pdf>.
8. Zheng, L., et al., 2023, Take a Step Back: Evoking Reasoning via Abstraction in Large Language Models. Available at: <https://openreview.net/pdf?id=3bq3jsvcQ1>
9. Wei, J., et al., 2023, Chain of Thought Prompting. Available at: <https://arxiv.org/pdf/2201.11903.pdf>.
10. Google Cloud Platform, 2023, Chain of Thought and React. Available at: https://github.com/GoogleCloudPlatform/generative-ai/blob/main/language/prompts/examples/chain_of_thought_react.ipynb.
11. Wang, X., et al., 2023, Self Consistency Improves Chain of Thought reasoning in language models. Available at: <https://arxiv.org/pdf/2203.11171.pdf>.
12. Yao, S., et al., 2023, Tree of Thoughts: Deliberate Problem Solving with Large Language Models. Available at: <https://arxiv.org/pdf/2305.10601.pdf>.
13. Yao, S., et al., 2023, ReAct: Synergizing Reasoning and Acting in Language Models. Available at: <https://arxiv.org/pdf/2210.03629.pdf>.
14. Google Cloud Platform, 2023, Advance Prompting: Chain of Thought and React. Available at: https://github.com/GoogleCloudPlatform/applied-ai-engineering-samples/blob/main/genai-on-vertex-ai/advanced_prompting_training/cot_react.ipynb.
15. Zhou, C., et al., 2023, Automatic Prompt Engineering - Large Language Models are Human-Level Prompt Engineers. Available at: <https://arxiv.org/pdf/2211.01910.pdf>.

Embeddings & Vector Stores

Authors: Anant Nawalgaria,
Xiaoqi Ren, and Charles Sugnet



Google

Acknowledgements

Content contributors

Antonio Gulli

Grace Mollison

Ruiqi Guo

Iftekhar Naim

Jinhyuk Lee

Alan Li

Patricia Florissi

Andrew Brook

Omid Fatemieh

Zhuyun Dai

Lee Boonstra

Per Jacobsson

Siddhartha Reddy Jonnalagadda

Xi Cheng

Raphael Hoffmann

Curators and Editors

Antonio Gulli

Anant Nawalgaria

Grace Mollison

Technical Writer

Joey Haymaker

Designer

Michael Lanning



Table of contents

| | |
|---|----|
| Introduction | 5 |
| Why embeddings are important | 6 |
| Evaluating Embedding Quality | 9 |
| Search Example | 11 |
| Types of embeddings | 16 |
| Text embeddings | 16 |
| Word embeddings | 19 |
| Document embeddings | 23 |
| Shallow BoW models | 24 |
| Deeper pretrained large language models | 26 |
| Image & multimodal embeddings | 30 |
| Structured data embeddings | 32 |
| General structured data | 32 |
| User/item structured data | 33 |

| | |
|---|-----------|
| Graph embeddings | 33 |
| Training Embeddings | 34 |
| Vector search | 36 |
| Important vector search algorithms | 37 |
| Locality sensitive hashing & trees | 38 |
| Hierarchical navigable small worlds | 41 |
| ScaNN | 44 |
| Vector databases | 47 |
| Operational considerations | 49 |
| Applications | 51 |
| Q & A with sources (retrieval augmented generation) | 52 |
| Summary | 57 |
| Endnotes | 59 |

These low-dimensional numerical representations of real-world data significantly helps efficient large-scale data processing and storage by acting as means of lossy compression of the original data.

Introduction

Modern machine learning thrives on diverse data—images, text, audio, and more. This whitepaper explores the power of embeddings, which transform this heterogeneous data into a unified vector representation for seamless use in various applications.

We'll guide you through:

- **Understanding Embeddings:** Why they are essential for handling multimodal data and their diverse applications.
- **Embedding Techniques:** Methods for mapping different data types into a common vector space.

- **Efficient Management:** Techniques for storing, retrieving, and searching vast collections of embeddings.
- **Vector Databases:** Specialized systems for managing and querying embeddings, including practical considerations for production deployment.
- **Real-World Applications:** Concrete examples of how embeddings and vector databases are combined with large language models (LLMs) to solve real-world problems.

Throughout the whitepaper, code snippets provide hands-on illustrations of key concepts.

Why embeddings are important

In essence, embeddings are numerical representations of real-world data such as text, speech, image, or videos. The name embeddings refers to a similar concept in mathematics where one space can be mapped, or embedded, into another space. For example, the original BERT Model [ref] embeds text into a vector of 768 numbers, thus mapping from the very high dimensional space of all sentences to a much smaller 768 dimensions. Embeddings are expressed as low-dimensional vectors where the geometric distance between two vectors in the vector space is a projection of the relationship and semantic similarity between the two real-world objects that the vectors represent. In other words, they help you with providing compact representations of data of different types, while simultaneously also allowing you to compare two different data objects and tell how similar or different they are on a numerical scale. For example: the word ‘computer’ has a similar meaning to the picture of a computer, as well as to the word ‘laptop’ but not to the word ‘car’. These low-dimensional numerical representations of real-world data significantly help efficient large-scale data processing and storage by acting as means of lossy compression of the original data while retaining its important semantic properties.

For some intuition about embeddings consider the familiar latitude and longitude which are used to map locations on earth to a pair of numbers, or a vector of length two. Latitude and longitude can be thought of as an embedding of a particular location. While seemingly obvious now, this simple mapping of a location to a pair of numbers transformed human navigation and is still critical to this day. Given the latitude and longitude of two addresses it is relatively easy to see how distant they are from each other, or look up other nearby locations. As with latitude and longitude if two text embeddings are close to each other in the embeddings space they will be semantically similar in their text meaning. Also, it is possible to find new semantically similar text phrases by looking nearby in that vector space. This ability to find similar items in very large data sets with very low latency using vector databases is critical for many production use cases today including search, recommendations, advertising, fraud detection and many more. Note that while the latitude and longitude embedding model was designed based on the spherical shape of the earth, the embedding space for text is learned by the neural network model. Importantly, the embeddings learned by different models will not be comparable to each other and it is critical to make sure in practice that compatible and consistent versions of embeddings are being used.

Key applications for embeddings are retrieval and recommendations, where the results are usually selected from a massive search space. For example, Google Search is a retrieval task over the search space of the entire internet. Today's retrieval and recommendation systems' success depends on the following steps:

1. Precomputing the embeddings for billions of items in the search space.
2. Mapping query embeddings into the same embedding space.
3. Efficient computing and retrieving of the items whose embeddings are the nearest neighbors of the query embeddings in the search space.

Embeddings also shine in the world of multimodality. Many applications work with large amounts of data of various modalities: text, speech, image, and videos to name a few. Joint embeddings are when multiple types of objects are being mapped into the same embeddings space, for example retrieving videos based on text queries. These embedding representations are designed to capture as much of the original object's characteristics as possible.

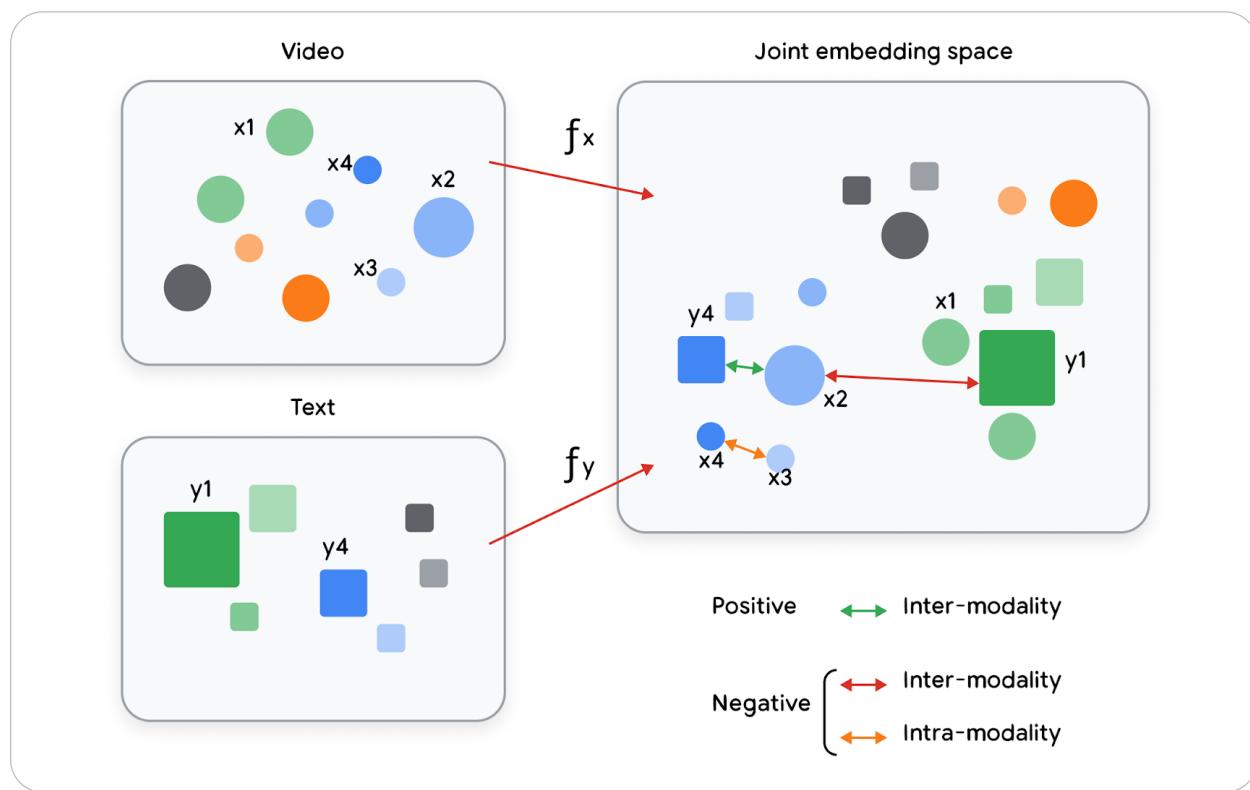


Figure 1. Projecting objects/content into a joint vector space with semantic meaning

Embeddings are designed so they place objects with similar semantic properties closer in the embedding space (a low-dimensional vector space where items can be projected). The embeddings can then be used as a condensed, meaningful input in downstream applications. For example, you can use them as features for ML models, recommender systems, search

engines, and many more. So your data not only gets a compact numerical representation, but this representation also preserves the semantic meanings for a specific task or across a variety of tasks. The fact that these representations are task-specific means you can generate different embeddings for the same object, optimized for the task at hand.

Evaluating Embedding Quality

Embedding models are evaluated differently depending on the task. Many of the common metrics for evaluating quality focus on the ability to retrieve similar items, while excluding items that are not similar. This type of evaluation requires a labeled datasets for which the relevant, or correct, documents are already known as seen in Snippet 0 where the NFCorpus dataset is used to illustrate different metrics. For the search use case described above two important metrics for evaluating quality are: 1) *precision* - all documents retrieved should be relevant and 2) *recall* - all of the relevant documents should be retrieved. Intuitively, the optimal embedding model would retrieve all of the relevant documents and no documents that weren't relevant, however it is often the case that some relevant documents are excluded and some irrelevant ones get retrieved so more quantitative definitions are required for evaluating quality over large sets of documents and embedding models. Precision is quantified by dividing the number of relevant documents by the total number of retrieved documents. It is often quoted for a particular number of retrieved documents. For example if ten documents were retrieved for an embedding and seven of them were relevant and other three were not, the precision@10 would be $7/10 = 0.7$. Recall looks at how many of the relevant documents were retrieved and is calculated by dividing the number of relevant documents retrieved by the total number of relevant documents in the corpus. Recall is also often quoted for a particular number of documents retrieved. For example, if 20 documents were retrieved and three of them were relevant, but there were six total relevant documents in the corpus the recall@20 would be $3/6 = 0.5$.

Precision and recall are very useful when relevancy scores are binary, but don't capture the case when some documents are more relevant than others. For example, when using a search engine it is highly desirable that the most relevant result is at the top of the results list as end users are sensitive to the ordering of those results, even if they are all relevant. When the detailed ordering of document relevancy is known for a data set, metrics like the Normalized Discounted Cumulative Gain (nDCG) can measure the quality of the ranking produced by the embedding model compared to the desired ranking. The formula at

$$\text{position } p \text{ for DCG} = \sum_{i=1}^p \frac{\text{rel}_i}{\log_2(i+1)} \text{ where rel}_i \text{ is a relevancy score. The denominator}$$

penalizes documents for being lower on the list, and DCG maximizes the score when most relevant documents are at the top of the list. The normalized version is calculated by dividing the DCG score by the ideal ordering score and ranges from 0.0 to 1.0 for comparisons across different queries.

Public benchmarks like BEIR⁴² are widely used for evaluating performance on retrieval tasks and additional tasks are covered by benchmarks like the Massive Text Embedding Benchmark (MTEB)⁴³. Practitioners are encouraged to use a standard library like those originated by Text Retrieval Conference (TREC) for consistent benchmarking with other methods, such as trec_eval⁴⁴ or python wrappers like pytrec_eval⁴⁵ when calculating precision, recall, nDCG and others. The optimal way to evaluate embedding models for a particular application may be application specific, but the intuition that more similar objects should be closer in the embeddings space is often a good start. Additional metrics such as model size, embedding dimension size, latency, and overall cost are also important considerations for production applications.

Search Example

Before diving into details about the different types of embeddings and the history of embedding model development let's explore the search example previously described above in more detail. The goal is to find relevant documents in a large corpus given a query from the user. One approach then is to construct a joint embedding model where the question and answer are mapped to similar locations in the embedding space. As the question and answer are semantically different, even if complementary, it is often helpful to use two neural nets that have been trained together with one for the question and one for the documents. A visual representation of this can be seen in Figure 9(b) as an asymmetric dual encoder with a separate network for the query and document in contrast to 9(a) displaying a single neural network used for both query and document, also called a siamese network.

Figure 2 is a diagram of a search question and answer application using a retrieval augmented generation (RAG) approach where embeddings are used to identify the relevant documents before inserting them into the prompt of an LLM for summarization for the end user. The application is split into two main processes. First, the index creation where documents are divided into chunks which are used to generate embeddings and stored in a vector database for low latency searches. Specifically, the document embedding portion of the model of the dual encoder neural network is used for these chunks. The second phase when the user asks a question to the system that is embedded using the query portion of the model and which will map to relevant documents when using a similarity search in the vector database. This second phase is very latency sensitive as the end user is actively waiting for a response so the ability to identify relevant documents from a large corpus in milliseconds using a vector database of documents is a critical piece of infrastructure.

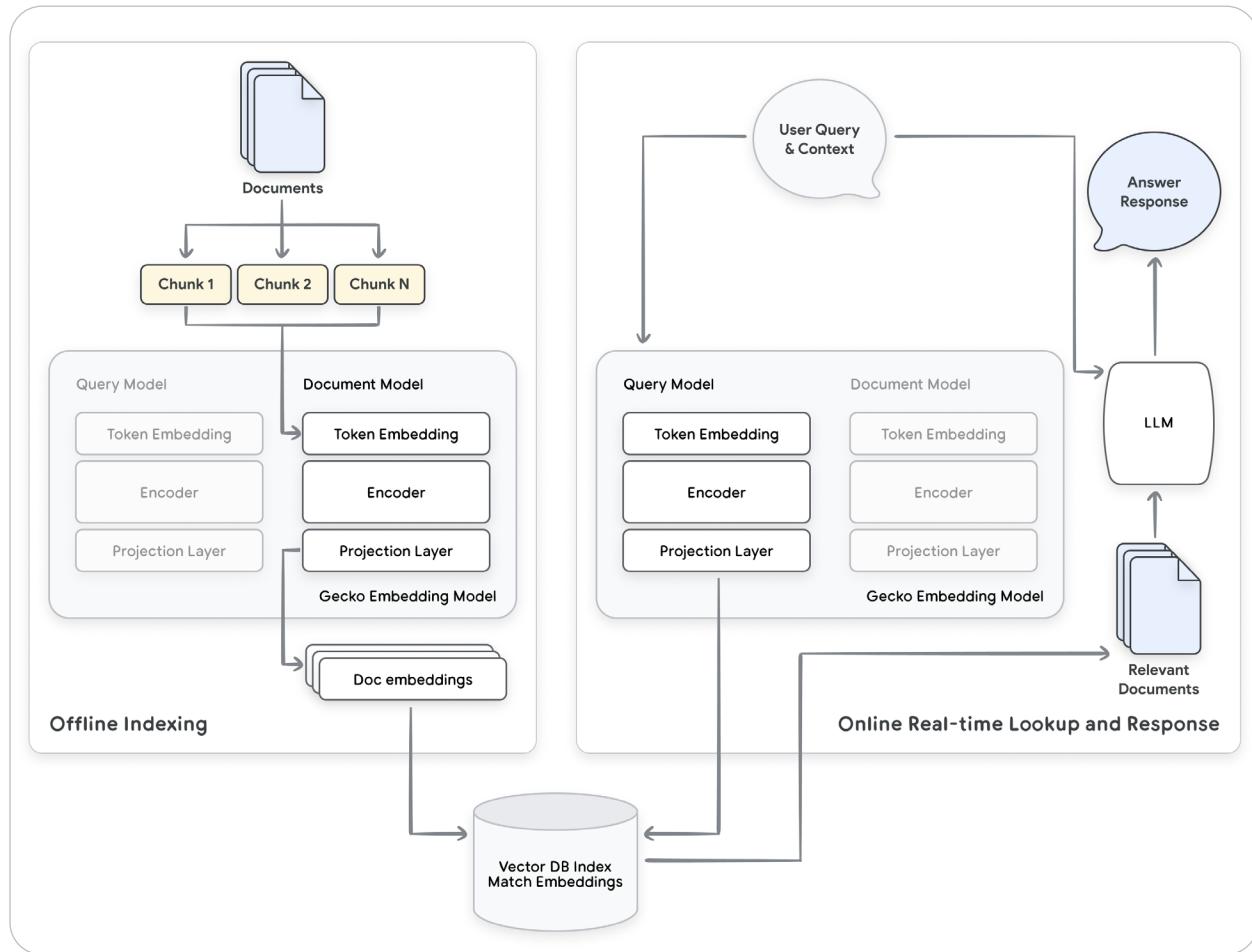


Figure 2. Example flow for RAG Search Application highlighting embeddings. Document embeddings are generated in the background and stored in a vector database. When the user enters a query, an embedding is generated using the query embedding portion of the dual encoder and used to look up relevant documents. Those documents can be inserted into the prompt for LLM to generate a relevant summary response for the user.

The quality of embedding models has been improving rapidly since the introduction of BERT and shows no signs of slowing any time soon. While LLMs have captured a lot of the attention in the AI space recently, the improvements in information retrieval and embedding models has also been transformative. The original BERT models were a leap forward at the time, and had an average score of 10.6 on the BEIR benchmark, current 2025 embeddings

from Google with a simple API call, and no AI knowledge required, now have an average BEIR score of 55.7. Models continue to improve rapidly, so when putting embedding models into production be sure to design with model upgrades in mind. Good evaluation suites designed for the particular application are critical for ensuring smooth upgrades. Choosing embedding models on platforms that have upgrade paths in place can help save developer time and reduce operational overhead for teams without deep AI expertise, for example Snippet 0 below uses a simple API call via Google Vertex.

Snippet 1 contains basic embedding code sample to illustrate some of the important concepts covered above for embeddings using the NFCorpus dataset⁴⁶ that contains health related questions and documents:

- The text documents with information relevant to the queries are embedded using the Google Vertex APIs for both high quality and operational ease. The RETRIEVAL_DOCUMENT task type is used as questions and answers are often phrased differently and use a single model with semantic similarity would result in a reduced performance compared to joint document and query embeddings.
- Embeddings are stored using the faiss⁴⁷ library for efficient similarity search.
- For a particular query the text embeddings is generated using the RETRIEVAL_QUERY task type.
- The query embedding is used by the faiss library to look up the ids for documents whose embeddings are close using the default euclidean distance metric.
- Embeddings for all of the queries are generated and most similar documents retrieved. Retrieval quality is evaluated against the “gold” values using the pytrec library to measure precision@1, recall@10, ndcg@10 metrics.

```

from beir import util
from beir.datasets.data_loader import GenericDataLoader
import faiss
import vertexai
from vertexai.language_models import TextEmbeddingInput, TextEmbeddingModel
import numpy as np
import pandas as pd
import pytrec_eval

def embed_text(texts, model, task, batch_size=5) :
    embed_mat = np.zeros((len(texts),768))
    for batch_start in range(0,len(texts),batch_size):
        size = min(len(texts) - batch_start, batch_size)
        inputs = [TextEmbeddingInput(texts[batch_start+i], task_type=task) for i in range(size)]
        embeddings = model.get_embeddings(inputs)
        for i in range(size) :
            embed_mat[batch_start + i, :] = embeddings[i].values
    return embed_mat

# Download smallish NFCorpus dataset of questions and document text
url = "https://public.ukp.informatik.tu-darmstadt.de/thakur/BEIR/datasets/nfcorpus.zip"
data_path = util.download_and_unzip(url, "datasets")
# Corpus of text chunks, text queries and "gold" set of query to relevant documents dict
corpus, queries, qrels = GenericDataLoader("datasets/nfcorpus").load(split="test")

# Note need to setup Google Cloud project and fill in id & location below
vertexai.init(project="PROJECT_ID", location="LOCATION")
model = TextEmbeddingModel.from_pretrained("text-embedding-005")
doc_ids,docs = zip(*[(doc_id, doc['text']) for doc_id,doc in corpus.items()])
q_ids,questions = zip(*[(q_id, q) for q_id,q in queries.items()])

```

Continues next page...

```

# Embed the documents and queries jointly using different models
doc_embeddings = embed_text(docs, model, "RETRIEVAL_DOCUMENT")
index = faiss.IndexFlatL2(doc_embeddings.shape[1])
index.add(doc_embeddings)

# Example look up example query to find relevant doc - note using 'RETRIEVAL_QUERY'
example_embed = embed_text(['Is Caffeinated Tea Really Dehydrating?'],
                           model, 'RETRIEVAL_QUERY')
s,q = index.search(example_embed,1)
print(f'Score: {s[0][0]:.2f}, Text: "{docs[q[0][0]]}"')
# Score: 0.49, Text: "There is a belief that caffeinated drinks, such as tea,
# may adversely affect hydration. This was investigated in a randomised
# controlled trial ... revealed no significant differences
# between tea and water for any of the mean blood or urine measurements..."

# Embed all queries to evaluate quality compared to "gold" answers
query_embeddings = embed_text(questions, model, "RETRIEVAL_QUERY")
q_scores, q_doc_ids = index.search(query_embeddings, 10)
# Create a dict of query to document scores dict for pytrec evaluation
# Multiply scores by -1 for sorting as smaller distance is better score for pytrec eval
search_qrels = { q_ids[i] : { doc_ids[_id] : -1*s.item() for _id, s in zip(q_doc_ids[i], q_scores[i])} for i in range(len(q_ids)) }
evaluator = pytrec_eval.RelevanceEvaluator(search_qrels, {'ndcg_cut.10','P_1','recall_10'})
eval_results = evaluator.evaluate(search_qrels)
df = pd.DataFrame.from_dict(eval_results, orient='index')
df.mean()
#P_1          0.517028 // precision@1
#recall_10    0.203507 // recall@10
#ndcg_cut_10  0.402624 // nDCG@10

```

Snippet 1. Example semantic search using text embeddings and evaluation for quality of retrieved documents.

Both training and evaluating neural networks requires datasets that contains pairs of questions and relevant documents such as the NFCorpus used in Snippet 0. The dataset that is best suited to train or evaluate for a particular application will depend on the nature of

that application. For example, a medical application will use different jargon and conventions than an application focusing on legal use cases. These labeled datasets can be expensive and time consuming to generate using human experts. The Gecko embedding model paper from Google DeepMind⁴⁸ discusses in detail how an LLM was used to generate a large set of synthetic question and document pairs for training, leading to an improved model and performance on many benchmarks. Using LLMs to assist experts in generating training data and also for the evaluation of answers can be an effective way to scale training, tuning, and evaluation datasets cost effectively.

Types of embeddings

Embeddings aim to obtain a low dimensional representation of the original data while preserving most of the ‘essential information’. The types of data an embedding represents can be of various different forms. Below you’ll see some standard techniques used for different types of data, including text and image.

Text embeddings

Text embeddings are used extensively as part of natural language processing (NLP). They are often used to embed the meaning of natural language in machine learning for processing in various downstream applications such as text generation, classification, sentiment analysis, and more. These embeddings broadly fall into two categories: token/word and document embeddings.

Before diving deeper into these categories, it’s important to understand the entire lifecycle of text: from its input by the user to its conversion to embeddings.

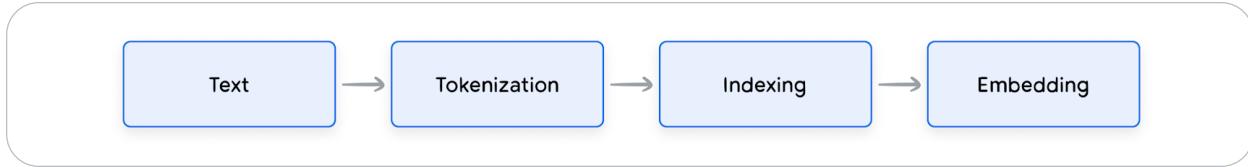


Figure 3. The process of turning text into embeddings

It all starts with the input string which is split into smaller meaningful pieces called tokens. This process is called *tokenization*. Commonly, these tokens are wordpieces, characters, words, numbers, and punctuations using one of the many existing tokenization techniques.¹ After the string is tokenized, each of these tokens is then assigned a unique integer value usually in the range: [0, cardinality of the total number of tokens in the corpus]. For example, for a 16 word vocabulary the IDs would range between 0-15. This value is also referred to as token ID. These tokens can be used to represent each string as a sparse numerical vector representation of documents used for downstream tasks directly, or after one-hot encoding. One-hot encoding is a binary representation of categorical values where the presence of a word is represented by 1, and its absence by 0. This ensures that the token IDs are treated as categorical values as they are, but often results in a dense vector the size of the vocabulary of the corpus. Snippet 2 and Figure 4 show an example of how this can be done using Tensorflow.

```

# Tokenize the input string data
from tensorflow.keras.preprocessing.text import Tokenizer
data = [
    "The earth is spherical.",
    "The earth is a planet.",
    "I like to eat at a restaurant."]

# Filter the punctiations, tokenize the words and index them to integers
tokenizer = Tokenizer(num_words=15, filters='!"#$%&()*+,-./:;<=>?[\\]^_`{|}~\\t\\n', lower=True,
split=' ')
tokenizer.fit_on_texts(data)

# Translate each sentence into its word-level IDs, and then one-hot encode those IDs
ID_sequences = tokenizer.texts_to_sequences(data)
binary_sequences = tokenizer.sequences_to_matrix(ID_sequences)
print("ID dictionary:\n", tokenizer.word_index)
print("\nID sequences:\n", ID_sequences)
print("\n One-hot encoded sequences:\n", binary_sequences)

```

Snippet 2. Tokenizing, indexing and one-hot encoding strings

```

ID dictionary:
{'the': 1, 'earth': 2, 'is': 3, 'a': 4, 'spherical': 5, 'planet': 6, 'i': 7, 'like': 8, 'to': 9, 'eat': 10, 'at': 11, 'restaurant': 12}

ID sequences:
[[1, 2, 3, 5], [1, 2, 3, 4, 6], [7, 8, 9, 10, 11, 4, 12]]

One-hot encoded sequences:
[[0. 1. 1. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 1. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 1. 1. 1. 1. 0. 0.]]
```

Figure 4. Output of Snippet 2

However, since these Integer IDs (or their corresponding one-hot encoded vectors) are assigned randomly to words, they lack any inherent semantic meaning. This is where embeddings are much more useful. Although it's possible to embed character and sub-word level tokens as well, let us look at word and document embeddings to understand some of the methods behind them.

Word embeddings

In this section, you'll see a few word embedding techniques and algorithms to both train and use word embeddings which were precursors to the modern text embedding currently being used. While there are many ML driven algorithms developed over time optimized for different objectives, the most common ones were GloVe,² SWIVEL,³ and Word2Vec.⁴ Word embeddings or sub-word embeddings can also be directly obtained from hidden layers of language models. However, the embeddings will be different for the same word in different contexts of the text. This section focuses on lightweight, context-free word embedding and leaves the context-aware document embeddings for the document embeddings section. Word embedding can be directly applied to downstream tasks like named entity extraction and topic modeling.

Word2Vec is a family of model architectures that operates on the principle of “the semantic meaning of a word is defined by its neighbors”, or words that frequently appear close to each other in the training corpus. This method can be both used to train your own embeddings from large datasets or be quickly integrated through one of the readily available pre-trained embeddings available online.⁵ The embeddings for each word - which are essentially fixed length vectors - are randomly initialized to kick off the process, resulting in a matrix of shape (size_of_vocabulary, size_of_each_embedding). This matrix can be used as a lookup table after the training process is completed using one of the following methods (see Figure 4).

- The Continuous bag of words (CBOW) approach: Tries to predict the middle word, using the embeddings of the surrounding words as input. This method is agnostic to the order of the surrounding words in the context. This approach is fast to train and is slightly more accurate for frequent words.
- The skip-gram approach: The setup is inverse of that of CBOW, with the middle word being used to predict the surrounding words within a certain range. This approach is slower to train but works well with small data and is more accurate for rare words.

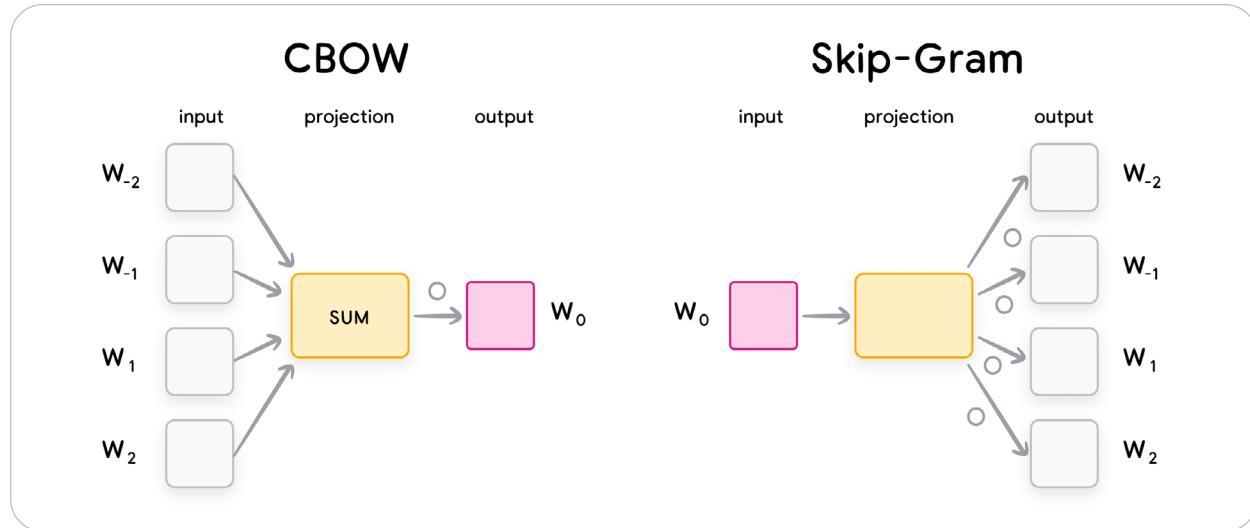


Figure 5. Diagram explaining how CBOW and Skip-Gram methods work

The Word2Vec algorithms can also be extended to the sub-word level, which has been the inspiration for algorithms such as FastText.⁶ However, one of the major caveats of Word2Vec is that although it accounts well for local statistics of words within a certain sliding window, it does not capture the global statistics (words in the whole corpus). This shortcoming is what methods like the GloVe algorithm address.

GloVe is a word embedding technique that leverages both global and local statistics of words. It does this by first creating a co-occurrence matrix, which represents the relationships between words. GloVe then uses a factorization technique to learn word representations from the co-occurrence matrix. The resulting word representations are able to capture both global and local information about words, and they are useful for a variety of NLP tasks.

In addition to GloVe, SWiVEL is another approach which leverages the co-occurrence matrix to learn word embeddings. SWiVEL stands for Skip-Window Vectors with Negative Sampling. Unlike GloVe, it uses local windows to learn the word vectors by taking into

account the co-occurrence of words within a fixed window of its neighboring words. Furthermore, SWIVEL also considers unobserved co-occurrences and handles it using a special piecewise loss, boosting its performance with rare words. It is generally considered only slightly less accurate than GloVe on average, but is considerably faster to train. This is because it leverages distributed training by subdividing the Embedding vectors into smaller sub-matrices and executing matrix factorization in parallel on multiple machines. Snippet 2 below demonstrates loading pre-trained word embeddings for both Word2Vec and GloVe and visualizing them in a 2D space, and computing nearest neighbors.

```

from gensim.models import Word2Vec
import gensim.downloader as api
import pprint
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
import numpy as np
def tsne_plot(models, words, seed=23):
    "Creates a TSNE models & plots for multiple word models for the given words"

    plt.figure(figsize=(len(models)*30, len(models)*30))
    model_ix = 0
    for model in models:
        labels = []
        tokens = []

        for word in words:
            tokens.append(model[word])
            labels.append(word)

        tsne_model = TSNE(perplexity=40, n_components=2, init='pca', n_iter=2500, random_state=seed)
        new_values = tsne_model.fit_transform(np.array(tokens))
        x = []
        y = []
        for value in new_values:
            x.append(value[0])
            y.append(value[1])

        model_ix +=1
        plt.subplot(10, 10, model_ix)
        for i in range(len(x)):
            plt.scatter(x[i],y[i])
            plt.annotate(labels[i],
                         xy=(x[i], y[i]),
                         xytext=(5, 2),
                         textcoords='offset points',
                         ha='right',
                         va='bottom')
    plt.tight_layout()
    plt.show()

v2w_model = api.load('word2vec-google-news-300')
glove_model = api.load('glove-twitter-25')
print("words most similar to 'computer' with word2vec and glove respectively:")
pprint.pprint(v2w_model.most_similar("computer")[:3])
pprint.pprint(glove_model.most_similar("computer")[:3])
pprint.pprint("2d projection of some common words of both models")
sample_common_words= list(set(v2w_model.index_to_key[100:10000]) & set(glove_model.index_to_key[100:10000]))[:100]
tsne_plot([v2w_model, glove_model], sample_common_words)

```

Snippet 3. Loading and plotting GloVe and Word2Vec embeddings in 2D

Figure 6 Shows semantically similar words are clustered differently for the two algorithms.

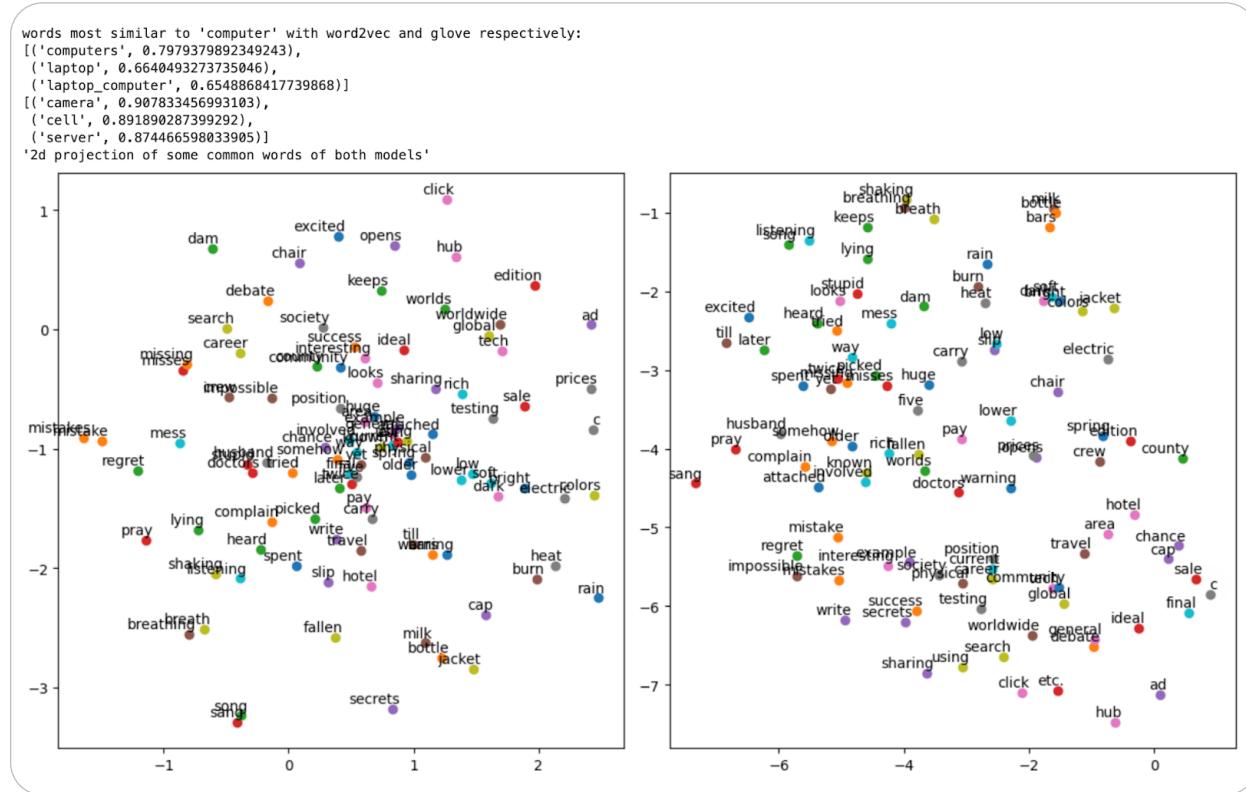


Figure 6. 2D visualization of pre-trained GloVe and Word2Vec word embeddings

Document embeddings

Embedding documents to low dimensional dense embedding has attracted long-lasting interests since the 1980s. Document embeddings can be used in various applications, including semantic search, topic discovery, classification, and clustering to embed the meaning of a series of words in paragraphs and documents and use it for various downstream applications. The evolution of the embeddings models can mainly be categorized into two stages: shallow Bag-of-words (BoW) models and deeper pretrained large language models.

Shallow BoW models

Early document embedding works follow the bag-of-words (BoW) paradigm, assuming a document is an unordered collection of words. These early works include latent semantic analysis (LSA)⁷ and latent dirichlet allocation (LDA).⁸ Latent semantic analysis (LSA) uses a co-occurrence matrix of words in documents and latent dirichlet allocation (LDA) uses a bayesian network to model the document embeddings. Another well known bag-of-words family of document embeddings is TF-IDF (term frequency-inverse document frequency) based models, which are statistical models that use the word frequency to represent the document embedding. TF-IDF-based models can either be a sparse embedding, which represents the term-level importance, or can be combined with word embeddings as a weighting factor to generate a dense embedding for the documents. For example, BM25⁴⁹, a TF-IDF-based bag-of-words model, is still a strong baseline in today's retrieval benchmarks.⁹

However, the bag-of-words paradigm also has two major weaknesses: both the word ordering and the semantic meanings are ignored. BoW models fail to capture the sequential relationships between words, which are crucial for understanding meaning and context.

Inspired by Word2Vec, Doc2Vec¹⁰ was proposed in 2014 for generating document embeddings using (shallow) neural networks. The Doc2Vec model adds an additional 'paragraph' embedding or, in other words, document embedding in the model of Word2Vec as illustrated in Figure 6. The paragraph embedding is concatenated or averaged with other word embeddings to predict a random word in the paragraph. After training, for existing paragraphs or documents, the learned embeddings can be directly used in downstream tasks. For a new paragraph or document, extra inference steps need to be performed to generate the paragraph or document embedding.

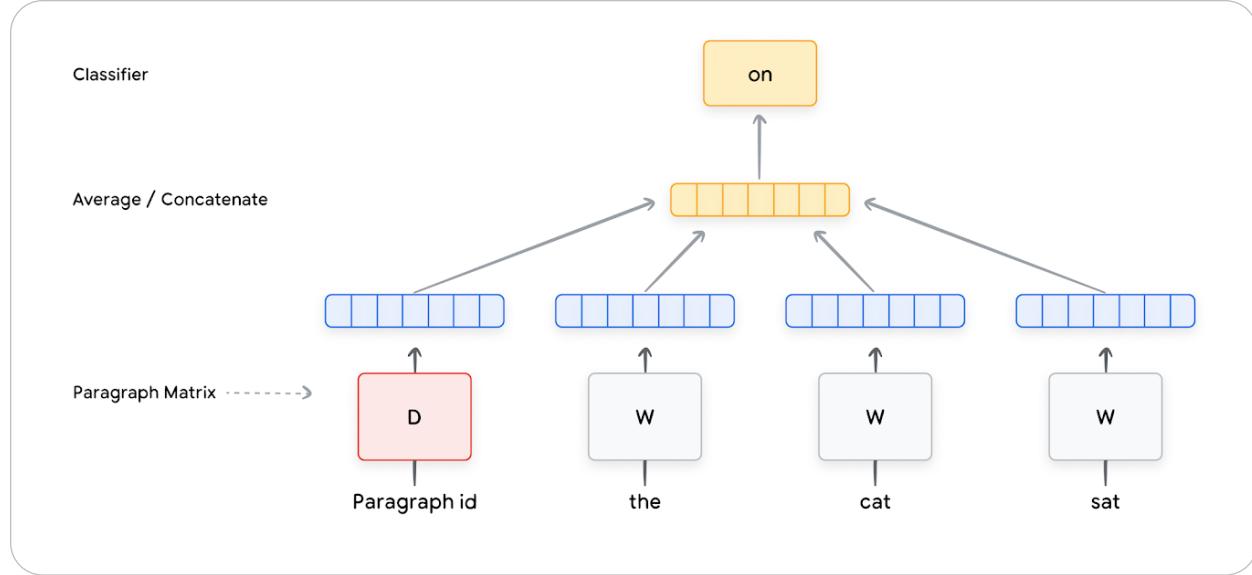


Figure 7. Doc2vec CBOW model

Snippet 4 below shows how you can train your own doc2Vec models on a custom corpus:

```

from gensim.test.utils import common_texts
from gensim.models.Doc2Vec import Doc2Vec, TaggedDocument
from gensim.test.utils import get_tmpfile
#train model on a sequence of documents tagged with their IDs
documents = [TaggedDocument(doc, [i]) for i, doc in enumerate(common_texts)]
model = Doc2Vec(documents, vector_size=8, window=3, min_count=1, workers=6)
# persist model to disk, and load it to infer on new documents
model_file = get_tmpfile("Doc2Vec_v1")
model.save(model_file)
model = Doc2Vec.load(model_file)
model.infer_vector(["human", "interface"])

```

Snippet 4. Self-supervised Training and inference using Doc2Vec on private corpus

Deeper pretrained large language models

Motivated by the development of deep neural networks, different embedding models and techniques were proposed, and the state-of-the-art models are progressing rapidly. Main changes of the models include:

1. Using more complex learning models, especially bi-directional deep neural network models.
2. The use of massive pre-training on unlabeled text.
3. The use of a subword tokenizer.
4. Using fine-tuning for various downstream NLP tasks.

In 2018, BERT¹¹ - which stands for bidirectional encoder representations from transformers - was proposed with groundbreaking results on 11 NLP tasks. Transformer, the model paradigm BERT based on, has become the mainstream model paradigm until today. Besides using a transformer as the model backbone, another key of BERT's success is from pre-training with a massive unlabeled corpus. In pretraining, BERT utilized masked language model (MLM) as the pre-training objective. It did this by randomly masking some tokens of the input and using the masked token id as the prediction objective. This allows the model to utilize both the right and left context to pretrain a deep bidirectional transformer. BERT also utilizes the next sentence prediction task in pretraining. BERT outputs a contextualized embedding for every token in the input. Typically, the embedding of the first token (a special token named [CLS]) is used as the embedding for the whole input.

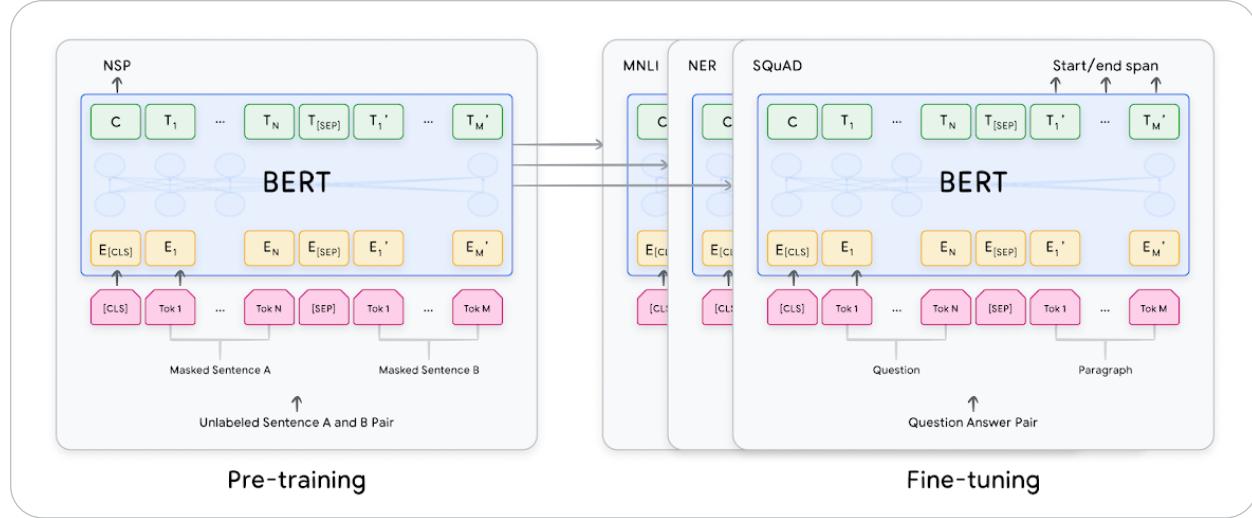


Figure 8. The BERT architecture

BERT became the base model for multiple embedding models, including Sentence-BERT¹², SimCSE¹³ and E5.¹⁴ Meanwhile, the evolution of language models - especially large language models - never stops. T5⁵⁰ was proposed in 2019 with up to 11B parameters. PaLM⁵¹ was proposed in 2022 to push the large language model to a surprising 540B parameters. Models like Gemini⁵² from Google, GPT⁵³ models from OpenAI and Llama⁵⁴ models from Meta are also evolving to newer generations at astonishing speed. Please refer to the whitepaper on Foundational models for more information about some common LLMs.

New embedding models based on large language models have been proposed. For example, GTR and Sentence-T5 show better performance on retrieval and sentence similarity (respectively) than BERT family models. Recently, a new embedding model powered by the Gemini model backbone has been released on Vertex AI, achieving superior results on all public benchmarks. Matryoshka Embeddings^{55,56} allow the downstream user to select how many dimensions are appropriate for their task to reduce data required for storage and indexing when possible.

Another approach to new embeddings models development is generating multi-vector embeddings instead of a single vector to enhance the representational power of the models. Embedding models in this family include ColBERT¹⁵ and XTR.¹⁶ ColPali⁵⁷ is also an approach using mult-vectors, but extending their application from text only to join embedding text and images for multi-modal documents.

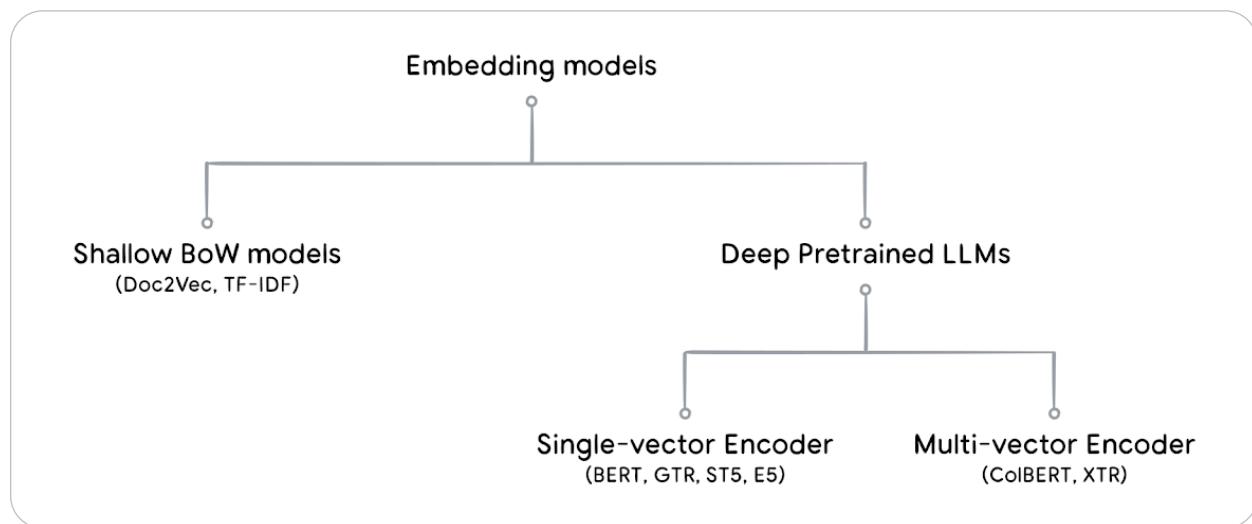


Figure 9. An illustration of the taxonomy diagram of the embedding models

Although the deep neural network models require a lot more data and compute time to train, they have much better performance compared to models using bag-of-words paradigms. For example, for the same word the embeddings would be different with different contexts, but by definition that is not true for bag-of-words. Snippet 4 demonstrates how pre-trained document embedding models from Tensorflow-hub¹⁷ (for example, Sentence t5)^A and Vertex AI^B can be used for training models with Keras and TF datasets. Vertex Generative AI text embeddings can be used with the Vertex AI SDK, Langchain, and Google's BigQuery (Snippet 5) for embedding and advanced workflows.¹⁸

A. Note: not all models on <https://tfhub.dev/> can be commercially used. Please check the licenses of the models and the training datasets and consult the legal team before commercial usage.

B. Note: not all models on <https://tfhub.dev/> can be commercially used. Please check the licenses of the models and the training datasets and consult the legal team before commercial usage.

```

import vertexai
from vertexai.language_models import TextEmbeddingInput, TextEmbeddingModel

# Set the model name. For multilingual: use "text-multilingual-embedding-002"
MODEL_NAME = "text-embedding-004"
# Set the task_type, text and optional title as the model inputs.
# Available task_types are "RETRIEVAL_QUERY", "RETRIEVAL_DOCUMENT",
# "SEMANTIC_SIMILARITY", # "CLASSIFICATION", and "CLUSTERING"
TASK_TYPE = "RETRIEVAL_DOCUMENT"
TITLE = "Google"
TEXT = "Embed text."

# Use Vertex LLM text embeddings
embeddings_vx = TextEmbeddingModel.from_pretrained("textembedding-gecko@004")

def LLM_embed(text):
    def embed_text(text):
        text_inp = TextEmbeddingInput(task_type="CLASSIFICATION", text=text.numpy())
        return np.array(embeddings_vx.get_embeddings([text_inp])[0].values)
    output = tf.py_function(func=embed_text, inp=[text], Tout=tf.float32)
    output.set_shape((768,))
    return output

# Embed strings using vertex LLMs
LLM_embeddings=train_data.map(lambda x,y: ((LLM_embed(x), y)))
# Embed strings in the tf.dataset using one of the tf hub models
embedding = "https://tfhub.dev/google/sentence-t5/st5-base/1"
hub_layer = hub.KerasLayer(embedding, input_shape=[], dtype=tf.string, trainable=True)

# Train model
model = tf.keras.Sequential()
model.add(hub_layer) # omit this layer if using Vertex LLM embeddings
model.add(tf.keras.layers.Dense(16, activation='relu'))
model.add(tf.keras.layers.Dense(1))
model.compile(optimizer='adam', loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])
history = model.fit(train_data.shuffle(100).batch(8))

```

Snippet 4. Creating & integrating text embeddings (Vertex, Tfhub) into keras text classification models

```
SELECT * FROM ML.GENERATE_TEXT_EMBEDDING(  
  MODEL my_project.my_company.llm_embedding_model,  
  (  
    SELECT review as content  
    FROM `bigquery-public-data.imdb.reviews`));
```

Snippet 5. Creating LLM based text embeddings in BigQuery for selected columns in a table

Image & multimodal embeddings

Much like text, it's also possible to create both image and multimodal embeddings.

Unimodal image embeddings can be derived in many ways such as by training a CNN or Vision Transformer model on a large scale image classification task (for example, Imagenet), and then using the penultimate layer as the image embedding. This layer has learnt some important discriminative feature maps for the training task. It contains a set of feature maps that are discriminative for the task at hand and can be extended to other tasks as well.

To obtain *multimodal embeddings*¹⁹ you take the individual unimodal text and image embeddings and create the joint embedding of their semantic relationships learnt via another training process. This gives you a fixed size semantic representation in the same latent space. Snippet 6 computes image and multimodal embeddings for images and text and can be used with a keras model directly (much like the text embedding example). Multimodal embedding approaches like ColPali⁵⁷ use image models to enable retrieval from text queries on multimodal documents without complex OCR or layout preprocesing. The model searches the images as they would be displayed to a user in a web browser or pdf viewer rahter than having to convert to a text only form for indexing.

```

import base64
import tensorflow as tf
from google.cloud import aiplatform
from google.protobuf import struct_pb2

#fine-tunable layer for image embeddings which can be used for downstream keras modelimage_
embed=hub.KerasLayer("https://tfhub.dev/google/imagenet/efficientnet_v2_imagenet21k_ft1k_s/feature_
vector/2",trainable=False)

class EmbeddingPredictionClient:
    """Wrapper around Prediction Service Client."""
    def __init__(self, project : str,
                 location : str = "us-central1",
                 apiRegionalEndpoint: str = "us-central1-aiplatform.googleapis.com"):
        client_options = {"apiEndpoint": apiRegionalEndpoint}
        self.client = aiplatform.gapic.PredictionServiceClient(clientOptions=clientOptions)
        self.location = location
        self.project = project

    def get_embedding(self, text : str = None, gs_image_path : str = None):
        #load the image from a bucket in google cloud storage
        with tf.io.gfile.GFile(gs_image_path, "rb") as f:
            image_bytes = f.read()
        if not text and not image_bytes:
            raise ValueError('At least one of text or image_bytes must be specified.')
        #Initialize a protobuf data struct with the text and image inputs
        instance = struct_pb2.Struct()
        if text:
            instance.fields['text'].string_value = text
        if image_bytes:
            encoded_content = base64.b64encode(image_bytes).decode("utf-8")
            image_struct = instance.fields['image'].struct_value
            image_struct.fields['bytesBase64Encoded'].string_value = encoded_content

        #Make predictions using the multimodal embedding model
        instances = [instance]
        endpoint = (f"projects/{self.project}/locations/{self.location}"
                    "/publishers/google/models/multimodalembedding@001")
        response = self.client.predict(endpoint=endpoint, instances=instances)

        text_embedding = None
        if text:
            text_emb_value = response.predictions[0]['textEmbedding']
            text_embedding = [v for v in text_emb_value]

        image_embedding = None
        if image_bytes:
            image_emb_value = response.predictions[0]['textEmbedding']
            image_embedding = [v for v in image_emb_value]

```

Continues next page...

```
return EmbeddingResponse (text_embedding=text_embedding, image_embedding=image_embedding)
#compute multimodal embeddings for text and images
client.get_embedding(text="sample_text", gs_image_path="gs://bucket_name..../image_filename..")
```

Snippet 6. Using Vertex API to create Multimodal embeddings Graph embeddings

Structured data embeddings

Structured data refers to data has a defined schema, like an table in a database where individual fields have known types and definitions. Unlike unstructured text and image data, where a pre-trained embedding model is typically available, we have to **create the embedding model for the structured data** since it would be specific to a particular application.

General structured data

Given a general structured data table, we can create embeddings for each row. This can be done by the ML models in the dimensionality reduction category, such as the PCA model.

One use case for these embeddings are for anomaly detection. For example, we can create embeddings for anomaly detection using large data sets of labeled sensor information that identify anomalous occurrences.²⁰ Another case use is to feed these embeddings to downstream ML tasks such as classification. Compared to using the original high-dimensional data, using embeddings to train a supervised model requires less data. This is particularly important in cases where training data is not sufficient.

User/item structured data

The input is no longer a general structured data table as above. Instead, the input includes the user data, item/product data plus the data describing the interaction between user and item/product, such as rating score.

This category is for recommendation purposes, as it maps two sets of data (user dataset, item/product/etc dataset) into the same embedding space. For recommender systems, we can create embeddings out of structured data that correlate to different entities such as products, articles, etc. Again, we have to create our own embedding model. Sometimes this can be combined with unstructured embedding methods when images or text descriptions are found.

Graph embeddings

Graph embeddings are another embedding technique that lets you represent not only information about a specific object but also its neighbors (namely, their graph representation). Take an example of a social network where each person is a node, and the connections between people are defined as edges. Using graph embedding you can model each node as an embedding, such that the embedding captures not only the semantic information about the person itself, but also its relations and associations hence enriching the embedding. For example, if two nodes are connected by an edge, the vectors for those nodes would be similar. You might then be able to predict who the person is most similar to and recommend new connections. Graph embeddings can also be used for a variety of tasks, including node classification, graph classification, link prediction, clustering, search, recommendation systems, and more. Popular algorithms^{21,22} for graph embedding include DeepWalk, Node2vec, LINE, and GraphSAGE.²³

Training Embeddings

Current embedding models usually use dual encoder (two tower) architecture. For example, for the text embedding model used in question-answering, one tower is used to encode the queries and the other tower is used to encode the documents. For the image and text embedding model, one tower is used to encode the images and the other tower is used to encode the text. The model can have various sub architectures, depending on how the model components are shared between the two towers. The following figure shows some architectures of the dual encoders.²⁴

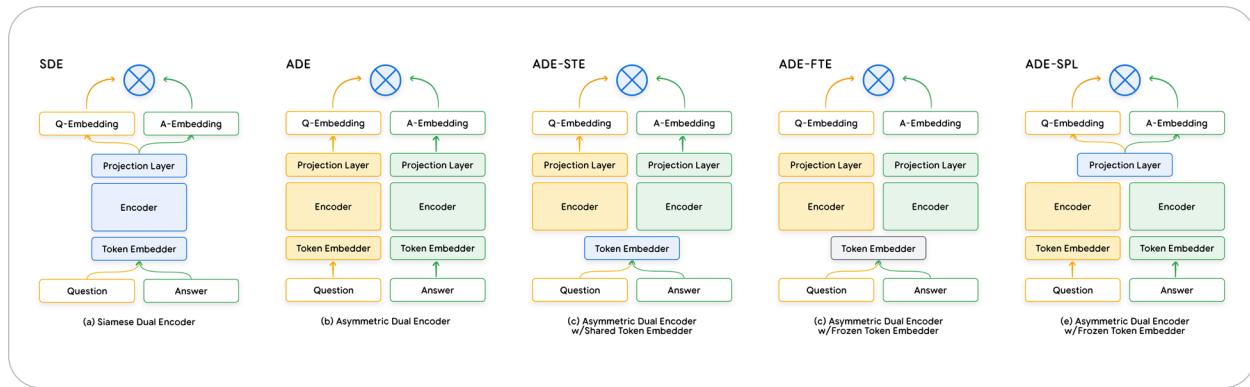


Figure 10. Some architectures of dual encoders

The loss used in embedding models training is usually a variation of contrastive loss, which takes a tuple of <inputs, positive targets, [optional] negative targets> as the inputs. Training with contrastive loss brings positive examples closer and negative examples far apart.

Similar to foundation model training, training of an embedding model from scratch usually includes two stages: pretraining (unsupervised learning) and fine tuning (supervised learning). Nowadays, the embedding models are usually directly initialized from foundation models such as BERT, T5, GPT, Gemini, CoCa. You can use these base models to leverage the massive knowledge that has been learned from the large-scale pretraining of the foundation

models. The fine-tuning of the embedding models can have one or more phases. The fine-tuning datasets can be created in various methods, including human labeling, synthetic dataset generation, model distillation, and hard negative mining.

To use embeddings for downstream tasks like classification or named entity recognition, extra layers (for example, softmax classification layer) can be added on top of the embedding models. The embedding model can either be frozen (especially when the training dataset is small), trained from scratch, or fine-tuned together with the downstream tasks.

Vertex AI provides the ability to customize the Vertex AI text embedding models.²⁵ Users can also choose to fine-tune the models directly. An example is fine tuning the BERT model using tensorflow model garden²⁶. You can also directly load the embedding models from tfhub and fine-tune on top of the model. Snippet 7 shows an example how to build a classifier based on tfhub models.

```
# Can switch the embedding to different embeddings from different modalities on #
tfhub. Here we use the BERT model as an example.
tfhub_link = "https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/4"

class Classifier(tf.keras.Model):
    def __init__(self, num_classes):
        super(Classifier, self).__init__(name="prediction")
        self.encoder = hub.KerasLayer(tfhub_link, trainable=True)
        self.dropout = tf.keras.layers.Dropout(0.1)
        self.dense = tf.keras.layers.Dense(num_classes)

    def call(self, preprocessed_text):
        encoder_outputs = self.encoder(preprocessed_text)
        pooled_output = encoder_outputs["pooled_output"]
        x = self.dropout(pooled_output)
        x = self.dense(x)
        return x
```

Snippet 7. Creating a Keras model using trainable tfhub layer

So far you've seen the various types of embeddings, techniques and best practices to train them for various data modalities, and some of their applications. The next section discusses how to persist and search the embeddings that have been created in a fast and scalable way for production workloads.

Vector search

Full-text keyword search has been the lynchpin of modern IT systems for years. Full-text search engines and databases (relational and non-relational) often rely on explicit keyword matching. For example, if you search for 'cappuccino' the search engine or database returns all documents that mention the exact query in the tags or text description. However, if the key word is misspelled or described with a differently worded text, a traditional keyword search returns incorrect or no results. There are traditional approaches which are tolerant of misspellings and other typographical errors. However, they are still unable to find the results having the closest underlying semantic meanings to the query. This is where vector search is very powerful: it uses the vector or embedded semantic representation of documents. As vector search works on any sort of embedding it also allows search on images, videos, and other data types in addition to text.

Vector search lets you to go beyond searching for exact query literals and allows you to search for the meaning across various data modalities. This allows you to find relevant results even when the wording is different. After you have a function that can compute embeddings of various items, you compute the embedding of the items of interest and store this embedding in a database. You then embed the incoming query in the same vector space as the items. Next, you have to find the best matches to the query. This process is analogous

to finding the most ‘similar’ matches across the entire collection of searchable vectors: similarity between vectors can be computed using a metric such as euclidean distance, cosine similarity, or dot product.

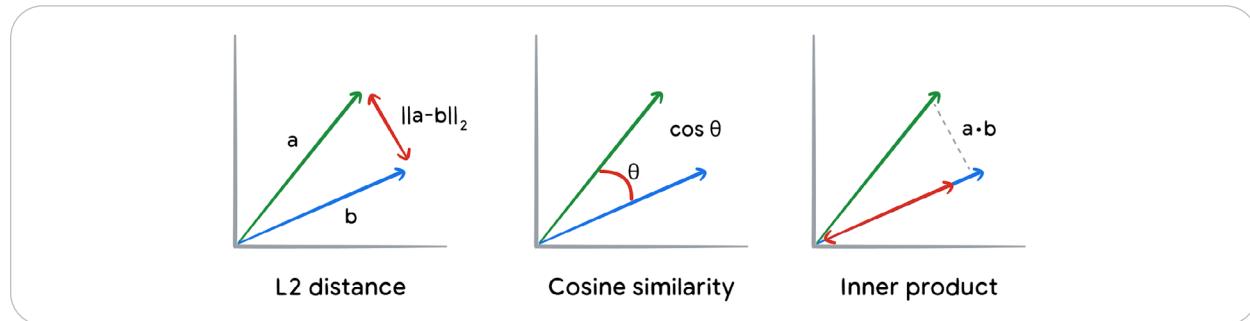


Figure 11. Visualization of how different metrics compute vector similarity

Euclidean distance (i.e., L2 distance) is a geometric measure of the distance between two points in a vector space. This works well for lower dimensions. Cosine similarity is a measure of the angle between two vectors. And inner/dot product, is the projection of one vector onto another. They are equivalent when the vector norms are 1. This seems to work better for higher dimensional data. Vector databases store and help manage and operationalize the complexity of vector search at scale, while also addressing the common database needs.

Important vector search algorithms

The most straightforward way to find the most similar match is to run a traditional linear search by comparing the query vector with each document vector and return the one with the highest similarity. However, the runtime of this approach scales linearly ($O(N)$) with the amount of documents or items to search. This approach is unacceptably slow for most

use cases involving several millions of documents or more. Using approximate nearest neighbour (ANN) search for that purpose is more practical. ANN is a technique for finding the closest points to a given point in a dataset with a small margin of error - but with far less computations required as the search space is greatly reduced to $O(\log N)$. There are many approaches with varying trade-offs across scale, indexing time, performance, simplicity and more.²⁶ They use one or more implementations of the following techniques: quantization, hashing, clustering and trees, among others. Some of the most popular approaches are discussed below.

Locality sensitive hashing & trees

Locality sensitive hashing (LSH)²⁷ is a technique for finding similar items in a large dataset. It does this by creating one or more hash functions that map similar items to the same hash bucket with high probability. This means that you can quickly find all of the similar items to a given item by only looking at the candidate items in the same hash bucket (or adjacent buckets) and do a linear search amongst those candidate pairs. This allows for significantly faster lookups within a specific radius. The number of hash functions/tables and buckets determine the search recall/speed tradeoff, as well as the false positive / true positive one. Having too many hash functions might cause similar items to different buckets, while too few might result in too many items falsely being hashed to the same bucket and the number of linear searches to increase.

Another intuitive way to think about LSH is grouping residences by their postal code or neighborhood name. Then based on where someone chooses to move you look at the residences for only that neighborhood and find the closest match.

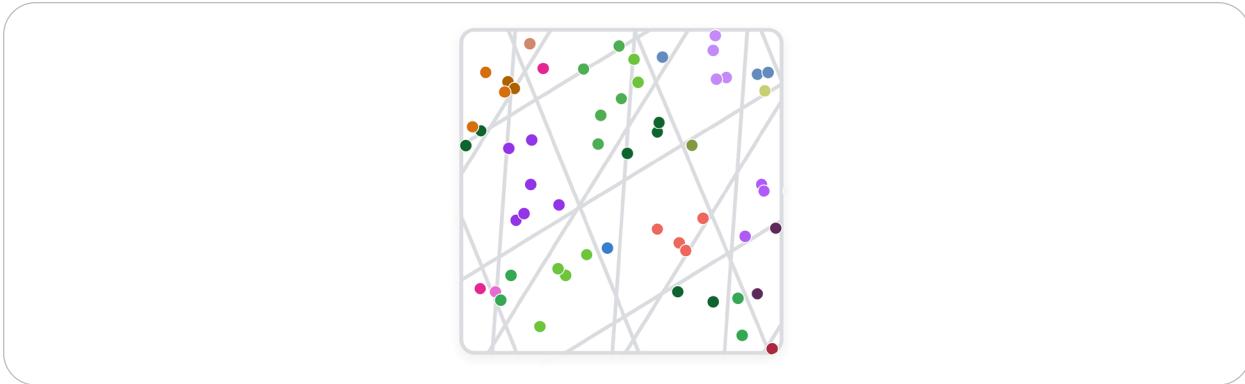


Figure 12. Visualization of how LSH uses random hyperplanes to partition the vector space

Tree-based algorithms work similarly. For example, the Kd-tree approach works by creating the decision boundaries by computing the median of the values of the first dimension, then that of the second dimension and so on. This approach is very much like a decision tree. Naturally this can be ineffective if searchable vectors are high dimensional. In that case, the Ball-tree algorithm is better suited. It is similar in functionality, except instead of going by dimension-wise medians it creates buckets based on the radial distance of the data points from the center. Here is an example of the implementation of these three approaches:

```

from sklearn.neighbors import NearestNeighbors
from vertexai.language_models import TextEmbeddingModel
from lshashing import LSHRandom
import numpy as np

model = TextEmbeddingModel.from_pretrained("textembedding-gecko@004")
test_items= [
    "The earth is spherical.",
    "The earth is a planet.",
    "I like to eat at a restaurant."]
query = "the shape of earth"
embedded_test_items = np.array([embedding.values for embedding in model.get_embeddings(test_items)])
embedded_query = np.array(model.get_embeddings([query])[0].values)

#Naive brute force search
n_neighbors=2
nbrs = NearestNeighbors(n_neighbors=n_neighbors, algorithm='brute').fit(embedded_test_items)
naive_distances, naive_indices = nbrs.kneighbors(np.expand_dims(embedded_query, axis = 0))

#algorithm- ball_tree due to high dimensional vectors or kd_tree otherwise
nbrs = NearestNeighbors(n_neighbors=n_neighbors, algorithm='ball_tree').fit(embedded_test_items)
distances, indices = nbrs.kneighbors(np.expand_dims(embedded_query, axis = 0))

#LSH
lsh_random_parallel = LSHRandom(embedded_test_items, 4, parallel = True)
lsh_random_parallel.knn_search(embedded_test_items, embedded_query, n_neighbors, 3, parallel = True)

#output for all 3 indices = [0, 1] , distances [0.66840428, 0.71048843] for the first 2 neighbours
#ANN retrieved the same ranking of items as brute force in a much scalable manner

```

Snippet 8. Using scikit-learn²⁸ and lshashing²⁹ for ANN with LSH, KD/Ball-tree and linear search

Hashing and tree-based approaches can also be combined and extended upon to obtain the optimal tradeoff between recall and latency for search algorithms. FAISS with HNSW and ScaNN^{32,33} are good examples.

Hierarchical navigable small worlds

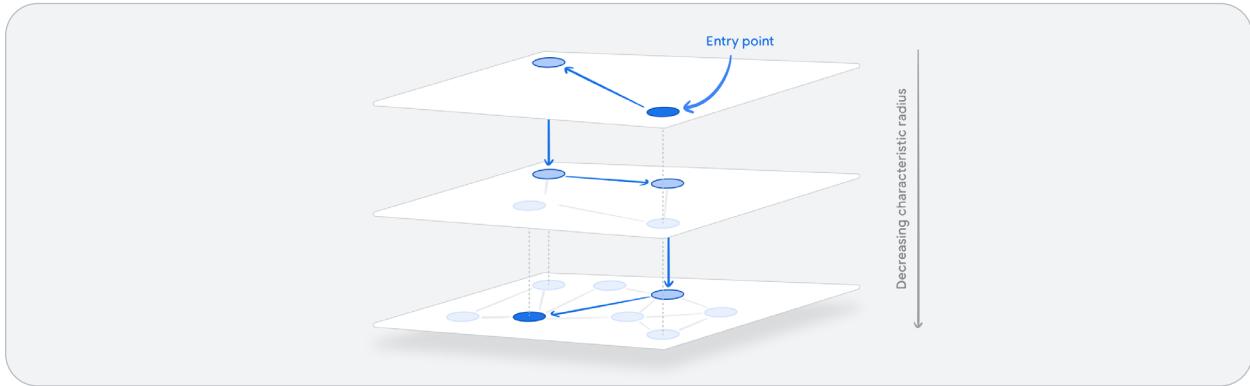


Figure 13. Diagram showing how HNSW ‘zooms in’ to perform ANN

One of the FAISS (Facebook AI similarity search) implementations leverages the concept of hierarchical navigable small world (HNSW)³⁰ to perform vector similarity search in sub-linear ($O(\log n)$) runtime with a good degree of accuracy. A HNSW is a proximity graph with a hierarchical structure where the graph links are spread across different layers. The top layer has the longest links and the bottom layer has the shortest ones. As shown in Figure 13, the search starts at the topmost layer where the algorithm greedily traverses the graph to find the vertex most semantically similar to the query. Once the local minimum for that layer is found, it then switches to the graph for the closest vertex on the layer below. This process continues iteratively until the local minimum for the lowest layer is found, with the algorithm keeping track of all the vertices traversed to return the K-nearest neighbors. This algorithm can be optionally augmented with quantization and vector indexing to boost speed and memory efficiency.

```

# Create an endpoint
my_index_endpoint = aiplatformMatchingEngineIndexEndpoint.create(
    display_name=f'{DISPLAY_NAME}-endpoint', public_endpoint_enabled=True
)

# NOTE : This operation can take upto 20 minutes
my_index_endpoint = my_index_endpoint.deploy_index(
    index=my_index, deployed_index_id=DEPLOYED_INDEX_ID
)

# retrieve the id of the most recently deployed index or manually look up the index
# deployed above
index_id=my_index_endpoint.deployed_indexes[-1].index.split('/')[-1]
endpoint_id= my_index_endpoint.name

# TODO : replace 1234567890123456789 with your acutial index ID
my_index = aiplatformMatchingEngineIndex(index_id)

# TODO : replace 1234567890123456789 with your acutial endpoint ID
# Be aware that the Index ID differs from the endpoint ID
my_index_endpoint = aiplatformMatchingEngineIndexEndpoint(endpoint_id)

# Input texts
texts= [
    "The earth is spherical.",
    "The earth is a planet.",
    "I like to eat at a restaurant.",
]

# Create a Vector Store
vector_store = VectorSearchVectorStore.from_components(
    project_id=PROJECT_ID,
    region=REGION,
    gcs_bucket_name=BUCKET,
    index_id=my_index.name,
    endpoint_id=my_index_endpoint.name,
    embedding=embedding_model,
    stream_update=True,
)

# Add vectors and mapped text chunks to your vectore store
vector_store.add_texts(texts=texts)

# Initialize the vectore_store as retriever
retriever = vector_store.as_retriever()

```

Continues next page...

```

retriever=vector_store.as_retriever(search_kwargs={'k':1 })

#create custom prompt for your use case
prompt_template="""You are David, an AI knowledge bot.
Answer the questions using the facts provided. Use the provided pieces of context to answer
the users question.
If you don't know the answer, just say that "I don't know", don't try to make up an answer.
{summaries}"""

messages = [
    SystemMessagePromptTemplate.from_template(prompt_template),
    HumanMessagePromptTemplate.from_template("{question}")
]
prompt = ChatPromptTemplate.from_messages(messages)

chain_type_kwargs = {"question": prompt}

#initialize your llm model
llm = VertexAI(model_name="gemini-pro")

#build your chain for RAG+C
chain= RetrievalQA.from_chain_type(llm=llm, chain_type="stuff",
retriever=retriever, return_source_documents=True)

#print your results with Markup language
def print_result(result):
    output_text = f"""### Question:
{query}
### Answer:
{result['result']}
### Source:
{' '.join(list(set([doc.page_content for doc in result['source_documents']])))}
"""

    return(output_text)

chain= "What shape is the planet where humans live?"
result = chain(query)
display(Markdown(print_result(result)))

```

Snippet 9. Build/deploy ANN Index for Vertex AI Vector Search and use RAG with LLM prompts to generate grounded results/sources.

```
import faiss
M=32 #creating high degree graph:higher recall for larger index & searching time
d=768 # dimensions of the vectors/embeddings
index = faiss.IndexHNSWFlat(d, M)
index.add(embedded_test_items) #build the index using the embeddings in Snippet 9
#execute the ANN search
index.search(np.expand_dims(embedded_query, axis=0), k=2)
```

Snippet 10. Indexing and executing ANN search with the FAISS library using HNSW

ScaNN

Google developed the scalable approximate nearest neighbor (ScaNN)^{31,32} approach which is used across a lot of its products and services. This includes being externally available to all customers of Google Cloud through the Vertex AI Vector Search and Google Cloud Databases, including AlloyDB, Cloud Spanner, and Cloud SQL MySQL. Below is how ScaNN uses a variety of steps to perform efficient vector search, with each one of them having their own subset of parameters.

The first step is the optional partitioning step during training: it uses one of the multiple algorithms available to partition the vector store into logical partitions/clusters where the semantically related are grouped together. The partitioning step is optional for small datasets. However, for larger datasets with >100k embedding vectors, the partitioning step is crucial since by pruning the search space it cuts down the search space by magnitudes therefore significantly speeds up the query. The space pruning is configured through the number of partitions and the number of partitions to search. A larger number leads to better recall but larger partition creation time. A good heuristic is to set the number of partitions to be the square root of the number of vectors.

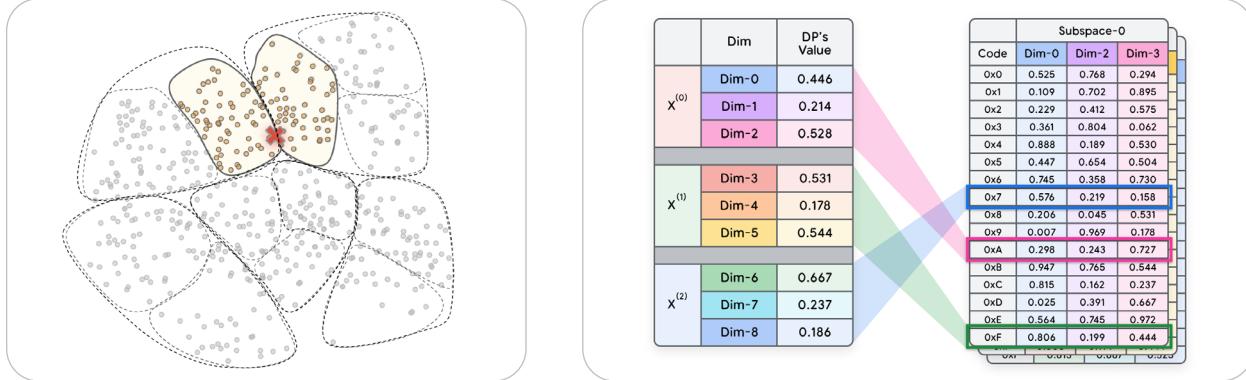
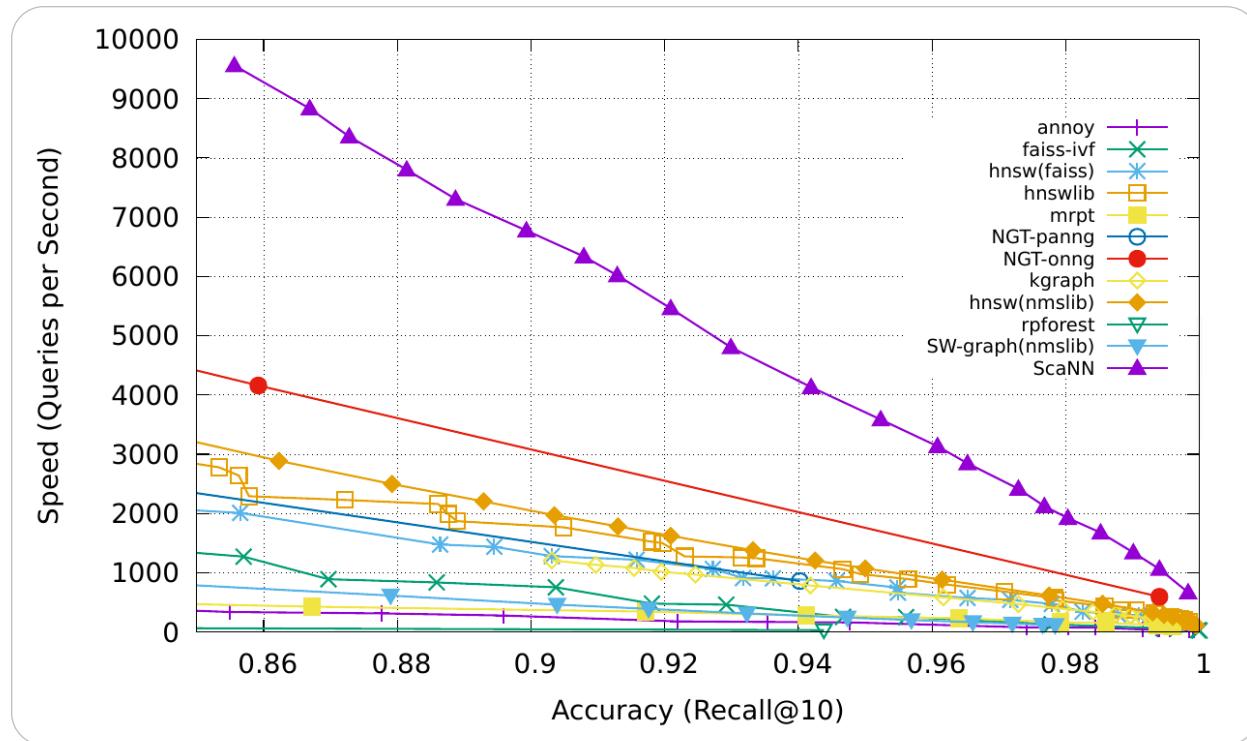


Figure 14. Search space partitioning & pruning (left) & Approximate scoring (right)

At query time ScaNN uses the user-specified distance measure to select the specified number of top partitions (a value specified by the user), and then executes the scoring step next. In this step ScaNN compares the query with all the points in the top partitions and selects the top K'. This distance computation can be configured as exact distance or approximate distance. The approximate distance computation leverages either standard product quantization or anisotropic quantization techniques, the latter of which is a specific method employed by ScaNN which gives the better speed and accuracy tradeoffs.

Finally, as a last step the user can optionally choose to rescore the user specified top K number of results more accurately. This results in an industry leading speed/accuracy tradeoff ScaNN is known for as can be inferred from Figure 14. Snippet 10 shows a code example.

Figure 15. Accuracy/speed tradeoffs for various SOTA ANN search algorithms⁵⁸

```

import tensorflow as tf
import tensorflow_recommenders as tfrs
from vertexai.language_models import TextEmbeddingModel, TextEmbeddingInput

# Embed documents & query(from snip 9.) and convert them to tensors and tf.datasets
embedded_query = tf.constant((LM_embed(query, "RETRIEVAL_QUERY")))
embedded_docs = [LM_embed(doc, "RETRIEVAL_DOCUMENT") for doc in searchable_docs]
embedded_docs = tf.data.Dataset.from_tensor_slices(embedded_docs).enumerate().batch(1)

# Build index from tensorflow dataset and execute ANN search based on dot product metric
scann = tfrs.layers.factorized_top_k.ScaNN(
    distance_measure= 'dot_product',
    num_leaves = 4, #increase for higher number of partitions / latency for increased recall
    num_leaves_to_search= 2) # increase for higher recall but increased latency
scann = scann.index_from_dataset(embedded_docs)
scann(embedded_query, k=2)

```

Snippet 11. Using Tensorflow Recommenders³³ to perform ANN search using the ScaNN algorithm

In this whitepaper we have seen both current and traditional ANN search algorithms: ScaNN, FAISS , LSH, KD-Tree, and Ball-tree, and examined the great speed/accuracy tradeoffs that they provide. However, to use these algorithms they need to be deployed in a scalable, secure and production-ready manner. For that we need vector databases.

Vector databases

Vector embeddings embody semantic meanings of data, while vector search algorithms provide a means for efficiently querying them. Historically traditional databases lacked the means to combine semantic meaning and efficient querying. This is what gave rise to vector databases, which are built ground-up to manage these embeddings for production scenarios. Due to the recent popularity of Generative AI, an increasing number of traditional

databases are starting to incorporate supporting vector search functionality in addition to traditional search ('hybrid search') functionalities. Let's look at the workflow for a simple Vector Database, with hybrid search capabilities.

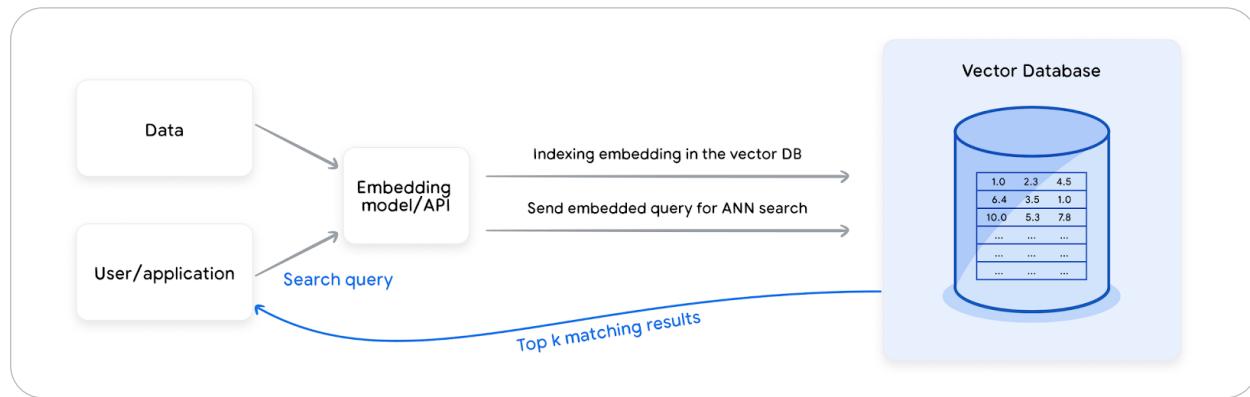


Figure 16. Populating and querying vector databases

Each vector database differs in its implementation, but the general flow is shown in Figure 16:

1. An appropriate trained embedding model is used to embed the relevant data points as vectors with fixed dimensions.
2. The vectors are then augmented with appropriate metadata and complementary information (such as tags) and indexed using the specified algorithm for efficient search.
3. An incoming query gets embedded with the appropriate model, and used to search for the most semantically similar items and their associated unembedded content/metadata. Some databases might provide caching and pre-filtering (based on tags) and post-filtering capabilities (reranking using another more accurate model) to further enhance the query speed and performance.

There are quite a few vector databases available today, each tailored to different business needs and considerations. A few good examples of commercially managed vector databases include Google Cloud's Vertex Vector Search,³⁴ Google Cloud's AlloyDB & Cloud SQL Postgres ElasticSearch,³⁵ and Pinecone³⁶ to name a few. Vertex AI Vector Search is a vector database built by Google that uses the ScaNN algorithm for fast vector search, while still maintaining all the security and access guarantees of Google Cloud. AlloyDB & Cloud SQL Postgres supports vector search through the OSS pgvector³⁷ extension, which allows for SQL queries to combine ANN search with traditional predicates and the usual transactional semantics for ANN search index. AlloyDB also has a ScaNN index extension that is a native implementation of ScaNN and is pgvector-compatible. Similarly, many other traditional databases have also started to add plugins to enable vector search. Pinecone³⁷ and Weaviate³⁹ leverage HNSW for their fast vector search in addition to the ability to filter data using traditional search. Amongst their open source peers: Weaviate³⁸ and ChromaDB³⁹ provide a full suite of functionality upon deployment and can be tested in memory as well during the prototyping phase.

Operational considerations

Vector Databases are critical to managing the majority of technical challenges that arise with storing and querying embeddings at scale. Some of these challenges are specific to the nature of vector stores, while others overlap with that of traditional databases. These include horizontal and vertical scalability, availability, data consistency, real time updates, backups, access control, compliance, and much more. However, there are also many more challenges and considerations you need to take into account while using embedding and vector stores.

Firstly, embeddings, unlike traditional content, can mutate over time. This means that the same text, image, video or other content could and should be embedded using different embedding models to optimize for the performance of the downstream applications. This is

especially true for embeddings of supervised models after the model is retrained to account for various drifts or changing objectives. Similarly, the same applies to unsupervised models when they are updated to a newer model. However, frequently updating the embeddings - especially those trained on large amounts of data - can be prohibitively expensive. Consequently, a balance needs to be struck. This necessitates a well-defined automated process to store, manage, and possibly purge embeddings from the vector databases taking the budget into consideration.

Secondly, while embeddings are great at representing semantic information, sometimes they can be suboptimal at representing literal or syntactic information. This is especially true for domain-specific words or IDs. These values are potentially missing or underrepresented in the data the embeddings models were trained on. For example, if a user enters a query that contains the ID of a specific number along with a lot of text, the model might find semantically similar neighbors which match the meaning of the text closely, but not the ID, which is the most important component in this context. You can overcome this challenge by using a combination of full-text search to pre-filter or post-filter the search space before passing it onto the semantic search module.

Another important point to consider is that depending on the nature of the workload in which the semantic query occurs, it might be worth relying on different vector databases. For example, for OLTP workloads that require frequent reads/write operations, an operational database like AlloyDB, Spanner, Postgres, or CloudSQL is the best choice. For large-scale OLAP analytical workloads and batch use cases, using BigQuery's vector search is preferable.

In conclusion, a variety of factors need to be considered when choosing a vector database. These factors include size and type of your dataset (some are good at sparse and others dense), business needs, the nature of the workload, budget, security, privacy guarantees, the needs for semantic and syntactic search as well as the database systems that are already

in use. In this section we have seen the various ANN search approaches as well the need and benefits of vector databases. The next section demonstrates an example of using a Vector AI Vector Search for semantic search.

Applications

Embeddings models are one of the fundamental machine learning models that power a variety of applications. We summarize some popular applications in the following table.

| Task | Description |
|--------------------------|--|
| Retrieval | Given a query and a set of objects (for example, documents, images, and videos), retrieve the most relevant objects. Based on the definition of relevant objects, the subtasks include question answering and recommendations. |
| Semantic text similarity | Determine whether two sentences have the same semantic meaning. The subtasks include: paraphrasing, duplicate detection, and bitext mining. |
| Classification | Classify objects into possible categories. Based on the number of labels, the subtasks include binary classification, multi-class classification, and multilabel classifications. |
| Clustering | Cluster similar objects together. |
| Reranking | Rerank a set of objects based on a certain query. |

Embeddings together with vector stores providing ANN are powerful tools which can be used for a variety of applications. These include Retrieval Augmented Generation (RAG) for LLMs, Search, Recommendation Systems, Anomaly detection, few shot- classification and much more.

For ranking problems like search and recommendations, embeddings are normally used at the first stage of the process. They retrieve the potentially good candidates that are semantically similar and consequently improve the relevance of search results. Since the amount of information to sort through can be quite large (in some cases even millions or billions) ANN techniques like ScANN greatly aids in scalably narrowing the search space. This initial set of results can be further refined with a more sophisticated model on this smaller set of candidates.

Let's look at an application which combines both LLMs and RAG to help answer questions.

Q & A with sources (retrieval augmented generation)

Retrieval augmented generation (RAG) for Q&A is a technique that combines the best of both worlds from retrieval and generation. It first retrieves relevant documents from a knowledge base and then uses prompt expansion to generate an answer from those documents. Prompt expansion is a technique that when combined with database search can be very powerful. With prompt expansion the model retrieves relevant information from the database (mostly using a combination of semantic search and business rules), and augments the original prompt with it. The model uses this augmented prompt to generate much more interesting, factual, and informative content than with retrieval or generation alone.

RAG can help with two common problems with LLMs: 1) their tendency to 'hallucinate' and generate factually incorrect but plausible sounding responses and 2) the high cost of retraining to keep up with current information as newer data can be supplied via the prompt, rather than at model training. Although RAG can reduce hallucinations, it does not completely eliminate them. What can help mitigate this problem further is to also return the sources from the retrieval and do a quick coherence check either by a human or an LLM. This ensures the

LLM response is consistent with the semantically relevant sources. Let's look at an example (Snippet 11) of RAG with sources, which can be scalably implemented using Vertex AI LLM text embeddings and Vertex AI Vector Search in conjunction with libraries like langchain.⁴⁰

```

# Before you start run this command:
# pip install --upgrade --user --quiet google-cloud-aiplatform langchain_google_vertexai
# after running pip install make sure you restart your kernel

# TODO : Set values as per your requirements
# Project and Storage Constants
PROJECT_ID = "<my_project_id>"
REGION = "<my_region>"
BUCKET = "<my_gcs_bucket>"
BUCKET_URI = f"gs://{BUCKET}"

# The number of dimensions for the text-embedding-005 is 768
# If other embedder is used, the dimensions would probably need to change.
DIMENSIONS = 768

# Index Constants
DISPLAY_NAME = "<my_matching_engine_index_id>"
DEPLOYED_INDEX_ID = "yourname01" # you set this. Start with a letter.

from google.cloud import aiplatform
from langchain_google_vertexai import VertexAIEMBEDDINGS
from langchain_google_vertexai import VertexAI
from langchain_google_vertexai import (
    VectorSearchVectorStore,
    VectorSearchVectorStoreDatastore,
)
from langchain.chains import RetrievalQA
from langchain.prompts.chat import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)
from IPython.display import display, Markdown

aiplatform.init(project=PROJECT_ID, location=REGION, staging_bucket=BUCKET_URI)
embedding_model = VertexAIEMBEDDINGS(model_name="text-embedding-005")

# NOTE : This operation can take upto 30 seconds
my_index = aiplatform.MatchingEngineIndex.create_tree_ah_index(
    display_name=DISPLAY_NAME,
    dimensions=DIMENSIONS,
    approximate_neighbors_count=150,
    distance_measure_type="DOT_PRODUCT_DISTANCE",
    index_update_method="STREAM_UPDATE", # allowed values BATCH_UPDATE , STREAM_UPDATE
)

```

Continues next page...

```

# Create an endpoint
my_index_endpoint = aiplatform.MatchingEngineEndpoint.create(
    display_name=f"{DISPLAY_NAME}-endpoint", public_endpoint_enabled=True
)

# NOTE : This operation can take upto 20 minutes
my_index_endpoint = my_index_endpoint.deploy_index(
    index=my_index, deployed_index_id=DEPLOYED_INDEX_ID
)

my_index_endpoint = my_index_endpoint.deploy_index(
    index=my_index, deployed_index_id=DEPLOYED_INDEX_ID
)
my_index_endpoint.deployed_indexes

# TODO : replace 1234567890123456789 with your acutial index ID
my_index = aiplatform.MatchingEngineIndex("1234567890123456789")

# TODO : replace 1234567890123456789 with your acutial endpoint ID
# Be aware that the Index ID differs from the endpoint ID
my_index_endpoint = aiplatform.MatchingEngineEndpoint("1234567890123456789")

from langchain_google_vertexai import (
    VectorSearchVectorStore,
    VectorSearchVectorStoreDatastore,
)

# Input texts
texts = [
    "The cat sat on",
    "the mat.",
    "I like to",
    "eat pizza for",
    "dinner.",
    "The sun sets",
    "in the west.",
]

```

Continues next page...

```
# Create a Vector Store
vector_store = VectorSearchVectorStore.from_components(
    project_id=PROJECT_ID,
    region=REGION,
    gcs_bucket_name=BUCKET,
    index_id=my_index.name,
    endpoint_id=my_index_endpoint.name,
    embedding=embedding_model,
    stream_update=True,
)

# Add vectors and mapped text chunks to your vectore store
vector_store.add_texts(texts=texts)

# Initialize the vectore_store as retriever
retriever = vector_store.as_retriever()

# perform simple similarity search on retriever
retriever.invoke("What are my options in breathable fabric?")
```

Snippet 12. Build/deploy ANN Index for Vertex AI Vector Search and use RAG with LLM prompts to generate grounded results/sources.

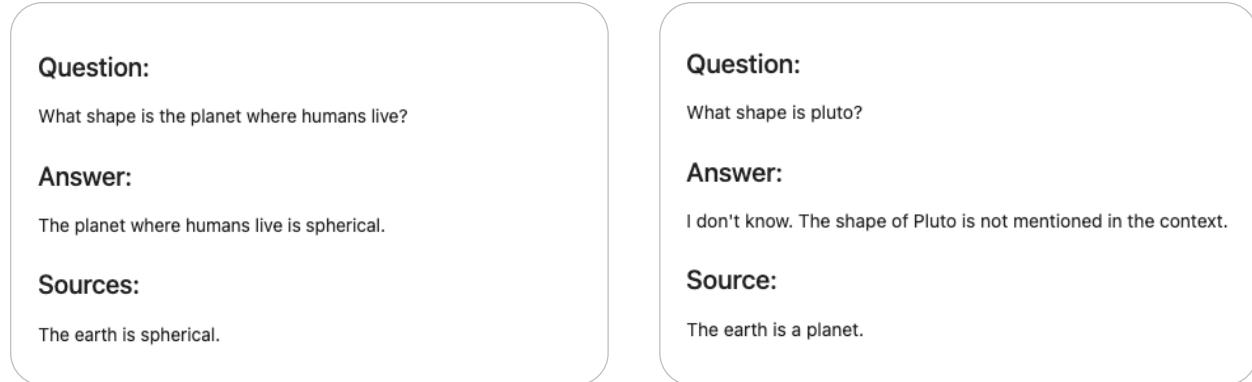


Figure 17. Model responses along with sources demonstrating the LLM being grounded in the database

As we can infer from Figure 16, the output not only grounds LLM in the semantically similar results retrieved from the database (hence refusing to answer when context cannot be found in the database). This not only significantly reduces hallucination, but also provides sources for verification, either human or using another LLM.

Summary

In this whitepaper we have discussed various methods to create, manage, store, and retrieve embeddings of various data modalities effectively in the context of production-grade applications. Creating, maintaining and using embeddings for downstream applications can be a complex task that involves several roles in the organization. However, by thoroughly operationalizing and automating its usage, you can safely leverage the incredible benefits they offer across some of the most important applications. Some key takeaways from this whitepaper include:

1. Choose your embedding model wisely for your data and use case. Ensure the data used in inference is consistent with the data used in training. The distribution shift from training to inference can come from various areas, including domain distribution shift or downstream

task distribution shift. If no existing embedding models fit the current inference data distribution, fine-tuning the existing model can significantly help on the performance. Another tradeoff comes from the model size. The large deep neural network (large multimodal models) based models usually have better performance but can come with a cost of longer serving latency. Using Cloud-based embedding services can conquer the above issue by providing both high-quality and low-latency embedding service. For most business applications using a pre-trained embedding model provides a good baseline, which can be further fine-tuned or integrated in downstream models. In case the data has an inherent graph structure, graph embeddings can provide superior performance.

2. Once your embedding strategy is defined, it's important to make the choice of the appropriate vector database that suits your budget and business needs. It might seem quicker to prototype with available open source alternatives, but opting for a more secure, scalable, and battle-tested managed vector database can save significant developer time. There are various open source alternatives using one of the many powerful ANN vector search algorithms, but ScaNN and HNSW have proven to provide some of the best accuracy and performance trade offs.
3. Embeddings combined with an appropriate ANN powered vector database is an incredibly powerful tool and can be leveraged for various applications, including Search, Recommendation systems, and Retrieval Augmented Generation for LLMs. This approach can mitigate the hallucination problem and bolster verifiability and trust of LLM-based systems.

Endnotes

1. Rai, A., 2020, Study of various methods for tokenization. In Advances in Natural Language Processing. Available at: https://doi.org/10.1007/978-981-15-6198-6_18
2. Pennington, J., Socher, R. & Manning, C., 2014, GloVe: Global Vectors for Word Representation. [online] Available at: <https://nlp.stanford.edu/pubs/glove.pdf>.
3. Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V. & Hinton, G., 2016, Swivel: Improving embeddings by noticing what's missing. ArXiv, abs/1602.02215. Available at: <https://arxiv.org/abs/1602.02215>.
4. Mikolov, T., Sutskever, I., Chen, K., Corrado, G. & Dean, J., 2013, Efficient estimation of word representations in vector space. ArXiv, abs/1301.3781. Available at: <https://arxiv.org/pdf/1301.3781.pdf>.
5. Rehurek, R., 2021, Gensim: open source python library for word and document embeddings. Available at: <https://radimrehurek.com/gensim/intro.html>.
6. Bojanowski, P., Grave, E., Joulin, A. & Mikolov, T., 2016, Enriching word vectors with subword information. ArXiv, abs/1607.04606. Available at: <https://arxiv.org/abs/1607.04606>.
7. Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., & Harshman, R., 1990, Indexing by latent semantic analysis. Journal of the American Society for Information Science, 41(6), pp. 391-407.
8. Blei, D. M., Ng, A. Y., & Jordan, M. I., 2001, Latent Dirichlet allocation. In T. G. Dietterich, S. Becker, & Z. Ghahramani (Eds.), Advances in Neural Information Processing Systems 14. MIT Press, pp. 601-608. Available at: <https://proceedings.neurips.cc/paper/2001/hash/296472c9542ad4d4788d543508116cbc-Abstract.html>.
9. Muennighoff, N., Tazi, N., Magne, L., & Reimers, N., 2022, Mteb: Massive text embedding benchmark. ArXiv, abs/2210.07316. Available at: <https://arxiv.org/abs/2210.07316>.
10. Le, Q. V., Mikolov, T., 2014, Distributed representations of sentences and documents. ArXiv, abs/1405.4053. Available at: <https://arxiv.org/abs/1405.4053>.
11. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K., 2019, BERT: Pre-training deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pp. 4171-4186. Available at: <https://www.aclweb.org/anthology/N19-1423/>.
12. Reimers, N. & Gurevych, I., 2020, Making monolingual sentence embeddings multilingual using knowledge distillation. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 254-265. Available at: <https://www.aclweb.org/anthology/2020.emnlp-main.21/>.

13. Gao, T., Yao, X. & Chen, D., 2021, Simcse: Simple contrastive learning of sentence embeddings. ArXiv, abs/2104.08821. Available at: <https://arxiv.org/abs/2104.08821>.
14. Wang, L., Yang, N., Huang, X., Jiao, B., Yang, L., Jiang, D., Majumder, R. & Wei, F., 2022, Text embeddings by weakly supervised contrastive pre-training. ArXiv. Available at: <https://arxiv.org/abs/2201.01279>.
15. Khattab, O. & Zaharia, M., 2020, colBERT: Efficient and effective passage search via contextualized late interaction over BERT. In Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 39-48. Available at: <https://dl.acm.org/doi/10.1145/3397271.3401025>.
16. Lee, J., Dai, Z., Duddu, S. M. K., Lei, T., Naim, I., Chang, M. W. & Zhao, V. Y., 2023, Rethinking the role of token retrieval in multi-vector retrieval. ArXiv, abs/2304.01982. Available at: <https://arxiv.org/abs/2304.01982>.
17. TensorFlow, 2021, TensorFlow hub, a model zoo with several easy to use pre-trained models. Available at: <https://tfhub.dev/>.
18. Zhang, W., Xiong, C., & Zhao, H., 2023, Introducing BigQuery text embeddings for NLP tasks. Google Cloud Blog. Available at: <https://cloud.google.com/blog/products/data-analytics/introducing-bigquery-text-embeddings>.
19. Google Cloud, 2024, Get multimodal embeddings. Available at: <https://cloud.google.com/vertex-ai/generative-ai/docs/embeddings/get-multimodal-embeddings>.
20. Pinecone, 2024, IT Threat Detection. [online] Available at: <https://docs.pinecone.io/docs/it-threat-detection>.
21. Cai, H., Zheng, V. W., & Chang, K. C., 2020, A survey of algorithms and applications related with graph embedding. In Proceedings of the 29th ACM International Conference on Information & Knowledge Management. Available at: <https://dl.acm.org/doi/10.1145/3444370.3444568>.
22. Cai, H., Zheng, V. W., & Chang, K. C., 2017, A comprehensive survey of graph embedding: problems, techniques and applications. ArXiv, abs/1709.07604. Available at: <https://arxiv.org/pdf/1709.07604.pdf>.
23. Hamilton, W. L., Ying, R. & Leskovec, J., 2017, Inductive representation learning on large graphs. In Advances in Neural Information Processing Systems 30. Available at: <https://cs.stanford.edu/people/jure/pubs/graphsage -nips17.pdf>.
24. Dong, Z., Ni, J., Bikel, D. M., Alfonseca, E., Wang, Y., Qu, C. & Zitouni, I., 2022, Exploring dual encoder architectures for question answering. ArXiv, abs/2204.07120. Available at: <https://arxiv.org/abs/2204.07120>.
25. Google Cloud, 2021, Vertex AI Generative AI: Tune Embeddings. Available at: <https://cloud.google.com/vertex-ai/docs/generative-ai/models/tune-embeddings>.

26. Matsui, Y., 2020, Survey on approximate nearest neighbor methods. ACM Computing Surveys (CSUR), 53(6), Article 123. Available at: <https://wangzwhu.github.io/home/file/acmmm-t-part3-ann.pdf>.
27. Friedman, J. H., Bentley, J. L. & Finkel, R. A., 1977, An algorithm for finding best matches in logarithmic expected time. ACM Transactions on Mathematical Software (TOMS), 3(3), pp. 209-226. Available at: <https://dl.acm.org/doi/pdf/10.1145/355744.355745>.
28. Scikit-learn, 2021, Scikit-learn, a library for unsupervised and supervised neighbors-based learning methods. Available at: <https://scikit-learn.org/>.
29. lshashing, 2021, An open source python library to perform locality sensitive hashing. Available at: <https://pypi.org/project/lshashing/>.
30. Malkov, Y. A., Yashunin, D. A., 2016, Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. ArXiv, abs/1603.09320. Available at: <https://arxiv.org/pdf/1603.09320.pdf>.
31. Google Research, 2021, A library for fast ANN by Google using the ScaNN algorithm. Available at: <https://github.com/google-research/google-research/tree/master/scann>.
32. Guo, R., Zhang, L., Hinton, G. & Zoph, B., 2020, Accelerating large-scale inference with anisotropic vector quantization. ArXiv, abs/1908.10396. Available at: <https://arxiv.org/pdf/1908.10396.pdf>.
33. TensorFlow, 2021, TensorFlow Recommenders, an open source library for building ranking & recommender system models. Available at: <https://www.tensorflow.org/recommenders>.
34. Google Cloud, 2021, Vertex AI Vector Search, Google Cloud's high-scale low latency vector database. Available at: <https://cloud.google.com/vertex-ai/docs/vector-search/overview>.
35. Elasticsearch, 2021, Elasticsearch: a RESTful search and analytics engine. Available at: <https://www.elastic.co/elasticsearch/>.
36. Pinecone, 2021, Pinecone, a commercial fully managed vector database. Available at: <https://www.pinecone.io>.
37. pgvector, 2021, Open Source vector similarity search for Postgres. Available at: <https://github.com/pgvector/pgvector>.
38. Weaviate, 2021, Weaviate, an open source vector database. Available at: <https://weaviate.io/>.
39. ChromaDB, 2021, ChromaDB, an open source vector database. Available at: <https://www.trychroma.com/>.

40. LangChain, 2021.,LangChain, an open source framework for developing applications powered by language model. Available at: <https://langchain.com>.
42. Thakur, N., Reimers, N., Ruckl'e, A., Srivastava, A., & Gurevych, I. (2021). BEIR: A Heterogenous Benchmark for Zero-shot Evaluation of Information Retrieval Models. ArXiv, abs/2104.08663.
Available at: <https://github.com/beir-cellar/beir>
43. Niklas Muennighoff, Nouamane Tazi, Loic Magne, and Nils Reimers. 2023. MTEB: Massive Text Embedding Benchmark. In Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, pages 2014–2037, Dubrovnik, Croatia. Association for Computational Linguistics.
Available at: <https://github.com/embeddings-benchmark/mteb>
44. Chris Buckley. trec_eval IR evaluation package. Available from https://github.com/usnistgov/trec_eval
45. Christophe Van Gysel and Maarten de Rijke. 2018. Pytrec_eval: An Extremely Fast Python Interface to trec_eval. In The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval (SIGIR '18). Association for Computing Machinery, New York, NY, USA, 873–876.
Available at: <https://doi.org/10.1145/3209978.3210065>
46. Boteva, Vera & Ghaliour Ghalandari, Demian & Sokolov, Artem & Riezler, Stefan. (2016). A Full-Text Learning to Rank Dataset for Medical Information Retrieval. 9626. 716-722. 10.1007/978-3-319-30671-1_58. Available at <https://www.cl.uni-heidelberg.de/statnlpgroup/nfcorpus/>
47. Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvassy, G., Mazaré, P.E., Lomeli, M., Hosseini, L. and Jégou, H., 2024. The Faiss library. arXiv preprint arXiv:2401.08281. Available at <https://arxiv.org/abs/2401.08281>
48. Lee, J., Dai, Z., Ren, X., Chen, B., Cer, D., Cole, J.R., Hui, K., Boratko, M., Kapadia, R., Ding, W. and Luan, Y., 2024. Gecko: Versatile text embeddings distilled from large language models. arXiv preprint arXiv:2403.20327.
Available at: <https://arxiv.org/abs/2403.20327>
49. Okapi BM25: a non-binary model” Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze. An Introduction to Information Retrieval, Cambridge University Press, 2009, p. 232.
50. Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach. Learn. Res. 21, 1, Article 140 (January 2020), 67 pages.
Available at <https://dl.acm.org/doi/abs/10.5555/3455716.3455856>

51. Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sashank Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2023. PaLM: scaling language modeling with pathways. *J. Mach. Learn. Res.* 24, 1, Article 240 (January 2023), 113 pages. Available at <https://dl.acm.org/doi/10.5555/3648699.3648939>
52. Gemini: A Family of Highly Capable Multimodal Models, Gemini Team, Dec 2023. Available at: https://storage.googleapis.com/deepmind-media/gemini/gemini_1_report.pdf
53. Radford, Alec and Karthik Narasimhan. "Improving Language Understanding by Generative Pre-Training." (2018). Available at: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf
54. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F. and Rodriguez, A., 2023. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971. Available at: <https://arxiv.org/abs/2302.13971>
55. Kusupati, A., Bhatt, G., Rege, A., Wallingford, M., Sinha, A., Ramanujan, V., Howard-Snyder, W., Chen, K., Kakade, S., Jain, P. and Farhadi, A., 2022. Matryoshka representation learning. Advances in Neural Information Processing Systems, 35, pp.30233-30249. Available at: https://proceedings.neurips.cc/paper_files/paper/2022/file/c32319f4868da7613d78af9993100e42-Paper-Conference.pdf
56. Nair, P., Datta, P., Dean, J., Jain, P. and Kusupati, A., 2025. Matryoshka Quantization. arXiv preprint arXiv:2502.06786. Available at: <https://arxiv.org/abs/2502.06786>
57. Faysse, M., Sibille, H., Wu, T., Omrani, B., Viaud, G., Hudelot, C. and Colombo, P., 2024. Colpali: Efficient document retrieval with vision language models. arXiv preprint arXiv:2407.01449. Available at: <https://arxiv.org/abs/2407.01449>
58. Aumüller, M., Bernhardsson, E. and Faithfull, A., 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87, p.101374.

