**Advanced Gen AI from Skillboost**

**Few image generation models :**

**Varational autoencoders (VAE’s**) 🡪 These models learn to compress an image to smaller , simpliefed representation and then decode it back to original size. In doing so, they learn the essential features

**Generative Adversarial networks ( GAN ) 🡪**This are popular methods involves two competing neural networks.

Generator 🡪 Create fake images

Discriminator 🡪Tries to tell the difference between fake images and real ones.

As they compete, the generator gets better at making realistic fakes, and the discriminator gets better at spotting them, This is technology behind deepfake

**Autoregressive models 🡪** These models build an image one pixel at a time in sequence, much like how a LLM writes one word at a time

**Diffusion models 🡪**

This are new trend of gen AI models. It comes from concept of thermodynamics.

The core concept :

**Forward process ( adding noise ) 🡪** Start with a clear image and slowly add little bit of random “noise” over many steps. By the end, original image is completely destroyed and has turned into pure, random static.

**Reverse process ( removing noise ) 🡪** This is magic, ML model is trained to do exact opposite .It learns how to look at a nosily image and predict exactly what noise was added to it

**How is diffusion model trained ?**

1. Take a real image from a large dataset.
2. Randomly pick a step in the "adding noise" process to create a noisy version of that image.
3. Give this noisy image to the AI model.
4. The model's goal is to predict the *exact noise* that was added.
5. Since we know what noise we added, we can compare the model's prediction to the actual noise.
6. The model adjusts itself to minimize the difference (the error).
7. By repeating this millions of times with many different images and noise levels, the model becomes extremely good at identifying and removing noise.

**How Does a Trained Model Create a New Image?**

Once the model is trained, generating a brand-new image is surprisingly straightforward:

1. Start with a screen of pure random noise (static).
2. Feed this noise into the trained model.
3. The model predicts the "noise" it sees in the image.
4. Subtract this predicted noise from the image. The result is a slightly less noisy, more structured image.
5. Repeat this process over and over. With each step, the model removes more noise, and a coherent image begins to appear.
6. After all the steps are complete, the initial random static has been transformed into a completely new, realistic-looking image.

**Use Cases and Future of Diffusion Models**

Diffusion models are very versatile:

* **Unconditioned Generation:** The model can be trained on a specific type of image (like faces) and then generate new, unique faces on its own. It's also used for **super-resolution**, which enhances the quality of blurry, low-resolution images.
* **Conditioned Generation:** This is where you give the model instructions.
  + **Text-to-Image:** You provide a text prompt (e.g., "an astronaut riding a horse"), and the model generates an image that matches.
  + **Image Editing (Inpainting):** You can remove objects from an image or add new ones based on text instructions.

The talk concludes by mentioning that many state-of-the-art systems, like Google's **Imagen**, combine the power of diffusion models with Large Language Models (LLMs) to create incredibly detailed and context-aware images from text. This technology is rapidly improving and being integrated into enterprise-level products.

**Attention mechanism 🡪** It is technique on neural networks that allows to focus on specific part of input sequence and it done by assigning specific weights to different part of input sequence and important words receive the highest weights

Encoder passed more data to decoder

It takes extra step before it produces the output

**Encoder and Decoder Architecture**

**What is the Encoder-Decoder Architecture?**

This is a type of AI model designed for **sequence-to-sequence** tasks. This means it takes a sequence of items (like words in a sentence) as input and produces another sequence of items as output.

* **Example 1: Translation:**
  + **Input:** The English sentence, "The cat ate the mouse."
  + **Output:** The French translation, "Le chat a mange la souris."
* **Example 2: Chatbots/LLMs:**
  + **Input:** A user's prompt or question.
  + **Output:** The model's generated response.

The architecture has two main parts that work one after the other:

* **The Encoder:** Its only job is to read the entire input sentence and compress it into a single numerical representation (a vector). This vector acts as a summary of the meaning of the whole sentence.
* **The Decoder:** Its job is to take that numerical summary from the encoder and "un-pack" it into a new sentence in the target language or format.

The talk mentions that while early models used a simpler mechanism called a **Recurrent Neural Network (RNN)**, modern powerful models like those at Google use a more advanced building block called a **Transformer**, which is based on the "attention mechanism."

**2. How Are These Models Trained?**

Training is the process of teaching the model to produce the correct output.

* **The Dataset:** You need a huge dataset of example pairs. For a translation model, this would be millions of sentences and their correct translations.
* **The Process:**
  + You give the model an input sentence from your dataset.
  + The model tries to generate the correct output.
  + You compare the model's output to the actual correct output from the dataset. The difference between them is called the **error**.
  + The model then adjusts its internal parameters (weights) to reduce this error. This is repeated millions of times.
* **A Special Trick: "Teacher Forcing"** During training, the decoder generates the output one word at a time. To predict the third word, it needs to know what the first two words were. If the model generated a wrong second word, it could throw off the rest of the sentence. To prevent this and speed up training, a method called **teacher forcing** is used. Instead of feeding the decoder its own previous (and possibly incorrect) word, you always feed it the correct previous word from the dataset. You are "forcing" it to learn from the right path.

**3. How Does a Trained Model Generate New Text? (Serving)**

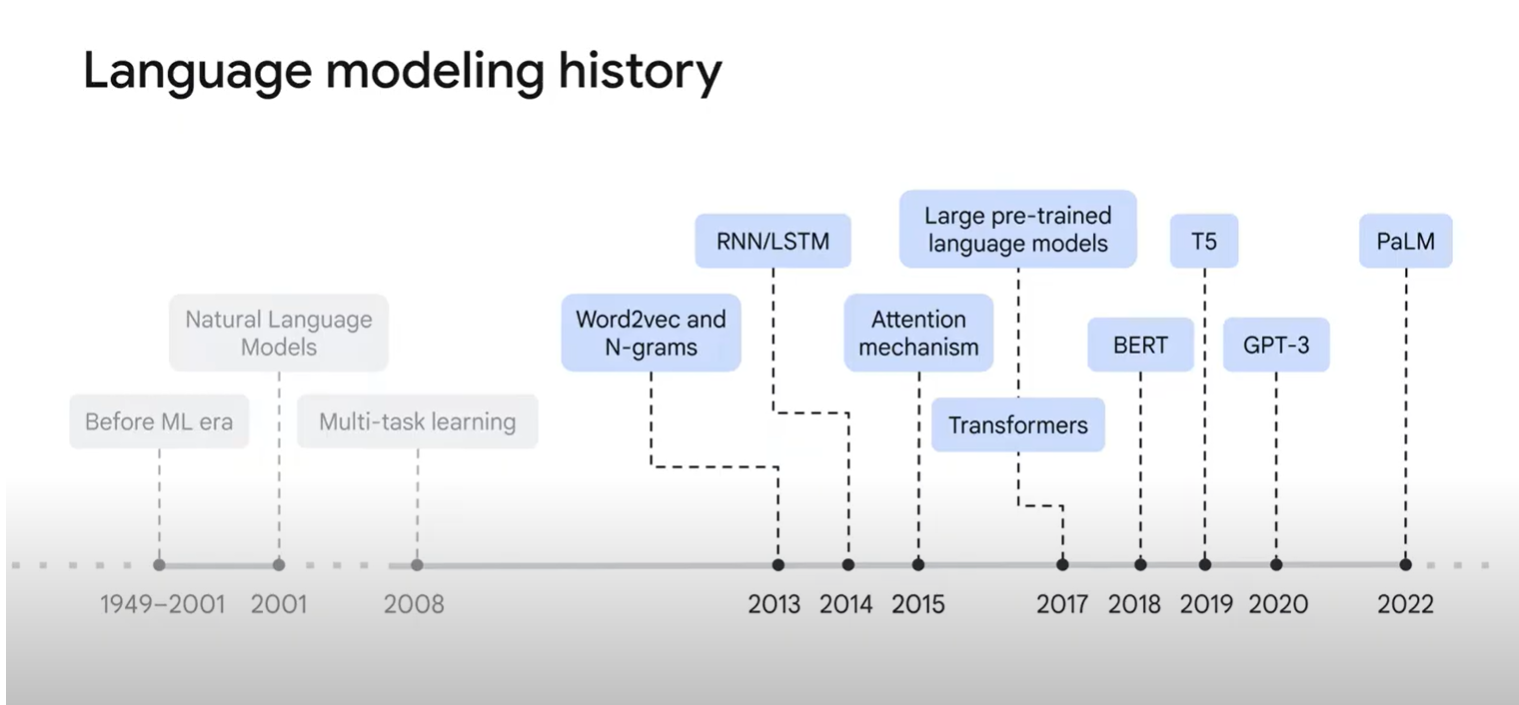
Once the model is trained, it can be used to generate new text for inputs it has never seen before. This is called "serving."

1. The input sentence (e.g., a prompt) is fed into the **encoder**, which creates the meaning vector.
2. This vector is passed to the **decoder**. To kick things off, a special "GO" token is given to the decoder.
3. The decoder uses the meaning vector and the "GO" token to predict the very first word of the output sentence.
4. This newly generated first word is then fed back into the decoder as input.
5. The decoder now uses the meaning vector and the first word to predict the second word.
6. This process repeats—generating one word at a time and feeding it back in—until the model generates a special "end of sentence" token or reaches a maximum length.
7. **Choosing the Next Word:** The decoder doesn't just output one word; it outputs a probability for every word it knows. There are different strategies to pick the next word:
   1. **Greedy Search:** The simplest method. Always pick the single word with the highest probability.
   2. **Beam Search:** A better method. It keeps track of several of the most likely sentence fragments (or "beams") at each step, leading to more fluent and accurate results.

Finally, the talk mentions that this encoder-decoder concept is the foundation for more advanced architectures like **Transformers** and **BERT**, which are the powerhouses behind today's state-of-the-art Generative AI.

`

Google History :



Word2 vec 🡪 RNN/LSTM 🡪 Attenntion mechanism 🡪 Transformers 🡪 BERT 🡪 T5 🡪 GPT3 🡪 PaLM

RNN/LSTM ( seq2seq models) 🡪 Helped in improve the performance of ML models on NLP tasks such as translation and text classification

Before Attention mechanism , the word vectors does not have context and from attention mechanism context is gained by assigning some weights and scores

**Transformers 🡪** Take advantage of parallezation GPU/TPU. Process all tokens at once

**Encoder**

1. Self attention 🡪 I/p of encoder first flows to self attention which helps to look at relevant part of words and identify the centre word in sentence. Dependency is maintained/exists. The I/p embedding is broken up into query, key and value vectors and these vectors are computed using weights that transformers learns during the training process and all these computations happen parallel in model. Once we have Q,K,V vectors we will multiple by softmax score. This will help in the values of words we need to focus ( relevant words).Then sum up the weight value vectors which produces the o/p of the self attention layer at the position for first word and

2. Feedforward 🡪 O/p of self attention layer is fed to feedforward NN. Dependency is NOT maintained/exists. So parallel flow can happen here

**Decoder**

1. Self attention 🡪 I/p of encoder first flows to self attention which helps to look at relevant part of words and identify the centre word in sentence

2. Encoder-decoder attention

3. Feedforward 🡪 O/p of self attention layer is fed to feedforward NN

**BERT Models:**

**1. The Problem Transformers Solved**

Before Transformers, AI models had a hard time understanding that the meaning of a word can change based on the sentence it's in. For example, the word "bank" means something different in "river bank" than it does in "bank robber." Older models would give "bank" the same meaning in both cases. Transformers, introduced in a 2017 paper called "Attention Is All You Need," solved this by using a mechanism called **attention**.

**2. What is a Transformer?**

A Transformer is a type of AI architecture designed to handle sequences of data, like sentences. It has two main parts:

* **The Encoder:** Its job is to read the input sentence and build a rich, context-aware numerical representation of it.
* **The Decoder:** Its job is to take that representation and generate an output sentence, for example, a translation.

Both the encoder and decoder are built from smaller, repeating blocks. A typical Transformer might have 6 encoder blocks stacked on top of each other and 6 decoder blocks.

**3. The Magic Ingredient: The "Attention" Mechanism**

The core idea of Transformers is "self-attention." Here’s a simple way to think about it:

1. When the model processes a word in a sentence, self-attention allows it to look at **all the other words** in the same sentence.
2. It then assigns a "score" or "weight" to every other word, deciding how relevant it is to understanding the current word.
3. For example, when processing the word "it" in the sentence "The cat drank the milk because **it** was thirsty," the attention mechanism learns to pay more attention to "cat" than to "milk."

**How it works technically (in simple terms):** For each word, the model creates three vectors: a **Query (Q)**, a **Key (K)**, and a **Value (V)**.

* The **Query** is like a question: "What am I looking for?"
* The **Key** is like a label on other words: "Here's what I have to offer."
* The **Value** is the actual meaning of the other words.

The model compares the Query of one word with the Key of all other words to calculate a score. This score determines how much "attention" to pay to each word's Value. The final representation of the word is a weighted sum of the Values of all words in the sentence.

This whole process is done multiple times in parallel ("multi-headed attention"), allowing the model to focus on different sentence relationships at the same time (e.g., one "head" might focus on grammar, another on meaning).

**4. What is BERT?**

BERT (which stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers) is a famous and powerful model developed by Google in 2018. It is now a core part of Google Search.

* **Architecture:** BERT is an **encoder-only** model. It doesn't have a decoder because its main job isn't to generate new sentences, but to deeply *understand* them.
* **Bidirectional:** This is BERT's superpower. Unlike older models that read a sentence from left-to-right or right-to-left, BERT looks at the entire sentence at once. This allows it to understand the full context of a word from both sides.

**5. How Was BERT Trained?**

BERT was pre-trained on a massive amount of text (all of Wikipedia and a huge collection of books) using two clever tasks:

1. **Masked Language Model (MLM):** The model was given sentences where about 15% of the words were hidden (or "masked"). Its only job was to predict what the hidden words were. This forced it to learn deep relationships between words and sentence structure.
2. **Next Sentence Prediction (NSP):** The model was given two sentences, A and B, and had to predict whether sentence B was the actual sentence that followed sentence A in the original text. This taught BERT to understand the relationships between sentences.

**6. How is BERT Used?**

Because it was pre-trained to have such a deep understanding of language, BERT can be easily fine-tuned for many specific tasks, such as:

* **Text Classification:** Determining the sentiment of a review (positive/negative).
* **Question Answering:** Finding the answer to a question within a given paragraph.
* **Sentence Similarity:** Figuring out if two sentences mean the same thing.

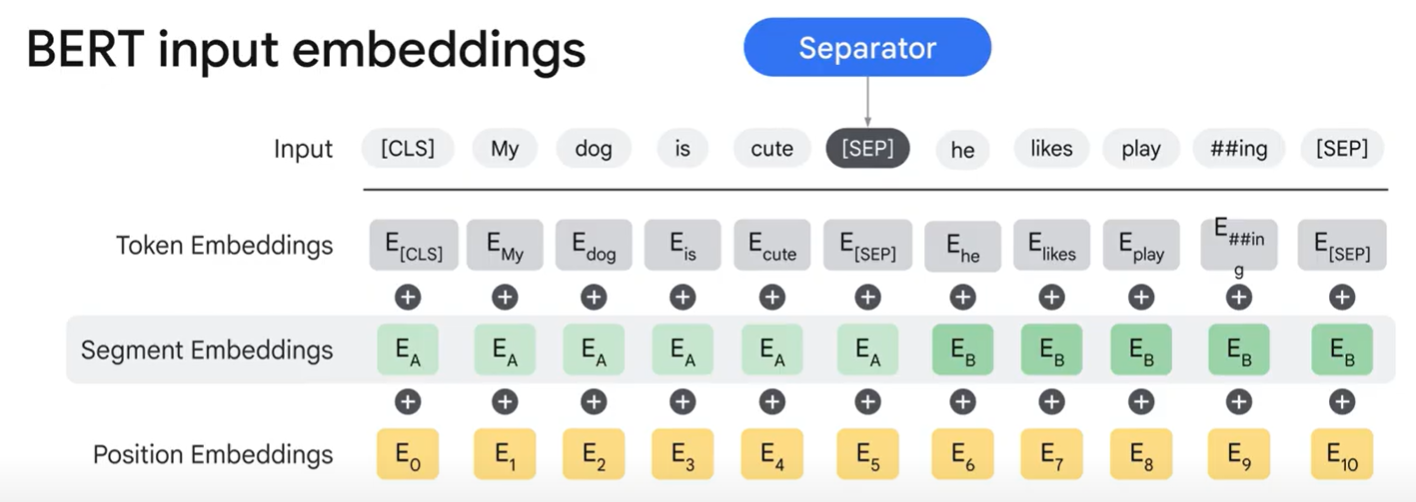
To do this, BERT uses special "embeddings" (numerical representations) that tell it about the words themselves (**token ,.**), which sentence a word belongs to (**segment embeddings**), and the order of the words (**position embeddings**).

For input sentences, we will get three embeddings

**Token 🡪** It is representation of each token as embedding in input sentences. Words are transformed into vector representations

**Segment 🡪** Bert distinguishes two sentences in a given pair and this has special token called SEP which separates two sentences

Position 🡪 It helps in learning the order of sentences



**Vertex AI studio:**

It is a gateway for GenAI in GCP

**3. How to Configure the AI Model**

After writing your prompt, you use the right side of the studio to configure the model:

* **Model Selection:** You can choose from various AI models.
  + **Google Models:** The best choice for most tasks in Vertex AI.
    - **Gemini (Flash or Pro):** The best option for general-purpose tasks and for prompts that include mixed data like text, images, and video (multimodal).
    - **Specialty Models:** Models fine-tuned for specific jobs, like **Imagen** for creating images or **Codey** for writing code.
  + **Third-Party Models:** You can also access models from other companies like Anthropic (Claude) or Meta (Llama).
* **Model Parameters (Controlling Randomness):** These settings adjust how creative or predictable the AI's response is.
  + **Temperature:** A low temperature (e.g., 0.2) makes the AI choose the most obvious, common words, which is good for factual tasks like summarization. A high temperature (e.g., 0.9) allows the AI to be more creative and use less common words, which is good for brainstorming or writing stories.
  + **Top-K:** This tells the model to randomly pick the next word from the "top K" most likely words. For example, if K=2, it will only choose between the two most probable words.
  + **Top-P:** This is a more dynamic approach. It tells the model to pick from the smallest group of words whose combined probability is at least "P". This avoids strange results that can sometimes happen with Top-K.

**Grounding and RAG (Retrieval-Augmented Generation)**

* **The Concept (Grounding):** This is the idea of connecting the AI model to a reliable, external source of information to "fact-check" its responses. It is concept
* **The Implementation (RAG):** This is the technical method used to achieve grounding. When you ask the AI a question, it first retrieves relevant, up-to-date information from a source (like Google Search or your own company's documents) and then uses that information to generate a more accurate and current answer and it is the implementation of Grounding

**Vector Search and embeddings:**

**Vertex AI vector search**

The main difference between Vector Search and traditional search is **understanding meaning (semantic search)**.

* **Traditional Keyword Search:** This method is good at matching exact words. If you search for "summer tops," it will find "summer tops." However, it won't understand that you might also be interested in "swimming suits" or other beachwear because it doesn't grasp the context.
* **Vector Search (Semantic Search):** This method focuses on the underlying meaning. It can find results that are similar in concept to your query, even if they don't use the exact same words. It understands that "attire for a beach party" is related to "summer tops."

**2. The Main Benefits of Vector Search**

1. **Semantic Understanding:** It finds results that are similar in meaning, which is great for natural, everyday language queries.
2. **Multimodal Search:** It works with different types of data, not just text. You can use it to search with images, audio, and more. For example, you could search for products that look visually similar to a photo you took.
3. **Personalization:** By understanding context, it can provide highly personalized search results and recommendations that are more relevant to you.
4. **Powering Generative AI:** It's a critical component for modern AI applications (like large language models) because it helps them retrieve information quickly and efficiently.

**3. How Does Vector Search Work?**

The process can be broken down into three main steps:

* **Step 1: Encode Data into Vectors:**
  + An AI model called an **Embedding Model** is used to convert all your data (text, images, etc.) into a numerical representation called a **vector**. This vector captures the semantic meaning of the data.
  + *Analogy:* This is like reading a book and creating a list of all the important keywords and concepts it contains.
* **Step 2: Create an Index:**
  + All these vectors are organized into a special index. This index is built in a way that makes searching through millions or billions of items incredibly fast.
  + *Analogy:* This is like creating a book index where you pair each keyword with the page numbers where it appears.
* **Step 3: Search the Vector Space:**
  + When you make a search query, your query is also converted into a vector.
  + The system then searches the index to find the vectors that are "closest" or most similar to your query's vector. These are your search results.
  + *Analogy:* This is like looking up your term in the book's index (e.g., alphabetically) to quickly find the right pages.

**4. The Two Big Challenges**

To make this work, you have to solve two main problems:

* + 1. **How to Encode:** How do you effectively convert different types of data (text, images) into numerical vectors that truly capture their meaning? This is done using **Embeddings**.
    2. **How to Index and Search:** How do you build a search system that can find the most similar items among billions of vectors almost instantly? This is the job of the **Vector Search** technology itself.

A diagram of a search engine

Description automatically generated

How does vector search works

A screenshot of a diagram

Description automatically generated

A screenshot of a computer

Description automatically generated

**Embedding :**

Computers don't understand words; they only understand numbers. The challenge is to convert text into numbers in a way that preserves the original *meaning* of the words. This process needs to solve two main problems:

1. The numbers must reflect the relationships between words (e.g., the numbers for "king" and "queen" should be similar).
2. The numerical format must be efficient for AI models to use (it should be "dense," not "sparse").

**2. A Simple (But Flawed) Approach: One-Hot Encoding**

An early and intuitive method for turning words into numbers is **one-hot encoding**.

* **How it works:**
  1. First, you create a list of every single unique word you have, called a "vocabulary." This list can have tens of thousands of words.
  2. For each word, you create a long vector (a list of numbers) that is the same size as your vocabulary.
  3. This vector is filled with zeros, except for a single "1" at the position that corresponds to that specific word.
* **Example:** If your vocabulary is [dog, chase, person, my, cat, run], the word "dog" would be represented as [1, 0, 0, 0, 0, 0].
* **The Problems (Why it's not great):**
  1. **No Meaning (No Semantic Relationship):** The vectors for "dog" and "cat" are just as different from each other as the vectors for "dog" and "run." The model can't tell that dogs and cats are both animals.
  2. **Inefficient (Sparse and High-Dimensional):** If your vocabulary has 20,000 words, each word becomes a vector with 20,000 numbers, where 19,999 of them are zero. This is called a **sparse embedding**. It's computationally wasteful and can cause AI models to perform poorly.

**3. A Better Approach: Text Embeddings (Dense Embeddings)**

To solve the problems of one-hot encoding, a more advanced technique called **text embedding** is used.

* **The Core Idea:** Instead of a long list of zeros and one "1," each word is represented by a shorter, "dense" vector of meaningful numbers (e.g., 300 numbers instead of 20,000). Each number in this vector represents a different abstract "feature" or "dimension" of the word's meaning.
* **The Magic of Embeddings:**
  + **They Capture Meaning:** Words with similar meanings will have similar vectors. The vectors for "king" and "queen" will be very close to each other in this "vector space," while the vector for "apple" will be far away.
  + **They Capture Relationships:** The vectors are so well-structured that you can do math with them. The most famous example is: vector('king') - vector('man') + vector('woman') results in a vector that is extremely close to vector('queen').
* **Dense vs. Sparse:** Because these vectors are shorter and filled with useful values (not mostly zeros), they are called **dense embeddings**. They are much more efficient and powerful for AI models.

**4. How Are Embeddings Created?**

You don't have to manually figure out these vector numbers. Instead, a **neural network** is trained on a massive amount of text (like all of Wikipedia). By processing this text, the network learns the relationships between words and automatically generates the dense embedding vectors for them.

* **Famous Models:** Popular algorithms for this include Word2Vec (by Google), GloVe (by Stanford), and FastText (by Meta).
* **Using Pre-trained Models:** You don't need to train these models yourself. Companies like Google provide powerful, **pre-trained embedding models** that you can easily use through an API. You just send your text to the API, and it sends back the dense vector embedding.

**Vector Search and indexing :**

**1. The Two Big Challenges After Creating Embeddings**

Once you have your data (text, images, etc.) turned into numerical vectors (embeddings), you face two main problems:

1. **How do you measure the distance between vectors?** In a complex, multi-dimensional space, how can you mathematically determine that the vectors for "king" and "queen" are closer to each other than they are to "apple"?
2. **How do you search for vectors quickly?** How can you find the most similar items in a database of billions of vectors without it taking forever?

**2. Part 1: How to Measure the Distance Between Vectors**

The text describes four common ways (metrics) to calculate the "distance" or "similarity" between two vectors:

* **Manhattan Distance (L1):** Imagine you're in a city with a grid-like street layout. This metric calculates the distance by only moving along the grid lines (like walking city blocks), not in a straight line. It's the sum of the absolute differences of the coordinates.
* **Euclidean Distance (L2):** This is the most intuitive metric. It measures the shortest, straight-line distance between two points ("as the crow flies").
* **Cosine Distance:** This metric doesn't care about the length of the vectors, only their **direction**. It measures the angle between them. If two vectors point in the exact same direction, they are considered identical. This is very useful for comparing the meaning of text.
* **Dot Product Distance:** This metric is a mix of the others. It considers both the **direction and the magnitude (length)** of the vectors when calculating similarity.

**3. Part 2: How to Search for Vectors Efficiently**

Once you can measure distance, you need a way to search. There are two main approaches:

* **Brute Force (The Slow Way):** This method is perfectly accurate but very slow. It works by taking your search query, calculating its distance to **every single item** in the database, sorting all those distances, and then returning the closest ones. This is impractical for databases with millions or billions of items.
* **Approximate Nearest Neighbor (ANN) (The Fast Way):** This is a much smarter approach used in production systems. The core idea is to trade a tiny bit of accuracy for a massive increase in speed. Instead of checking every single item, it uses clever algorithms to quickly narrow down the search to a small, relevant area.

**4. ScaNN: The "Secret Sauce" Behind Google Search**

The text introduces **ScaNN** (Scalable Approximate Nearest Neighbor), a state-of-the-art ANN algorithm developed by Google Research. It's the technology that powers services like Google Search and YouTube. ScaNN is so fast and accurate because it combines three powerful techniques:

1. **Space Pruning (Reducing the Search Area):**
   * ScaNN first organizes the entire vector space into a hierarchical tree structure, like a family tree.
   * When a search starts, it quickly navigates this tree, "pruning" (cutting off) entire branches that are irrelevant to the query.
   * This allows it to ignore huge portions of the database and only search within the most promising sub-sections.
2. **Data Quantization (Compressing the Vectors):**
   * To save space and speed up calculations, ScaNN compresses the large numerical vectors into a much smaller format. For example, it can shrink a vector of nine floating-point numbers down to just 12 bits.
3. **Filtering with Business Logic:**
   * ScaNN allows you to apply filters to restrict the search to only the data you care about. For example, you could search for "dresses" but filter the results to only show ones that are "red" and available in the "United States."

**5. Vector Search on Google Cloud**

The lesson concludes by mentioning that **Vertex AI Vector Search** (formerly known as Matching Engine) is a fully managed service that uses an even more advanced version of ScaNN. It provides a powerful, scalable, and cost-effective way for anyone to build applications with this cutting-edge similarity search technology.

**Vector AI Vector Search:**

**What is Vertex AI Vector Search?**

It's a fully managed service on Google's AI platform (Vertex AI) that makes it easy to create search engines that understand the *meaning* of a query, not just the keywords. This is also known as **semantic search**. The service brings together all the necessary Google Cloud tools, like:

* **BigQuery:** For storing your data.
* **Embedding APIs:** To convert your data into a numerical format (embeddings) that AI can understand.
* **MLOps:** Tools to deploy and manage your AI application.

**2. How Can You Use It?**

You can interact with Vertex AI Vector Search in three main ways:

1. **No-Code (UI):** Use a simple, graphical user interface in the Google Cloud console. This is great for experimenting and monitoring.
2. **Code-Based (APIs & Notebooks):** Write code (e.g., in Python) to build a more robust and automated search pipeline. This is ideal for production systems.
3. **Minimal Code (Command Line):** Use the gcloud command-line tool for some basic tasks.

The lesson focuses on the first two methods.

**3. How to Build a Search Engine (No-Code UI Walkthrough)**

The lesson walks through the steps of creating a search "index" (the core of the search engine) using the user interface:

1. **Basic Setup:** Give your index a name, description, and choose a region.
2. **Provide Your Data:** You need to tell the system where your data's numerical representations (vectors) are stored. These vectors must be in a file on Google Cloud Storage.
3. **Choose a Search Algorithm:**
   1. **Tree-AH (Default):** This is a fast and scalable algorithm based on **Approximate Nearest Neighbor (ANN)**. It's perfect for production because it can search through huge datasets very quickly.
   2. **Brute Force:** This is a simpler, slower method that checks every single item. It's useful for testing on small datasets during development.
4. **Configure the Index:**
   1. Tell the system the **dimensions** of your vectors (e.g., 768 numbers per vector).
   2. Set the **distance measure** to define how "similarity" is calculated. The default and recommended option is **Dot Product**.
   3. You can also configure things like **shard size** (how the data is split up for faster processing).
5. **Create and Deploy:** After you click "Create," the index is built. Once it's ready, you deploy it to an "endpoint," which is a live address that can receive search queries.

**How to Build a Search Engine (Code-Based Example)**

The lesson also shows how a data scientist would build this using code, for example, to create a "similar items" feature for an e-commerce site.

1. **Generate Embeddings:** Start with your product data in a database like **BigQuery**. Use a simple function (ml.generate\_embedding) to convert the text description of each item (e.g., "blue cotton pants") into a dense vector.
2. **Export Embeddings:** Save these generated vectors into a JSON file and upload it to Google Cloud Storage.
3. **Create and**
4. **Deploy via API:** Write code that calls the Vector Search API. In your code, you specify the algorithm, the location of your JSON file, the vector dimensions, and other settings. The code then creates the index and deploys it to an endpoint.
5. **Run a Query:** Your search engine is now ready. When a user searches for an item, your application sends the query to the endpoint. The Vector Search engine finds the most semantically similar items and returns the results in milliseconds.



**Hybrid Search**

Hybrid search is the "best of both worlds" because it merges two search methods that work in opposite ways:

* **Semantic Search:** This is the "smart" search that understands the *meaning* and context of your query. It's great for vague or conversational questions. For example, it knows that "clothes for a beach party" is related to "summer shorts" and "sunglasses."
* **Keyword Search:** This is the traditional, literal search that finds *exact word matches*. It's perfect for finding specific terms like a product name ("Chrome Dino Pin") or a barcode.

By combining them, you get a search engine that is both insightful and precise.

**2. Why Do We Need Both?**

Using just one type of search has its weaknesses:

* **Problem with Semantic Search Alone:** It can fail with new or very specific terms that its AI model wasn't trained on. For example, it might not know what a brand-new product name is.
* **Problem with Keyword Search Alone:** It's not very smart. If you search for "attire for cold weather," it won't find a "winter jacket" unless those exact words are in the description.

Hybrid search solves this by using semantic search for the general meaning and keyword search to catch the specific, important terms.

**3. How Does Hybrid Search Work?**

The process involves running two searches in parallel and then cleverly merging the results.

**Process 1: The Keyword Search Path**

1. **Encoding (Creating Sparse Embeddings):**
   * The text is broken down into individual words or "tokens."
   * It's then converted into a **sparse embedding**. This is a very long list of numbers (a vector) that corresponds to a huge vocabulary. Most of the numbers are zero.
   * Instead of just putting a "1" if a word is present, it often uses a weighting algorithm like **TF-IDF** (Term Frequency-Inverse Document Frequency). This algorithm gives a higher score to words that are important to a specific document but not common across all documents.
2. **Indexing and Searching:** The system organizes these sparse embeddings to quickly find documents with similar keyword patterns.

**Process 2: The Semantic Search Path**

1. **Encoding (Creating Dense Embeddings):**
   * The text is converted into a **dense embedding**. This is a much shorter list of meaningful numbers that captures the overall context and meaning of the text.
2. **Indexing and Searching:** The system organizes these dense embeddings to find documents with a similar meaning.

**4. The Final, Crucial Step: Merging the Results**

After running both searches, you have two different ranked lists of results. To combine them into a single, better list, a sophisticated algorithm called **Reciprocal Rank Fusion (RRF)** is used.

1. **How RRF Works:** Instead of just adding scores, RRF gives a big boost to items that rank highly on *either* list. An item that is #1 on the keyword list gets a lot of points, as does an item that is #1 on the semantic list. This ensures that items that are a perfect match for either meaning or keywords are prioritized in the final results.

**5. How Vertex AI Makes This Easy**

Building a hybrid search engine used to be very difficult. You had to manage two separate search systems and write complex code to merge the results.

**Vertex AI Vector Search** simplifies this entire process. It abstracts away the complexity, making it feel like you are just doing one search. You provide both the dense (semantic) and sparse (keyword) embeddings for your data in a single file. Vertex AI handles the separate indexing, parallel searching, and the RRF re-ranking for you automatically.

**Example from the text:** A hybrid search for "kids" returns "Google Blue **Kids** Sunglasses" at the top because it's a perfect match for both the keyword and the meaning. It also returns "Google White Classic **Youth** Tee" a bit lower down. This item doesn't contain the word "kids" (so it wouldn't be found by a simple keyword search), but the semantic search understands that "youth" is very similar in meaning to "kids."

**BQ ML**

You can build and run AI models using familiar SQL commands or Python. It supports two main types of AI tasks:

1. **Predictive AI:** This involves making predictions based on data.
   * **Examples:** Forecasting future sales, classifying emails as "spam" or "not spam," predicting house prices, or grouping customers into segments.
   * **Models Used:** Regression, Classification, Clustering, and Time Series (like ARIMA\_PLUS).
2. **Generative AI (Gen AI):** This involves creating new content.
   * **Examples:** Generating summaries of articles, creating marketing text, analyzing images, or translating languages.
   * **Models Used:** **Gemini** (Google's most advanced model that can work with text, images, audio, and more) and other Cloud AI services for specific tasks like language translation.

**The Workflow: How to Build an AI Model**

The process is broken down into two main stages: **Create** and **Use**.

**Stage 1: Create the Model**

This is where you build and train your model using the CREATE MODEL SQL statement. It's a cycle with three steps:

1. **Data Preparation:** Get your data ready. This includes cleaning and preparing structured data (like tables) and unstructured data (like text or images).
2. **Model Creation:** Train a built-in BigQuery model or connect to a powerful remote model like Gemini.
3. **Model Evaluation:** Check how well your model is performing using the ML.EVALUATE function. If it's not good enough, you can go back, adjust your data or settings, and train it again.

**Types of Models You Can Create**

BigQuery supports three main types of models based on where they are hosted:

1. **Local Models:** These models live and are trained inside BigQuery. They are great for predictive tasks like classification (LOGISTIC\_REGRESSION) and regression (LINEAR\_REGRESSION). It's recommended to start with these simpler models to get a baseline before trying more complex ones.
2. **Remote Models:** These models are hosted outside BigQuery (in Vertex AI) but can be used from within BigQuery. This includes powerful pre-trained models like **Gemini** and other Cloud AI services. To use them, you set up a connection and point to the model's endpoint (its access URL).
3. **Imported Models:** These are models you've trained somewhere else (using tools like TensorFlow or XGBoost) and then imported into BigQuery from Cloud Storage.

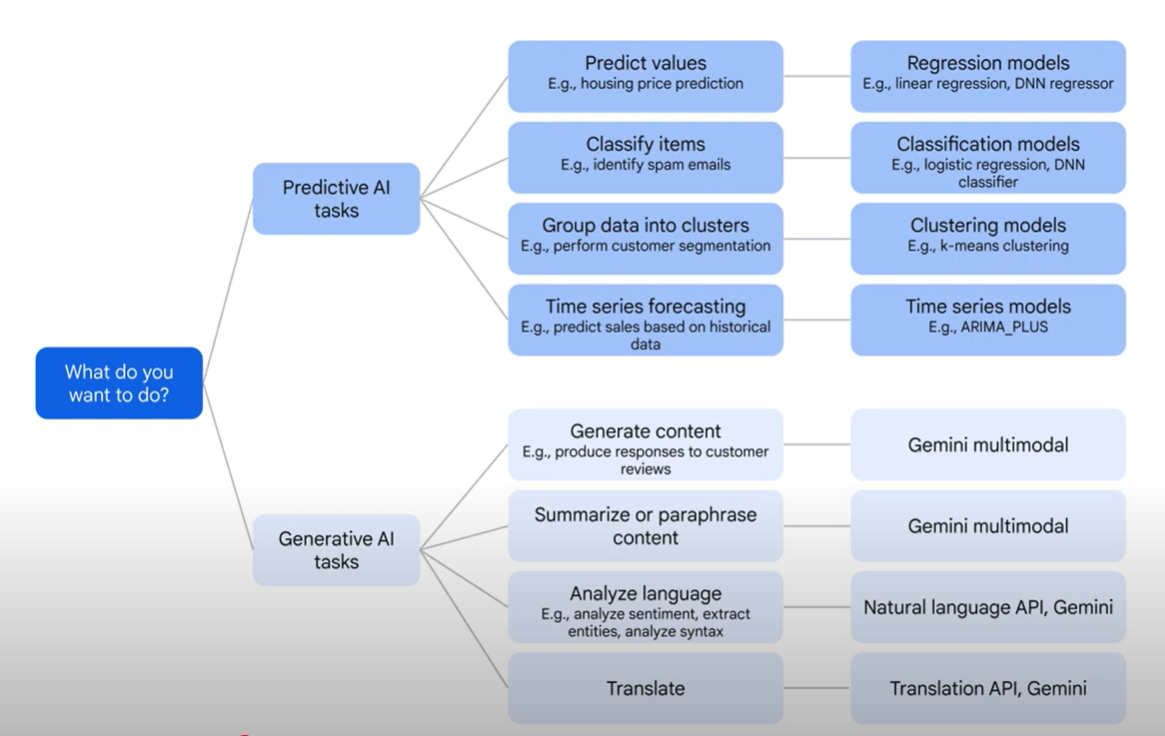
**Stage 2: Use the Model**

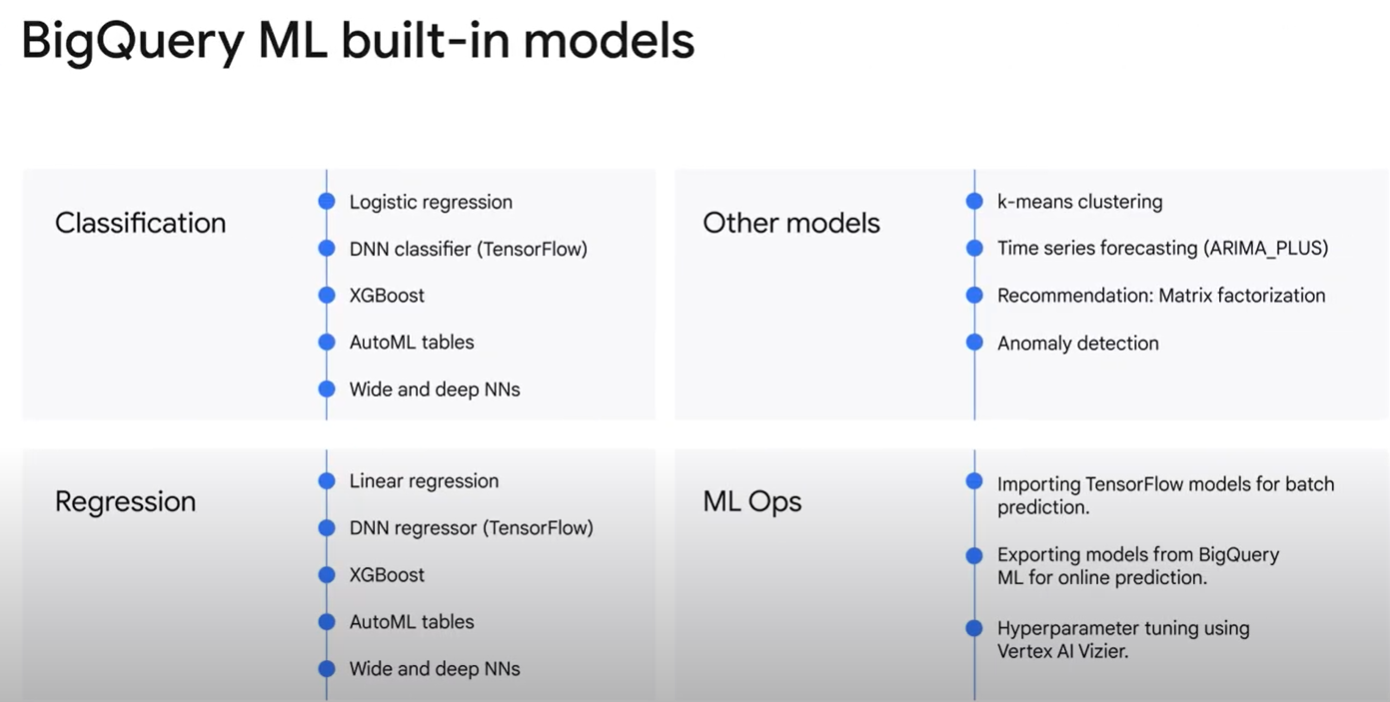
Once your model is created, you can put it to work. This stage also has three steps:

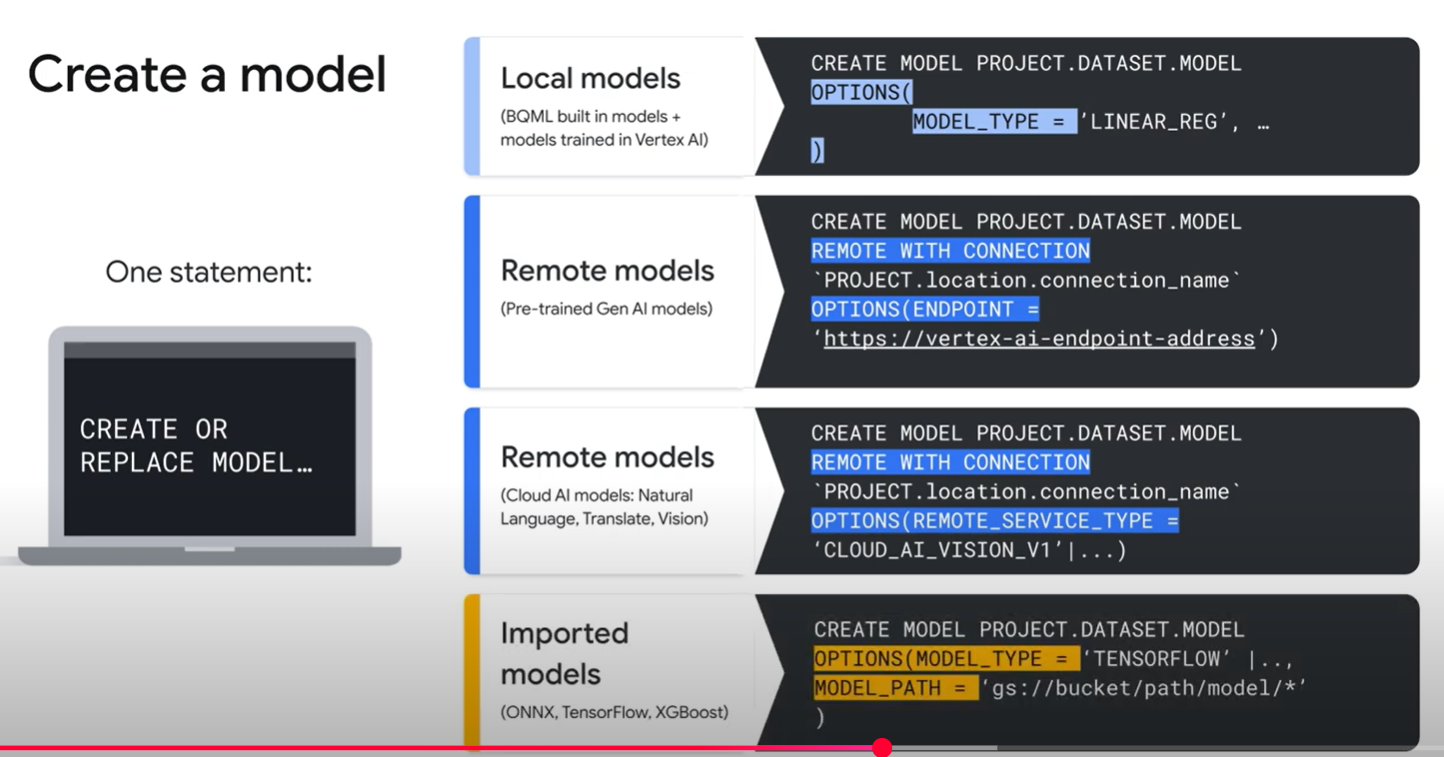
1. **Model Serving (Getting Results):** This is the main goal. You use different SQL functions depending on your task:
   * ML.PREDICT(): For predictive tasks like classification or regression.
   * ML.FORECAST(): For time-series forecasting.
   * ML.GENERATE\_TEXT(): For generative tasks like summarization with models like Gemini.
   * ML.UNDERSTAND\_TEXT() and ML.TRANSLATE(): For specific language tasks.
2. **Model Explanation (Optional):** This helps you understand *why* the model made a certain prediction by showing how much each feature contributed. This is only for predictive models and uses functions like ML.EXPLAIN\_PREDICT().
3. **Model Monitoring:** This involves watching your model's performance over time to ensure it remains accurate. You can check for:

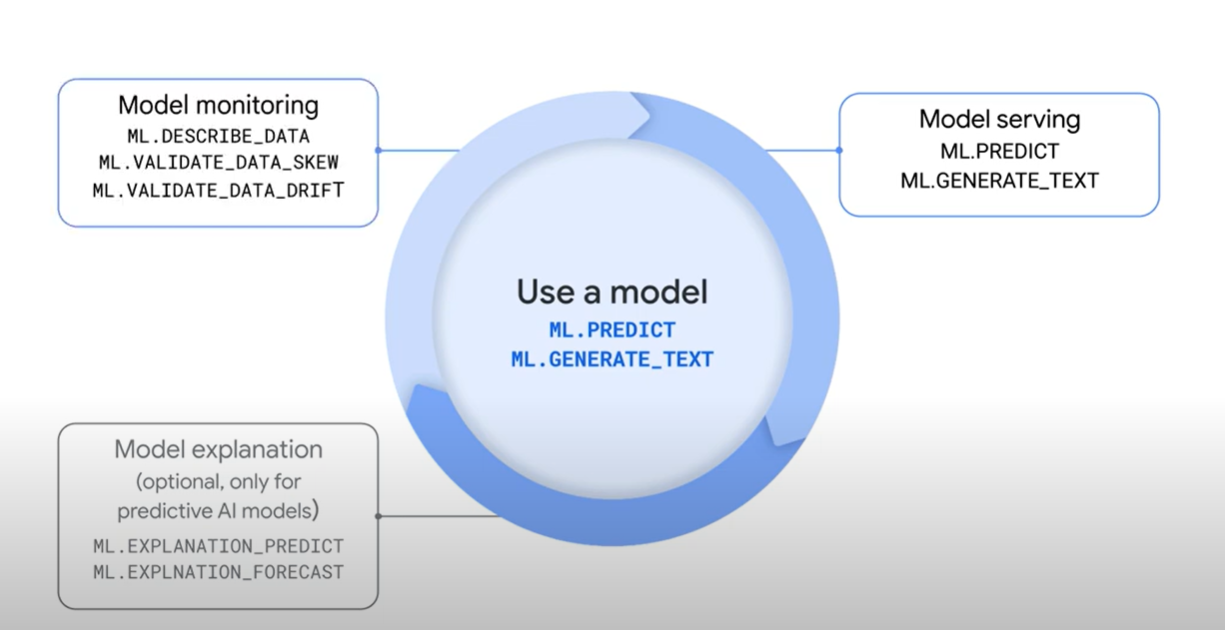
* **Data Skew:** Differences between the data used for training and the data used for predictions.
* **Data Drift:** Changes in the new data over time compared to older data.
* Functions like ML.VALIDATE\_DATA\_SKEW and ML.VALIDATE\_DATA\_DRIFT are used for this.

Finally, the text includes a quick quiz to check your understanding of the three model types (Local, Remote, and Imported) before moving on to a practical example.



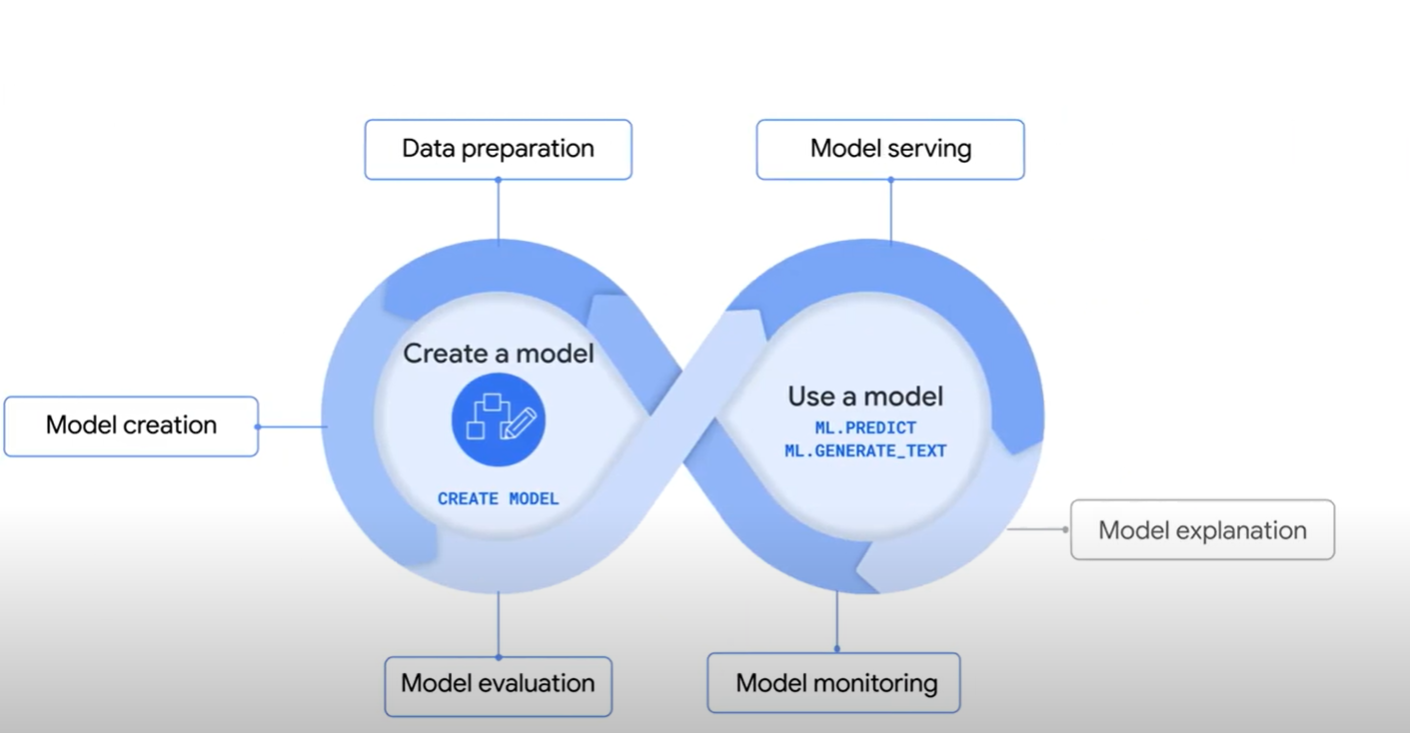


****



A diagram of a company

Description automatically generated with medium confidence



**Implementation**

**Summary: Analyzing Customer Reviews with Gemini in BigQuery**

This document explains how a data analyst can use Google's BigQuery to analyze customer feedback, including text and images, by using the powerful AI model, Gemini. The entire process is done using SQL, the standard language for databases.

**The Goal**

The main goal is to help a fictional company, "Coffee on Wheels," understand its customer reviews. This involves extracting keywords, determining if a review is positive or negative, and even analyzing images customers might have shared.

**The Workspace: BigQuery Studio**

This is the user interface where all the work happens. It has three main parts:

* **Menu (Left):** Your command center for different actions.
* **Explorer (Middle):** Like a file cabinet, it organizes your projects, saved queries, datasets, and AI models.
* **Workspace (Right):** The main area where you write SQL code, create Python notebooks, and do your analysis.

**The 5-Step Technical Plan**

The process is broken down into a five-step pipeline to connect BigQuery with Gemini and analyze the data.

**Step 1: Connect to the AI Model**

* **What it is:** Gemini is a powerful AI model hosted on Google's Vertex AI platform. Because it lives outside of BigQuery, it's called a **remote model**.
* **How to do it:** You must first create a connection from BigQuery to Vertex AI. This tells BigQuery where to find and how to talk to the Gemini model. This can be done easily through the user interface without writing any code. You also need to grant the correct permissions so BigQuery is allowed to use the model.

**Step 2: Prepare Your Data**

Customer feedback can be a mix of text (reviews) and images. BigQuery needs to be able to access this data, which is often stored in Google Cloud Storage.

* **For Text Data:** If your reviews are in a file like a CSV, you can load them into a BigQuery table using the LOAD DATA SQL command.
* **For Image Data (and other files):** You use a special kind of table called an **Object Table**. Think of it as a special index that points to your unstructured files (like images, audio, or videos) in Cloud Storage. It doesn't hold the images themselves, just their location (URI) and metadata. This allows you to use SQL to work with files that aren't in a typical table format.

**Step 3: Create a Reference to the Model**

Once the connection is set up, you use the CREATE MODEL command in SQL. This doesn't build a new AI model from scratch. Instead, it creates a reference inside BigQuery that points to the specific Gemini model you want to use (like gemini-pro for text or gemini-pro-vision for images).

**Step 4: Analyze the Reviews with ML.GENERATE\_TEXT**

This is where the magic happens. You use the ML.GENERATE\_TEXT() function in your SQL query to send tasks to the Gemini model.

* **How it works:** You tell the function which model to use and what data to look at (your table of reviews).
* **The Prompt:** The most important part is the **prompt**, which is the instruction you give to the model. For example:
  + To find keywords: "Extract the main keywords from this review."
  + For sentiment analysis: "Classify this review as 'positive' or 'negative'."
  + For image analysis: "Provide a caption for this image and list any important objects in it."
* **Tuning:** You can also adjust parameters like temperature (for creativity) and top\_p (for word choice) to fine-tune the AI's responses.

**Step 5: Take Action**

After analyzing the feedback, you can use Gemini to take the next steps.

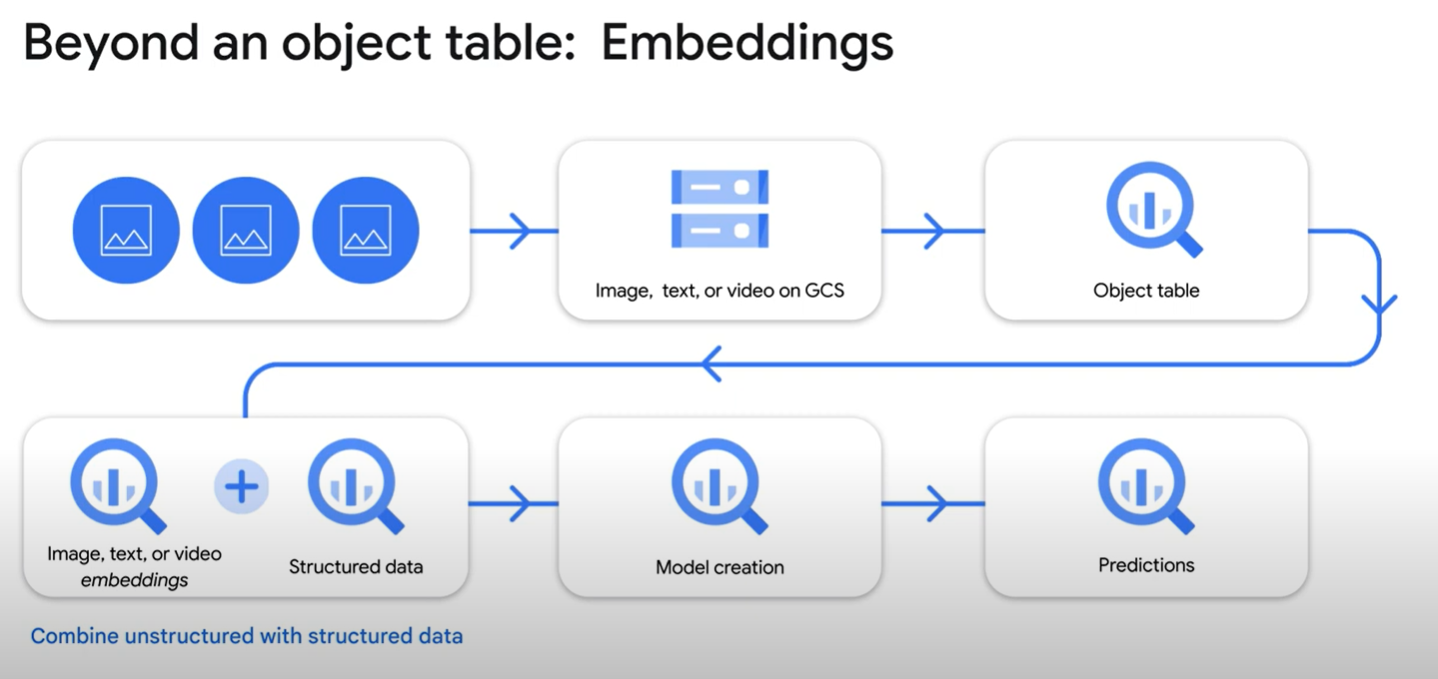
* **Auto-Generate Responses:** You can prompt Gemini to write a polite and relevant response to a customer's review.
* **Prompt Tuning:** To make the responses better and more personalized, you can give Gemini examples. This is called **prompt tuning**:
  + **Zero-shot:** Give an instruction with no examples. (e.g., "Write a response.")
  + **One-shot:** Give one example to guide the model.
  + **Few-shot:** Provide several examples of good and bad reviews and the desired responses.
* **Marketing Strategy:** You can even use prompts to ask Gemini to help brainstorm marketing campaign ideas based on the customer feedback you've analyzed.

Finally, the document ends with a quick quiz question to check understanding: the correct function for using remote AI models like Gemini is ML.GENERATE\_TEXT().

A screenshot of a computer

Description automatically generatedA diagram of a process

Description automatically generated



A screenshot of a computer program

Description automatically generated

**AI Infrastructure: Introduction to AI Hypercomputer**

AI Hypercomputer is an integrated system designed to efficiently scale and deploy AI applications.

* **Layer 1:** High-performance AI hardware (TPUs/GPUs), fast networking, and optimized storage for demanding AI.
* **Layer 2:** Open software (PyTorch, GKE, Kueue) simplifies AI workflows and boosts productivity.
* **Layer 3:** Flexible consumption models (on-demand, spot, CUDs, reservations, DWS) for various AI workloads.

This powerful stack is designed in concert to deliver the highest intelligence per dollar for intensive AI tasks, offering real-world applications such as:

* **Large-scale AI model training**: Leveraging its power for complex model development.
* **Efficient model serving at scale**: Delivering optimal price-performance for widespread AI deployment.
* **AI application development**: Facilitating innovation through the use of open frameworks.

Four main deployment options exist for AI Hypercomputer workloads on Google Cloud:

* **Direct management (GCE)**: Maximum control, high overhead, requires deep infrastructure expertise. This **self-managed** route directly interacts with fundamental compute resourcs like cloud TPU, GPU VM and these can be provisioned using individual VM’s,using Bulk APU for larger sets or employing Managed instance groups (MIG)
* **Foundational (GKE)**: Balances control and automation, ideal for Kubernetes experts.It Is easier team management. It can integrate autoscaling, intelligent job placement,efficent provisioning , queuing/scheduling jobs
* **Open frameworks via Toolkits**: Leverages best practices and simplifies complex setups (e.g., Cluster Toolkit).
* **Fully managed (Vertex AI)**: Easiest to use, Google handles infrastructure, less granular control.

**Vertex AI** provides a comprehensive suite of managed services—encompassing everything from data preparation **and model training (Vertex AI training)** to **deployment (Vertex AI endpoints), MLOps orchestration (Vertex AI Pipelines)**, and access to pre-built solutions and foundation models via the Model Garden.

* With Vertex AI, Google manages the vast majority of the underlying infrastructure and operational complexity. Your team interacts with higher-level APIs and interfaces, focusing primarily on the machine learning aspects of your project.

**Cloud Run**

Cloud Run lets you run your code directly on top of Google’s scalable infrastructure, combining the flexibility of containers with the simplicity of serverless to help boost your productivity.

Any workloads that perform AI inference, especially applications that demand real-time processing, require GPU acceleration to deliver responsive user experiences. Cloud Run, with support for NVIDIA GPUs, can perform on-demand online AI inference using the LLMs of your choice in seconds.

1. **When to choose:** Cloud Run is ideal for event-driven, real-time AI inference workloads where cost-efficiency and auto-scaling are key drivers.
2. **Expertise:** Its serverless nature reduces operational overhead, allowing your team to focus on application logic rather than infrastructure management.

Choosing the right strategy depends on control needs, team expertise, and project goals.

**Consumption options**

**Workload duration:**

How long do you need the resources? Is it a short burst, a fixed period, or an ongoing need?

**Cost sensitivity**

How important is saving money? Are you looking for deep discounts, or is guaranteed capacity more critical?

**Availability needs:**

Do you need immediate, guaranteed access, or is some flexibility in start time acceptable?

**Fault tolerance:**

Can your workload handle interruptions, or does it need to run without any possibility of being stopped unexpectedly?

**Workload predictability**

How predictable is your workload? Is it **steady-state** (consistent usage), **predictably variable** (e.g., higher usage nights and weekends), or **spiky and unpredictable** (sudden, irregular bursts)?

* Future reservations
* Spot 🡪 max saving and highly flexible

**AI Infrastructure: Cloud GPU**

How predictable is your workload? Is it **steady-state** (consistent usage), **predictably variable** (e.g., higher usage nights and weekends), or **spiky and unpredictable** (sudden, irregular bursts)?

**CPU (Central processing unit)**

* **Role in AI:** While not as specialized for deep learning tasks as GPUs or TPUs, CPUs are crucial for AI workloads. They handle general-purpose tasks such as data preprocessing, managing the control flow of machine learning programs, and executing less computationally demanding inference tasks, especially in low-cost scenarios.
* **Core Functionality:** A CPU is a general-purpose processor based on the **von Neumann architecture**. It operates sequentially by loading instructions and data from memory, performing a calculation, and then storing the result back into memory.
* **Primary Advantage (Flexibility):** The greatest strength of a CPU is its immense flexibility. It can execute a vast range of software, from simple applications like word processing to highly complex tasks like controlling rocket engines or processing financial transactions.
* **Primary Limitation (von Neumann Bottleneck):** The sequential nature of its interaction with memory creates a significant performance limitation known as the "von Neumann bottleneck." Because memory access is much slower than the CPU's calculation speed, the overall processing speed is often limited by how quickly data can be fetched from and stored back into memory.

**GPU (Graphical processing unit)**

* **Primary Function and Origin:** GPUs were initially created for rendering graphics in video games. However, their core capability is **massive parallel processing**, which allows them to perform a huge number of calculations simultaneously. This "superpower" makes them exceptionally well-suited for the computationally intensive tasks in deep learning.
* **How They Work:** A GPU's speed comes from its architecture, which includes thousands of **Arithmetic Logic Units (ALUs)** on a single chip. This high volume of ALUs enables the simultaneous execution of thousands of operations, such as the multiplications and additions that are fundamental to complex matrix calculations in neural networks.
* **Strength for AI:** The parallel processing nature of GPUs makes them highly effective for deep learning training. For these tasks, which involve massive parallelism (like matrix operations), GPUs can deliver throughput that is an **order of magnitude higher** than what CPUs can achieve.
* **The Limitations :** Despite their power, GPUs are not perfect. They are still considered **general-purpose processors**, meaning they are designed to handle a variety of different tasks, not just one. This creates a bottleneck:
  + - * + Each of the thousands of ALUs constantly needs to fetch data and store its results in memory. With so many units trying to access memory at once, it can create a traffic jam, slowing things down.
        + While newer generations of GPUs have introduced special optimizations to help with this memory-access problem, the fundamental constraint remains because they must support many different types of software and applications, not just AI.

**The Problem with GPUs for AI**

* Think of a standard GPU (Graphics Processing Unit) as a powerful, general-purpose workshop with thousands of workers (called ALUs). While they are great at handling many different tasks, they have a major bottleneck when it comes to AI.

**The Memory Traffic Jam:** Every single worker constantly needs to go to a central supply room (the memory) to get data and then return to store its results. When thousands of workers try to access the same supply room at once, it creates a massive traffic jam. This constant back-and-forth to memory slows down the entire process.

Even though newer GPUs have tricks to manage this traffic, the fundamental problem remains because they are designed to be jacks-of-all-trades, not specialists for AI.

**TPU (Tensor processing unit)**

A TPU (Tensor Processing Unit) is different. It's a specialized factory designed for one thing: the core math operations used in AI, like matrix multiplication. It solves the memory problem with a clever design called a **systolic array**.

Here’s how it works:

**1. Direct Connections:** Instead of individual workers running back and forth to a supply room, the TPU connects thousands of specialized "multiply-accumulator" units directly to each other in a large grid. It's like an assembly line where each station is physically connected to the next.

**2. Efficient Data Flow:**

* 1. Data is first loaded from the main computer into the TPU's own super-fast memory (HBM).
  2. To perform a calculation, the data is fed into one end of the assembly line (the matrix unit).
  3. As the first worker completes its small piece of math, it doesn't go back to memory. Instead, it **immediately passes its result directly to the next worker** in the line.
  4. This continues down the entire line. Data flows through the grid of processors, with calculations happening at each step, much like water flowing through a series of pipes.

**3. Final Results:** Only when the entire calculation is finished are the final results collected at the end of the assembly line and sent back to the main computer.

**Why GPUs are Essential for AI**

GPUs are like super-fast calculators that can perform a massive number of math problems all at once. This makes them perfect for AI, which relies on complex calculations. They are used in two main phases:

1. **Training (The Learning Phase):** This is where an AI model learns from vast amounts of data. GPUs dramatically speed up this process, reducing the time it takes to build powerful AI models from months to days.
2. **Inference (The Doing Phase):** This is when a trained model is put to work making real-time predictions or decisions (e.g., a chatbot answering a question). GPUs provide the speed needed for instant, responsive results.

**Google Cloud's GPU Options Simplified**

Google Cloud offers different "families" of GPU machines, each designed for specific AI tasks:

* **A4 Family (Newest & Most Powerful):** Built with cutting-edge NVIDIA B200 GPUs, this is the top choice for training and running the largest, most demanding AI models (like foundation models).
* **A3 Family (High-Performance & Scalable):** Featuring powerful NVIDIA H100/H200 GPUs, this family is the expert at training massive models very quickly.
* **A2 Family (Versatile All-Rounder):** Using NVIDIA A100 GPUs, this is a great, balanced option for both demanding training and high-speed inference.
* **G2 Family (Cost-Effective Inference):** With NVIDIA L4 GPUs, this family is optimized to run your trained models efficiently and affordably.
* **N1+GPU (Flexible & Customizable):** This is a general-purpose option that lets you attach a variety of different GPUs (like the T4 or V100). It's a flexible choice when you need a specific balance of computing power and GPU acceleration.

|  |  |
| --- | --- |
| **Workload Type** | **Recommended machine type or series** |
| Pre-training models | A4, A3 Ultra, A3 Mega, A3 High, A2 |
| Fine-tuning models | A4, A3 Ultra, A3 Mega, A3 High, A2 |
| Serving inference | A4, A3 Ultra, A3 Mega, A3 High, A2 |
| Graphics-intensive workloads | G2, N1+T4 |
| High performance computing | For high performance computing workloads, any accelerator-optimized machine series works well.  The best fit depends on the amount of computation that must be offloaded to the GPU |

**GPU cluster provisioning options**

1. **Vertex AI:** A fully managed, all-in-one platform for building and deploying AI models. Use GPUs here to speed up training, predictions, and development.
   1. **Train your own AI models faster.**
   2. U**se powerful, pre-built models** (like large language models or LLMs) from a shared library called the Model Garden
   3. **Get** **faster predictions** from your trained models (this is called "inference"
   4. **Speed up your coding and experiments** in development environments like Vertex AI Workbench
2. **Cluster Director:** Cluster Director is designed for building and managing enormous AI supercomputers. It allows you to treat **up to tens of thousands of GPUs** as a single, unified system.
   1. This is the best choice for the biggest and most demanding jobs, such as training huge foundation models or running large-scale scientific simulations (High-Performance Computing or HPC).
3. **Compute Engine:** Gives you direct control over virtual machines (VMs) with attached GPUs. It's flexible and ideal for graphics-intensive work, simulations, or smaller AI training jobs.
   * + Running applications with a lot of graphics.
     + Performing simulations.
     + Training smaller machine learning models
4. **Cloud Run:** A serverless option where you can run applications that need GPUs (like for AI predictions or video transcoding) without managing the underlying servers.
   * + **Running a trained AI model to make predictions** (inference).
     + **Other** intensive tasks like processing video files or rendering 3D graphics
5. **GKE (Google Kubernetes Engine):** The best choice for running containerized AI applications. It offers powerful and flexible ways to manage GPUs for complex tasks, like serving large open-source AI models efficiently.
   1. **Let GKE automatically manage the GPUs** for your applications.
   2. **Take more control** by manually attaching GPUs to your server nodes.
   3. Use advanced features to **share a single GPU among multiple applications** (called multi-instance GPUs or time-sharing), which helps you save money and get the most out of your hardware.

**Accelerating frameworks**

**1. Getting Max Speed on One Device (like a GPU)**

1. **CUDA:** Think of this as a set of highly-optimized, pre-built instructions specifically for NVIDIA GPUs. When you perform a calculation, the framework sends a fast CUDA "kernel" (a mini-program) to the GPU, which executes it much faster than a regular CPU could.
2. **XLA (Accelerated Linear Algebra):** This is a smart compiler. Instead of running operations one-by-one, it looks at a whole sequence of calculations, optimizes them by combining steps (called "operator fusion"), and then creates a single, super-efficient program to run on the hardware.

**2. Running on Many Different Devices (GPU, TPU, CPU)**

The goal is to write code once and have it run efficiently everywhere.

* 1. **Frameworks (PyTorch/JAX):** They provide a layer of abstraction, so you can write your model without worrying about the specific hardware underneath.
  2. **XLA Compiler:** It acts as a universal translator. It takes your model's computation graph and compiles an optimized version specifically for the target hardware, whether it's a GPU, TPU, or CPU.
  3. **PyTorch/XLA and JAX/XLA:** These are integrations that connect the easy-to-use frameworks with the powerful XLA compiler. This allows developers to write standard Python code and automatically get high performance across different types of hardware without needing to write custom code for each one.

**Optimizing GPU usage**

To avoid wasting time and money on AI training, it's important to measure **Goodput**, which is your *actual, useful training progress* compared to the maximum possible.

Goodput is broken down into three key parts:

**1. Scheduling Goodput: Getting and Keeping Your Resources**

This is all about making sure you have the computers and GPUs you need, for as long as you need them, without any delays.

* **The Goal:** Avoid waiting for resources to become available.
* **How to Maximize It:**
  + **Reserve Resources:** For shorter jobs, book your compute resources in advance to guarantee they are available when you need them.
  + **Use Hot Spares:** Keep some extra, pre-provisioned resources idle and ready. If your job is interrupted, it can immediately switch to a hot spare instead of waiting for a new machine to be scheduled, which significantly improves your scheduling goodput.

**2. Runtime Goodput: Minimizing Downtime**

This measures how much time your model spends doing useful training versus being idle due to interruptions or failures.

* **The Goal:** Complete as many useful training steps as possible and recover from interruptions quickly.
* **Key Factors in Downtime:**
  + **Time Lost (tch):** The training progress lost between your last save (checkpoint) and when a failure happens.
  + **Time to Resume (trm):** The time it takes to get the training job started again after an interruption.
* **How to Maximize It:**
  + **Auto-Checkpointing:** Set up your training job to automatically save its progress right before it's about to be interrupted (for example, due to system maintenance). This makes the amount of lost work (tch) very small.
  + **Rapid Resume:** Preload your large container images and models onto a secondary disk. If a machine fails and restarts, it can access the necessary files almost instantly instead of having to download them all over again. This drastically reduces the time it takes to get back up and running (trm).

**3. Program Goodput: Making Your Code Run Fast on the GPU**

Also known as Model FLOP Utilization (MFU), this is about how efficiently your actual code uses the GPU's processing power.

* **The Goal:** Get the maximum performance out of the hardware.
* **How to Maximize It:**
  + **Custom Kernels:** For very specific and complex math operations (like flash attention), you can write highly-optimized mini-programs (kernels) that run much faster on the GPU than standard code.
  + **Host Offload:** During training, some data (called "activations") is calculated and then needed again later. Instead of recalculating it, you can temporarily move it from the GPU's limited memory to the main computer's memory (DRAM) and bring it back when needed. This saves valuable GPU cycles.
  + **Quantized Training (int8):** For some parts of the calculation, you can use less precise numbers (8-bit integers instead of 32-bit floats). This is much faster for the GPU and can boost efficiency without hurting the model's final accuracy.

**The GPU Decision Tree: Choosing the Right Tool**

To apply these optimizations, you first need to choose the right GPU. The text describes a "decision tree" to help you decide:

1. **First, identify your task:** Are you **training** a new model or **inferencing** (using an already-trained model to make predictions)?
   * **Training** is very demanding and needs powerful GPUs.
   * **Inferencing** is usually less demanding but often needs to be very fast and responsive.
2. **Follow the flowchart:** Based on your task, model size, budget, and performance needs, a flowchart can guide you to the best GPU choice, helping you optimize for either cost, training time, or both.

**TPU in Detail**

At its core, a **TPU (Tensor Processing Unit)** is a special computer chip designed and built by Google. Unlike a general-purpose processor in your laptop (a CPU) or even a graphics card (a GPU), a TPU is an **ASIC (Application-Specific Integrated Circuit)**. This means it was created for one primary purpose: to be incredibly fast and efficient at the mathematical calculations required for Artificial Intelligence (AI) and machine learning.

For code to run on a TPU, it must first be processed by a special compiler called **XLA (Accelerator Linear Algebra)**. The XLA compiler acts as a translator. It takes the high-level instructions from AI frameworks (like PyTorch or JAX) and converts them into low-level machine code that the TPU hardware can execute directly and very quickly.

Here’s when and why you would use them:

**When to Use TPUs**

* **Training Massive AI Models:** They are ideal for training extremely large and complex models, such as the **Large Language Models (LLMs)** that power advanced chatbots and other generative AI. These models involve a staggering number of calculations, which TPUs are designed to handle efficiently.
* **Recommendation Systems:** If you're building a model that gives recommendations (like suggesting movies on a streaming service or products on an e-commerce site), TPUs are a great choice. They have special components called **SparseCores** that are specifically designed to speed up the "embedding" operations common in these types of models.
* **Scientific and Healthcare AI:** TPUs are powerful tools for computationally heavy scientific research. They can accelerate complex simulations and deep learning tasks in fields like healthcare, such as modeling how proteins fold (which is crucial for understanding diseases) and helping in the discovery of new drugs.

**Why Choose TPUs**

* **Cost-Effective:** TPUs provide top-tier performance for demanding AI jobs at a cost that makes large-scale projects more affordable. You get a lot of processing power for your money..
* **Flexible:** You are not locked into a single programming environment. TPUs support popular AI frameworks like **PyTorch** and **JAX**, giving your development team the freedom to use the tools they are most comfortable with.
* **Easy to Manage:** For very large projects, TPUs integrate smoothly with tools like **Google Kubernetes Engine (GKE)**, which helps manage and coordinate many machines working together. Furthermore, if you use **Vertex AI** (Google's all-in-one AI platform), the complexity of managing the underlying hardware is handled for you, letting you focus just on building your model.
* **Advanced Technology:** Newer TPUs (like TPU v4) use a groundbreaking technology called an **Optical Circuit Switch (OCS)** to connect the thousands of chips in a supercomputer. Instead of traditional networking, OCS uses light to create direct, reconfigurable physical connections between chips. This is significantly cheaper, uses much less power, and provides more bandwidth than older technologies like InfiniBand, leading to a faster and more efficient system overall.

**TPU system Architecture**

**The Building Blocks**

* **TPU Chip:** The basic unit is a single TPU chip, which contains a powerful component called an **MXU (Matrix Multiply Unit)**. The MXU is built to perform the massive number of multiplication and addition calculations needed for AI, making it incredibly fast.
* **TPU Cube & Pod:** To get more power, TPUs are physically grouped. A **cube** is a rack of 64 chips, and a **pod** is an even larger collection of TPUs connected by a high-speed network.

**Slices and Multislice: Creating a Supercomputer**

* **Slice:** A "slice" is a flexible portion of a TPU pod that you can use for your AI job. The chips within a slice are connected by an ultra-fast **Inter-Chip Interconnect (ICI)**, allowing them to work together as a single, powerful computer
* **Multislice:** For training truly massive AI models, "Multislice" technology allows you to connect multiple slices together. This lets you scale your job across tens of thousands of chips, far beyond the limits of a single pod. It's designed to be efficient, cost-effective, and easy for developers to use.

**Reliability**

* **ICI Resiliency:** The system has a built-in safety feature that automatically reroutes data if it detects a fault in the connections between chips. This makes the entire TPU system more reliable and ensures your training jobs can continue without interruption.

**TPU Cloud Architecture**

You can use Google's TPU VMs (specialized machines for AI) in a few different ways, depending on your needs.

**How to Use TPUs**

* **Directly:** You can manage the virtual machines (VMs) yourself for maximum control.
* **Managed Services:** For a more streamlined approach, you can use TPUs through other Google Cloud services like **GKE** (for containerized AI workloads) or **Vertex AI** (Google's all-in-one machine learning platform).

**How to Scale Your Work**

TPUs are flexible and can be configured based on the size of your AI job:

* **Single-Host:** Your entire job runs on one TPU machine. This is ideal for smaller models or initial experiments.
* **Multi-Host:** For very large models, your job is distributed across multiple TPU machines that work together. This provides immense power and speeds up training.
* **Sub-Host:** For small tasks, you can use just a portion of the AI chips on a single machine. This is a cost-effective way to run small experiments or development work.

**TPU Options**

Below are different TPU options we have

* **Ironwood:** The newest chip, designed specifically for AI **inference** (running trained models). It is extremely fast, has a large amount of memory, and is very power-efficient.
* **Cloud Trillium (v6e):** The latest generation of all-around AI accelerators. It provides a massive performance boost and double the memory of previous versions, making it ideal for training and serving modern AI models.
* **Cloud TPU v5p:** A high-performance chip for training AI models at a very large scale. Its main feature is a special 3D network that connects thousands of chips together efficiently.
* **Cloud TPU v4:** A powerful and flexible chip that also uses a 3D network for fast communication. It is noted for its unified memory system, which improves performance and coordination between its internal cores.

**TPU Consumption options**

**The Dynamic Workload Scheduler (DWS): A Smart Booking System**

Because TPUs are in high demand, Google has a smart system called the **Dynamic Workload Scheduler (DWS)** to manage who gets them and when. It's like a sophisticated reservation system for a popular restaurant. DWS offers two ways to get resources:

1. **Get in Line (Queued):** You can request TPUs for a specific amount of time (from one minute to seven days). DWS will get them for you as soon as they become available. This is great for shorter jobs or experiments.
2. **Book in Advance (Advanced Reservation):** If you know you'll need TPUs in the future, you can book them ahead of time, just like reserving a hotel room. This guarantees they will be ready for you when you need them.

* **Long-Term Reservations:**
  + **What it is:** You commit to using TPUs for a long period (like a year or more).
  + **Best for:** Large, stable, and long-running AI jobs.
  + **Why:** You get a significant discount and are guaranteed to have the resources when you need them.
* **On-Demand:**
  + **What it is:** The standard, flexible, pay-as-you-go option. You request TPUs and get them if they are available.
  + **Best for:** Urgent jobs, experiments, or any work that cannot be interrupted.
  + **Why:** It's flexible with no long-term commitment, but it costs more.

 **Spot:**

* + **What** **it is:** You use spare TPU resources at a very large discount.
  + **Best for:** Low-priority jobs that can be stopped and restarted without issues (like model pre-training).
  + **Why:** It's the cheapest option, but your job can be "preempted" (stopped) at any time if Google needs the capacity for other users.

**Interoperability between GPU/TPU**

You can set up your AI models (like large language models) to run on both GPUs and TPUs in the same system. This allows you to get the best of both worlds: the high performance of TPUs and the cost-effectiveness of GPUs.

Here’s how it works:

* **Smart Switching:** Using a tool called vLLM and Google Kubernetes Engine (GKE), you can create a setup where your AI workload automatically uses the best available hardware. It can prioritize fast TPUs and then switch to GPUs when needed.
* **Easy to Port:** A demo shows that moving an existing AI model from a GPU to a TPU is very simple. The configuration files are nearly identical, requiring only minor changes.
* **Identical Results:** Even though the underlying hardware is different, the AI model produces the exact same results on both GPUs and TPUs. This is because vLLM automatically handles the necessary conversions.
* **Simplified Operations:** Other configurations for monitoring and scaling your application are also identical for both GPU and TPU setups, making them easy to manage.

**How It Works: The Technology Behind the Magic**

This flexible setup is made possible by a few key technologies working together:

1. **vLLM:** This is a special software library designed to make running LLMs for inference (generating answers) extremely fast and efficient.
2. **GKE (Google Kubernetes Engine):** This is Google's platform for managing applications. It acts as the "brain" that decides where to run your code.
3. **Custom Compute Classes in GKE:** This is a feature that lets you tell GKE the order of preference for your hardware. You can set a rule like: "Always try to run my AI model on a TPU first. If no TPUs are available, then run it on a GPU."
4. **A Dual-Container Pod:** To make the switch seamless, the application is packaged in a clever way. A single "pod" (the basic unit of an application in Kubernetes) contains two separate containers:
   * One container is set up to run the model on a **GPU**.
   * The other container is set up to run the model on a **TPU**.

When GKE places the pod on a machine, only the container that matches the hardware will start up. If it's on a TPU machine, the TPU container runs and the GPU one sleeps. If it's on a GPU machine, the GPU container runs and the TPU one sleeps.

**Best practices for model deployment**

Here are the three key principles:

1. **Focus on Matrix Math:** The core of a TPU is the **MXU (Matrix Multiplier Unit)**, which is built for the heavy matrix calculations common in AI. Ensure your model is dominated by these operations, not simpler tasks like reshaping data, to fully utilize the TPU's power.
2. **Use Fixed Data Shapes:** TPUs analyze and compile your model based on the shape of the first batch of data it sees. If subsequent batches have different shapes, the model will fail because recompiling on the fly is too slow. Always use fixed, consistent tensor shapes.
3. **Avoid Padding:** TPUs work most efficiently when their processing units are completely filled with data. If your data dimensions aren't a multiple of 128, the system automatically adds "padding" (zeros) to make it fit. This wastes processing power and memory. To avoid this, choose tensor dimensions that are multiples of 8 or, ideally, 128.

**Principle 1: Arrange Your Data for Efficiency**

A special program called the **XLA compiler** acts as a translator, converting your model's code into instructions that the TPU hardware can execute efficiently. A key step it performs is **tiling**, where it breaks down very large matrix calculations into smaller, manageable blocks.

* **Hardware Preference:** The TPU hardware is designed to work best with these blocks, which are often sized 128x128 (or 256x256 on the newest TPUs). The TPU's memory also prefers data dimensions that are multiples of 8.
* **The Problem with Reshaping:** If your data isn't arranged in a way that's easy for the compiler to tile, the compiler has to perform **reshape operations** first. These reshapes can create a bottleneck, slowing down your model because they require a lot of memory access, which is slower than pure computation.
* **The Takeaway:** When designing your model, think about how your data is structured. Try to minimize the need for complex reshaping operations, especially right before the main matrix multiplication steps.

**Principle 2: Use Fixed Data Shapes for Predictable Speed**

* The XLA compiler uses a technique called **Just-In-Time (JIT) compilation**. It looks at the very first batch of data your model processes and compiles the entire model based on the *shape* (i.e., the dimensions) of that data. This one-time compilation makes the model run extremely fast on all subsequent batches.
* **The Critical Catch:** If any future batch of data has a different shape than the first one, the model will fail.
* **Why it Fails:** Re-compiling the entire model every time the data shape changes would be incredibly slow. This would completely defeat the purpose of using a high-speed accelerator like a TPU.
* **The Takeaway:** Models where the size of the data can change from one step to the next (known as **dynamic shapes**) are not a good fit for TPUs. You should always design your model so that the data tensors have a fixed, consistent shape from start to finish.

**Principle 3: Avoid Unnecessary Padding**

* To get the absolute best performance, you want your calculations to perfectly fill the TPU's processing units (those 128x128 chunks).
* **What is Padding?** If your data's dimensions don't fit perfectly (for example, if a dimension is 130 instead of a multiple of 128), the XLA compiler will automatically add **padding**. It fills the extra space with zeros to make the data fit into the required block size.
* **The Downsides of Padding:**
  + - **Wasted Power:** When the TPU processes a padded tensor, it's performing calculations on "empty" data (zeros). This wastes valuable processing power.
    - **Increased Memory Usage:** Padding makes your tensors larger, which means they take up more of the TPU's precious on-chip memory. In severe cases, this can even cause your model to run out of memory.

**How to Avoid It:** The best way to prevent padding is to choose your tensor dimensions carefully. For the best results, make sure your dimensions are multiples of 8, and for matrix operations, aim for dimensions that are multiples of **128**. You can use a tool called op\_profile to check how much padding is happening in your model.

**Build and modernize Applications with Gen AI**

Generative AI is a technology that learns from existing information to create brand new content, like text or images.

While it's not new, it has recently become much easier for developers to add to their applications. A traditional app usually just involves the application code and a database. A "Gen AI app" adds a third piece: a powerful AI model (often a Large Language Model, or LLM).

This allows apps to do powerful new things, such as:

* **Generate content:** Automatically write a product description from a picture.
* **Have smarter conversations:** Power a chatbot that remembers your entire conversation history.
* **Improve search:** Understand the *meaning* behind your search query, not just the keywords. For example, it knows a developer searching for "python" wants different results than a zookeeper.

**AI Models: Old vs. New**

* **Traditional AI:** Is like teaching a child one specific task. You show it many pictures of cats and tell it "this is a cat." It gets very good at identifying cats but can't do much else.
* **Foundation Models (like Generative AI):** Are like telling a child to go read an entire library. It learns about cats, dogs, history, science, and millions of other things on its own. Because it has such broad knowledge, you can then ask it to do many different tasks, like write a story, summarize a book, or answer complex questions.

**Should You Build Your Own AI?**

* **Build:** You can create your own AI model from scratch. This gives you total control but is very difficult, expensive, and requires a lot of expertise and data.
* **Consume:** It's much easier to use a powerful, pre-built foundation model from a company like Google. You get the benefits of a model trained on massive amounts of data without the cost and effort.

**Google's Foundation Models**

Google offers several ready-to-use foundation models on its Vertex AI platform, each specializing in different tasks:

* **Gemini:** A powerful, all-around model that can understand and work with text, images, audio, and video to summarize, answer questions, and more.
* **Imagen:** Creates and edits images based on your text descriptions.
* **Chirp:** Converts spoken words to text and text to natural-sounding speech.
* **Codey:** Helps you write, complete, and debug computer code.
* **Embeddings:** This is a more technical but very powerful tool. It converts data (like text or images) into a list of numbers (a "vector"). This numerical representation allows the AI to understand the **meaning and relationship** between different pieces of data. It's used for:

1. **Semantic Search:** Finding results that are conceptually similar, not just matching keywords.
2. **Recommendations:** Suggesting products or content similar to what a user has liked.
3. **Classification:** Grouping similar items together.

* **Other specialized models:** For tasks like translation (Cloud Translation) and healthcare (MedLM).

**Challenges of Using Generative AI**

1. **Building Your Own AI is Expensive:** Creating a custom AI model from scratch is very costly and requires a lot of work to gather data and train the model. This high cost may need to be repeated every time you want to update the model with new information.
2. **Pre-built AI Lacks Specific Data:** Using a pre-built foundation model is easier and cheaper, but it has limitations. It won't know your company's private data, specific user information, or very recent events that happened after it was trained.
3. **Risk of Harmful Content:** Generative AI can sometimes produce unexpected, offensive, or harmful responses. It's crucial to test the AI and follow responsible practices to minimize this risk.
4. **"Hallucinations" (Making Things Up):** The AI can generate responses that sound correct but are factually wrong. These are called "hallucinations" and can be caused by gaps in the AI's training data. You can use techniques like good "prompt design" to help reduce them.

**Prompts**

**What is a Prompt?**

A prompt is simply an instruction or question you give to a Generative AI model to get a response. The quality of your prompt directly impacts the quality of the AI's answer.

**How to Make a Good Prompt**

A good prompt has two key elements:

1. **Content:** Give the AI all the information it needs, including instructions, context, and examples.
2. **Structure:** Organize your information clearly. Using labels, ordering, and delimiters helps the AI understand your request better.

For example, asking "Tell me five best places to visit in New York" gives a general tourist list. But adding a **persona** like "You are a bot for architects" makes the AI provide a list of architecturally significant places.

**Key Components of a Prompt**

You can improve your results by including different components in your prompt:

* **Objective:** State exactly what you want the AI to achieve.
* **Instructions:** Provide clear, step-by-step directions.
* **Persona:** Tell the AI who it should act as (e.g., a math tutor, a travel guide).
* **Constraints:** Set rules for the response (e.g., "Don't give the answer directly").
* **Examples:** Show the AI exactly what a good output looks like.
* **Response Format:** Tell the AI how to format its answer (e.g., as a JSON file, a bulleted list, or a table).

**Prompts Structure :**

**How to Write Better AI Prompts**

Writing a good prompt is the key to getting a good response from an AI. Here are the main tips:

1. **Be Clear and Specific:** Instead of a vague request like "summarize this," give detailed instructions like, "Summarize this in one paragraph, then list the key points as bullets."
2. **Give the AI a Persona:** Tell the AI who to be. For example, "You are a seasoned travel blogger" will give you better travel tips than just asking for recommendations.
3. **Provide Positive Instructions:** It's better to tell the AI what to do ("Recommend a trending movie") rather than what not to do ("Don't ask for my interests").
4. **Use Examples:** Show the AI exactly what you want. For instance, provide an input and a perfect output to guide its response format.
5. **Structure Your Prompt:** Use separators like XML tags (e.g., <INSTRUCTIONS>, <CONTEXT>) to clearly separate different parts of your request. This helps the AI understand the task better.
6. **Adjust the "Temperature":** For creative tasks, use a higher temperature to get more random and interesting results. For factual tasks like coding, use a lower temperature for more predictable and accurate answers.
7. **Ask for Reasoning:** For complex problems, asking the AI to "explain its reasoning" can force it to think more logically and often leads to a better answer.
8. **Check Safety Settings:** Be aware that built-in safety filters can sometimes block responses. You may need to adjust them for your specific use case.

**Prompts Engineering :**

**How to Build and Improve AI Prompts**

1. **Use Prompt Templates:**
   * A prompt is a mix of **static content** (like instructions and persona, which don't change) and **dynamic content** (like a user's question or specific data, which does change).
   * Combining these into a template allows you to reuse the main structure while inserting new information each time.
2. **Prompt Engineering is a Cycle:**
   * Improving your prompts is a process of testing and refining. You design a prompt, test its performance, and then make adjustments to get better results.
3. **Use Tools like Vertex AI Studio:**
   * This is a tool where you can design, test, and customize your prompts for Google's AI models. It helps you experiment with different instructions, files, and settings to see what works best.
4. **Adjust Parameters for Better Responses:**
   * **Temperature:** Controls creativity. A low temperature gives more predictable, factual answers. A high temperature gives more creative and random responses.
   * **Tokens & Top-K/Top-P:** These settings help control the length and randomness of the AI's output.
5. **Understand the Limitations of AI Models:**
   * **Outdated Data:** The AI's knowledge isn't live; it doesn't know about very recent events.
   * **Limited to Public Data:** It doesn't know your company's private documents or specific user data.
   * **No Source Citing:** The AI can't tell you exactly where it got its information, making it hard to fact-check.

A technique called **Retrieval-Augmented Generation (RAG)** is mentioned as a way to solve these problems by connecting the AI to live, private data.

**How to make foundational AI models more accurate:**

Foundation models are trained for general purposes, but you can improve their performance for your specific needs in a few ways:

**1. Model Tuning**

* **What it is:** Think of this as sending the AI to a specialized training class. You teach it a new, specific skill by showing it hundreds of labeled examples. For instance, you could teach it to always respond in a certain format or to understand your company's unique jargon.
* **How it works:** You provide a dataset of examples, and the model learns new "parameters" to get better at that one task.
* **Pros:** The model becomes much more accurate for the specific skill it was trained on.
* **Cons:**
  + 1. It can be expensive to prepare the data and run the tuning process.
    2. The model is **static**. Once it's tuned, it doesn't learn any new information. If your data changes frequently, this might not be the best solution.

**2. Grounding (Connecting the AI to Real Data)**

Grounding is a technique used to prevent the AI from making things up (hallucinating) by forcing it to base its answers on real, verifiable information.

* **A. Grounding with Google Search**
  + **What it is:** This connects the AI to the live internet. It's perfect when you need answers about current events, a wide range of topics, or general world knowledge.
  + **How it works:**
    1. Your app sends a question to the AI.
    2. The AI uses Google Search to find relevant, up-to-date information.
    3. It then generates an answer based on what it found and includes **citations** (links to the websites it used), making the response more trustworthy.
* **B. Grounding with Your Own Data (using Vertex AI Search)**
  + **What it is:** This connects the AI to your company's private information, like internal documents, product catalogs, or HR policies.
  + **How it works:** You create a secure "data store" in Vertex AI Search containing your proprietary data. When a user asks a question like, "What is our company's time-off policy?", the AI will search *your* data store to find the correct answer instead of giving a generic response.
  + **Types of Data You Can Use:** You can feed it website content, structured data (like database tables), unstructured files (like PDFs and text documents), and more.

**3. Retrieval-Augmented Generation (RAG)**

* **What it is:** This is a powerful and flexible method where you don't change the model at all. Instead, you change the prompt. It's like giving the AI "cheat notes" with the exact information it needs to answer a question.
* **How it works:**
  1. When a user asks a question, your system first retrieves relevant information from your private database or knowledge base.
  2. This retrieved information is then added directly into the prompt you send to the AI.
  3. The AI uses this extra context to generate a highly accurate and relevant answer.
* **Pros:**
  1. It requires no expensive model training.
  2. It allows the AI to use real-time, private, or sensitive user data safely for a single request without that data ever becoming part of the model itself.

**Retrieval Augmented Generation (RAG)**

RAG is a technique to make AI models smarter and more accurate by giving them access to information they weren't originally trained on.

**The Problem RAG Solves**

AI models have two main limitations:

1. **They don't know your private data:** A standard AI model is trained on public internet data. It has no idea about your company's internal documents, product manuals, or a specific user's chat history.
2. **Their knowledge is not always current:** The AI's knowledge is frozen at the time it was trained, so it doesn't know about very recent events.

You can't just retrain the model with this new data every time because it's expensive, slow, and you wouldn't want to put sensitive user data into a shared model.

**How RAG Works in 3 Simple Steps**

**Step 1: Data Retrieval** When you ask a question, the system first searches for relevant information from a trusted source. This source could be:

* **User Data:** Information specific to the person asking, like their account details or past conversations. This is usually easy to find using a user ID.
* **Proprietary Data:** Your company's private knowledge base, like thousands of support documents or product specs.

The biggest challenge here is figuring out which of the thousands of documents are the most relevant to the user's natural language question.

**Step 2: Augment the Prompt** The information found in Step 1 is then added directly into the prompt that is sent to the AI. The prompt now contains:

* The user's original question.
* The retrieved data ("the cheat notes").
* An instruction telling the AI to use this new data to form its answer.

**Step 3: Generate the Response** The AI model receives this "augmented" (enhanced) prompt. It uses the extra context it was just given to generate a much more accurate, relevant, and trustworthy answer. The AI can even be asked to provide links to the documents it used, so the user can see the source of the information.

**Retrieving relevant data**

**The Problem: Keyword Search Isn't Smart Enough**

Traditional search matches exact words (keyword search). If you search for "vacations south of the equator," it might just look for those exact words. It doesn't understand that "south of the equator" means the southern hemisphere.

**The Solution: Semantic Search and Vectors**

**Semantic search** understands the *meaning* behind your words, not just the words themselves. It's powered by a technique called **vector search**.

Here's how it works:

1. **Turn Words into Numbers (Vectors):** An AI model, called an "embedding model," converts your data (text, images, etc.) into lists of numbers called **vector embeddings**. In these number lists, items with similar meanings have similar numbers. For example, the vector for "teddy bear" would be numerically closer to "stuffed animal" than to "grizzly bear."

**Semantic Search:** This is a smarter type of search that understands the *meaning* behind your words, not just the words themselves. It would correctly interpret "south of the equator" as the southern hemisphere and find relevant vacation spots there.

**How it's done:** This is achieved using a technique called **Vector Search**.

1. **Create a Searchable Index:** All these vectors are organized into a special, high-speed database (like Vertex AI Vector Search). This index allows for incredibly fast searching over billions of items. Once you have all your data converted into vectors, you need to organize it in a special, high-speed database called a **vector index**.

Searching through millions of vectors one-by-one would be too slow. A vector index uses clever algorithms (like Google's ScaNN) to group similar vectors into "partitions" or clusters.

This allows the search to quickly ignore large chunks of irrelevant data, making the process incredibly fast.

1. **Search for Meaning:** When you ask a question, your question is also converted into a vector. The system then searches the index to find the data vectors that are numerically closest to your question's vector, giving you results that are similar in meaning.

When you ask a question, your question is also converted into a vector using the same embedding model.

The system then searches the vector index to find the data vectors that are mathematically closest to your question's vector. These are called the "nearest neighbors."

The search first finds the closest partitions and then looks for the closest individual vectors within those partitions, returning the most semantically similar results.

**How This Connects to RAG (Retrieval-Augmented Generation)**

* + 1. RAG uses this process to make AI chatbots smarter.
    2. **Retrieve:** When you ask a chatbot a question, it first uses vector search to find the most relevant documents from its knowledge base.
    3. **Augment:** It then takes the information it found and stuffs it into the prompt it sends to the main AI model, along with your original question

**RAG Architecture using Vertex**

Traditional search matches exact words (keyword search). If you search for "vacations south of the equator," it might just look for those exact words. It doesn't understand that "south of the equator" means the southern hemisphere.

<https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai>

Here is a detailed breakdown and explanation of the RAG (Retrieval-Augmented Generation) architecture presented in the Google Cloud reference “RAG infrastructure for generative AI using Vertex AI and AlloyDB for PostgreSQL” [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

I’ll walk you through each component, the data flow, important design considerations, trade-offs, and how this fits into a practical system. If you like, I can also draw a simplified block‐diagram for you.

**What is this architecture for?**

This is a reference design for a *RAG-capable generative AI application* on Google Cloud, built using Vertex AI and a PostgreSQL-based vector store (AlloyDB with pgvector extension). The goal is: given user queries, combine a retrieval step (finding relevant context) with generation (LLM) to produce more grounded, accurate outputs. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

It is not about training models from scratch, but about serving an application that uses a foundation/model LLM and augments it with context from your data.

The architecture is composed of three major subsystems plus a shared database layer:

1. **Data ingestion**
2. **Serving (request / response)**
3. **Quality evaluation**
4. **Database / storage** (shared by all subsystems)

Let’s go subsystem by subsystem. A diagram of a cloud

Description automatically generated

**High-level architecture & components**

At a high level:

* External data (documents, files, streams) is ingested and processed, embeddings are created and stored.
* At runtime, a user query is turned into embedding, then used to retrieve relevant chunks from the vector store. These retrieved bits are combined with the user query (as context) to form a prompt, which is sent to an LLM via Vertex AI for generation.
* Optionally, generated responses are passed through quality evaluation, logging, filtering, etc.
* All state, config, and intermediate data is stored in a managed PostgreSQL (AlloyDB) plus logging/analytics in BigQuery etc.

Here is a more detailed view of each.

**Data Ingestion Subsystem**

Purpose: Bring in new or updated external data (documents, structured/unstructured) into the system, preprocess them, create embeddings, chunk them, index them, and store embeddings for later retrieval.

**Steps / Data flow**

1. **Upload / ingestion trigger**
   * Data arrives (e.g. a document uploaded to Cloud Storage).
   * A Pub/Sub notification is raised when new data arrives.  
     [Google Cloud+1](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
2. **Trigger processing via Cloud Run**
   * The Pub/Sub event triggers a Cloud Run job (containerized).
   * The job uses configuration (e.g. chunk size, embedding parameters) stored in AlloyDB.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
3. **Preprocessing / chunking**
   * Use Document AI (or custom logic) to parse, extract, clean, segment into chunks or passages.
   * E.g. splitting paragraphs, removing noise, normalizing text.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
4. **Embedding generation**
   * Use Vertex AI’s Embeddings for Text model to convert each chunk (and possibly metadata) into a high-dimensional embedding vector.
   * Note: It’s important that the same embedding model + parameters are used both here and in inference path so that embeddings are comparable.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
5. **Storing embeddings in vector store**
   * The embeddings (along with associated metadata: chunk id, original text, pointer to full document, etc.) are stored in AlloyDB (PostgreSQL) with the **pgvector** extension enabled, which supports vector search / similarity queries.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
6. **Configuration & prompts ingestion**
   * Also, prompts for quality evaluation (to be used later) may be ingested and stored via the same pipeline.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

So the ingestion pipeline ensures that the knowledge base (vector store) stays up-to-date.

**Serving / Request-Response Subsystem**

This is the “online” path: user queries → generate response.

**Steps / Data flow**

1. **User request**
   * A user sends a natural language query (through a frontend: UI, chatbot, mobile app).  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
2. **Embedding of query**
   * The system transforms the user’s query into an embedding using the same embeddings model + parameters used during ingestion.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
3. **Vector Retrieval (semantic search)**
   * The system searches the vector store (pgvector in AlloyDB) for nearest neighbors / semantically similar embeddings to the query.
   * This yields one or more chunks of content that are semantically relevant to the query.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
4. **Contextual prompt construction**
   * The system constructs a *contextualized prompt* by combining the original user request plus the retrieved text chunks (and possibly metadata or instructions).
   * Usually something like: “Given the following documents, answer the question …” or “Use the context below to help answer.”
5. **Send to LLM inference**
   * The constructed prompt is sent to an LLM inference endpoint (on Vertex AI).
   * The LLM can be a foundation model or a custom (fine-tuned) model.
   * The LLM is constrained (via prompt engineering) to use the provided context, reducing hallucination.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
6. **Logging / metrics / analytics**
   * You can log the request/response cycle in Cloud Logging.
   * Push data to BigQuery for offline analytics / reporting.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
7. **Screening / filtering**
   * Use *responsible AI filters* or other screening (e.g. disallowed content, profanity filters, factuality checks) to verify or sanitize responses.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
8. **Return response to user**
   * After filtering, the final response is returned to the user frontend.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

Thus, the serving subsystem implements the core RAG inference loop: embedding → retrieval → generation.

**Quality Evaluation Subsystem**

Purpose: Continuously evaluate (offline or in near-real-time) how well the system’s generated responses perform (accuracy, relevance, factuality, etc.). This helps monitor, debug, audit, and improve the system.

**Steps / Data flow**

1. **Trigger evaluation job**
   * Pub/Sub triggers a Cloud Run job, likely when new responses, prompts, or triggers occur.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
2. **Pull evaluation prompts**
   * The job fetches “evaluation prompts” from AlloyDB (ingested earlier). These are inputs whose quality you want to check.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
3. **Generate or fetch responses to evaluate**
   * Use the outputs generated by the serving subsystem (for the same prompts) or re-invoke the serving path.
   * The evaluation job then assesses them using metrics like factual accuracy, relevance, coherence, etc.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
4. **Store evaluation results**
   * The evaluation scores, prompts, responses, metadata are stored in BigQuery for subsequent analysis, dashboards, or feedback loops.  
     [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

This allows ongoing monitoring of system performance and helps detect degradation, bias, or errors.

**Database / Storage Layer**

This is the backbone that all subsystems use.

**AlloyDB for PostgreSQL (with pgvector)**

* The primary datastore for embeddings, prompt configurations, metadata, and system configuration.
* It uses the **pgvector extension** to support vector similarity queries (nearest neighbor search).  
  [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* It supports features like high availability, automatic failover, read replicas, cross-region replication, etc.  
  [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* Security: supports encryption-at-rest (Google-managed keys by default), optionally customer-managed keys (CMEK).
* Connections are typically via SSL, and you can use an Auth Proxy connector for IAM-based authorization.  
  [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

**BigQuery**

* Used for analytics, logs, storing request/response data, evaluation results, and building dashboards.  
  [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

**Cloud Storage**

* Stores raw ingested files and documents uploaded by users, which are then processed by the ingestion subsystem.  
  [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

**Pub/Sub**

* Serves as the eventing bus / messaging between ingestion triggers, serving, and evaluation subsystems.  
  [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

**Cloud Logging & Monitoring**

* Central observability tools: logging of operations, metrics, error tracking, alerting.  
  [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

**Alternative / Variant Design Options**

The Google reference also discusses alternate architectural choices. Some of these are:

* **Fully managed vector search**  
  Use Vertex AI’s Vector Search (a specialized managed vector search service) instead of managing vector search yourself via PostgreSQL + pgvector. This can simplify scaling and operational complexity. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* **Open-source tools / self-managed**  
  Deploy RAG-capable generative AI systems via GKE (Kubernetes) + tools like Ray, Hugging Face, LangChain, etc., using Cloud SQL or other vector stores. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* **Graph-based RAG (GraphRAG)**  
  Use a graph database (e.g. with Spanner Graph) to represent relationships and context, combining retrieval with graph traversal. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* **Vertex AI + Vector Search**  
  A variant that uses Vertex AI’s built-in vector search capabilities for more scalability and managed operations. [Google Cloud+1](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

**Design Considerations, Trade-offs, and Best Practices**

Below are important caveats, considerations, and tips when adopting or customizing this architecture.

**Security & Compliance**

* Use customer-managed encryption keys (CMEK) if you need control over encryption keys. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* Use VPC Service Controls to limit data exfiltration from the database (AlloyDB). [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* Default database connections must use SSL; consider using Auth Proxy connectors for stronger IAM-based access controls. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* For Cloud Run containers, enforce container image provenance via Binary Authorization. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* Manage data residency: ensure resources are provisioned in the appropriate region(s). [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

**Reliability / High Availability**

* Cloud Run is regional, and automatically balances across zones; use retry / checkpoint logic in jobs. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* AlloyDB supports HA with primary + standby in multiple zones; use cross-region replication for disaster recovery. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* Use read replicas / read pools to scale read/query traffic without overloading the primary. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

**Cost Optimization**

* Right-size Cloud Run container CPU / memory. Use committed-use discounts if predictable. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* For the database, if you don’t need HA, you can use a basic instance to save cost (though with higher risk). [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* Use object lifecycle rules for Cloud Storage to move older data to archival or cheaper tiers. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* Filter and control logs (exclude or limit unneeded logs) to reduce logging cost. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

**Performance & Scaling**

* Monitor and use Query Insights on AlloyDB to identify slow queries and optimize indexing or query patterns. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* Use read replica pools for scaling read-heavy loads (e.g. retrieval queries). [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* Make sure embedding generation and retrieval latency is low—optimize chunk size, embedding model size, caching, etc.
* Use parallel/composite uploads for large files in Cloud Storage to reduce upload time. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)

**Model / Embedding Consistency & Drift**

* Use the *same* embedding model and hyperparameters for both ingestion and inference/query time. Any mismatch can reduce retrieval quality. [Google Cloud](https://cloud.google.com/architecture/rag-capable-gen-ai-app-using-vertex-ai)
* Monitor for drift in embeddings (e.g. if you change embedding models) and plan re-indexing / re-embedding of the knowledge base.
* Be cautious with prompt construction: avoid embedding too much irrelevant context, limit token budget, and guard against overloading context to mislead generation.

**Freshness, Updates, and Deletions**

* Design ingestion so that updates, deletions, or corrections to source data are reflected in the vector store (e.g. re-embedding or removing embeddings).
* Use idempotent ingestion, versioning, or tombstone markers to manage updates gracefully.
* Consider scheduling full re-indexing or incremental updates based on data volatility.

**Trade-offs**

* **Simplicity vs Scalability**: pgvector in PostgreSQL is simpler to integrate (no separate vector DB), but may hit performance or scaling limits compared to a specialized vector search engine.
* **Cost vs management overhead**: Managed services (Vertex AI’s Vector Search, etc.) reduce operational burden but may cost more or have limits.
* **Latency vs context size**: More retrieved context may help answer better, but increases prompt size → latency or token limits.
* **Freshness vs cost**: Frequent re-ingestion keeps the knowledge base fresh but increases compute / storage overhead.
* **Consistency vs eventual consistency**: Ingestion and serving may miss the very latest changes (e.g. slight lag) — acceptable in many use cases but critical in some (e.g. live updates).

**Summary**

* This architecture couples **vector-based retrieval** with **LLM-based generation** to build a grounded generative AI system.
* The ingestion subsystem ensures that your domain data is processed, embedded, and indexed.
* The serving subsystem takes a user query, retrieves the best context, and feeds it to an LLM to generate a response.
* Quality evaluation is continuously running to monitor responses and maintain / improve system quality.
* The system relies on scalable, managed Google Cloud components: AlloyDB (PostgreSQL with pgvector), Vertex AI, Cloud Run, Pub/Sub, BigQuery, Cloud Storage, Logging/Monitoring.
* The design gives flexibility for alternative choices (e.g. use Vertex AI’s managed vector search, or open-source tools on Kubernetes) depending on scale, control, and cost trade-offs.

If you like, I can simplify this into a block diagram with labels, or map it to how you might build a similar system on AWS / Azure / on-prem. Do you want me to convert it into a more digestible architecture slide?

New ProjectTop of Form

Gemini Overview

**What is Gemini?**

Gemini is the name for Google's family of powerful generative AI models. A key feature of Gemini is that it is **multimodal**, which means the models can understand and work with different types of information at the same time, such as text, images, audio, and video.

Here is a simple breakdown of the different Gemini models mentioned:

* **1.0 Pro:** This is a great general-purpose model for a wide variety of tasks that involve text input and getting a text or code response.
* **1.0 Pro Vision:** This model has all the capabilities of Pro, but it can also "see." You can give it text, images, and videos as input, and it can understand and respond to them.
* **1.5 Pro:** A very advanced model that can process a mix of text, images, audio, video, and even PDF files in a single prompt. Its standout feature is its ability to understand an incredibly large amount of information at once (what's called "long-context understanding").
* **1.5 Flash:** This model is also multimodal like 1.5 Pro, but it's optimized for **speed and efficiency**. It's designed for applications that need to handle many requests quickly and cost-effectively.
* **1.0 Ultra:** This is a top-tier model focused on text. It's built to handle the most complex and difficult tasks that require deep reasoning and problem-solving.
* **1.0 Ultra Vision:** This is the most powerful multimodal model, combining the advanced reasoning of Ultra with the ability to understand text, images, and video inputs together for highly complex challenges.

Vertex AI

Vertex AI is Google's all-in-one platform for building, deploying, and managing machine learning models and AI applications. It allows you to access and customize powerful AI models like Gemini, as well as open-source models, to fit your specific needs.

1. **Easy Access to Powerful AI Models:**
   * You can directly use and test advanced models like Gemini. You can give them prompts using text, images, or even video to see how they respond.
   * You can **customize** these models. This means you can "tune" them with your own data to make them experts in your specific area, which can make them faster, cheaper, and better at their tasks.
2. **An Open and Flexible Platform:**
   * Vertex AI isn't a closed-off system. It supports popular open-source tools and frameworks, so data scientists can use the tools they already know and love.
   * This flexibility helps speed up the training process and makes it simpler to get your models out of the lab and into a real-world application.
3. **MLOps: Managing AI for the Long Term:**
   * **MLOps** (Machine Learning Operations) is like DevOps, but for AI. It's a set of practices to make sure your AI models stay reliable, accurate, and effective over time, even as new data comes in.
   * Vertex AI provides a full suite of MLOps tools to help you monitor your models, get alerts if their performance drops, and understand why they make certain decisions.
4. **Vertex AI Agent Builder:**
   * This is a powerful tool that lets you create your own generative AI "agents" (like a specialized chatbot or assistant) very easily.
   * A key feature is that you can "ground" these agents in your own company's data. This means you can build an AI assistant that knows all about your products, internal documents, or customer history, providing accurate and relevant answers. You can even do this using a simple, no-code interface.

**Main Features**

* **APIs for Developers:** Programmers can easily add the power of models like Gemini into their own applications using APIs (Application Programming Interfaces).
* **Model Garden:** This is like an app store for AI models. It contains a huge collection of models from Google, other companies, and popular open-source options like Gemma and Llama 3.
* **Integrated Notebooks:** Vertex AI offers a coding environment (notebooks) that is directly connected to Google's data warehouse, BigQuery. This creates a single workspace for all your data and AI tasks.
* **No-Code Agent Building:** The Agent Builder includes a visual, click-and-drag console that allows anyone, even non-programmers, to build powerful and customized AI agents.

Flutter

Vertex AI is Google's all-in-one platform for building, deploying, and managing machine learning models and AI applications. It allows you to access and customize powerful AI models like Gemini, as well as open-source models, to fit your specific needs.

Generative AI on vertex AI

Generative AI on Vertex AI is Google's platform for building professional-grade AI applications. It allows you to use powerful models like Gemini, which can understand text, images, audio, and video all at once.

This means you can build and launch AI apps that are:

* **Scalable:** They can handle many users at once without slowing down.
* **Secure:** They come with strong security features suitable for businesses.
* **Fast:** They respond to users with very little delay (low latency).
* **Powerful:** They can use advanced models like Gemini 1.5 Pro, which has a massive "memory" (a 2 million token context window), allowing it to understand and process very large amounts of information at once, like an entire book or hours of video.
* **Flexible:** You aren't limited to just Google's models; you can also use and integrate models from other companies.

**Core Capabilities Explained**

Here are the key things you can do with Generative AI on Vertex AI, explained simply:

1. **Multimodal Processing:** This is the ability of the AI to understand different types of information at the same time. For example, you can give the Gemini model a video with audio and a text question about it, and it can process all of those inputs together to give you a smart answer. It's like having a conversation with someone who can see, hear, and read simultaneously.
2. **Function Calling:** This feature allows you to give the AI model new powers by connecting it to other tools and services. Imagine you ask the AI, "What's the weather in Paris and can you book me a flight there?" The AI itself doesn't know the current weather or how to book flights. With function calling, it can connect to a weather service API to get the forecast and an airline's booking API to perform the action. It extends what the model can do beyond just generating text.
3. **Grounding:** AI models can sometimes make mistakes or "hallucinate" (make up facts). Grounding is a technique to prevent this by connecting the model to a reliable source of truth, like your company's internal database or a specific set of documents. When the AI answers a question, it first checks this data source to make sure its response is accurate and based on real information. This makes the AI's answers much more trustworthy.
4. **Model Tuning:** While the base models are very smart, you can make them even smarter for your specific needs. Tuning is the process of training a model on your own data to adapt it for a particular task. For example, you could tune a model on your company's customer support chats to create a chatbot that understands your customers' unique problems and speaks in your brand's voice with high accuracy.
5. **Image Generation**

vertex AI Agent Builder

Think of Vertex AI Agent Builder as Google's advanced workshop for creating your own smart AI assistants, often called "agents." It's a platform that provides all the tools you need to build and manage AI applications that can do much more than just chat. These agents can reason, use external tools, and access specific data to accomplish complex tasks.

The platform is designed for everyone, from people who don't write code to expert developers. You can build agents using simple natural language, or you can use powerful coding frameworks for more control.

**The Core Idea: What is an "AI Agent"?**

An AI Agent is a step beyond a simple chatbot. It uses a powerful Large Language Model (LLM), like Gemini, as its "brain" to do three main things:

1. **Reason and Plan:** It can understand a user's goal, break it down into smaller steps, and figure out how to achieve it.
2. **Use Tools:** It can connect to other programs and services (like a weather app, a flight booking system, or a company database) to get live information or perform actions.
3. **Access Specific Knowledge:** It can be connected to your company's private documents or other reliable data sources to provide accurate, relevant answers.

Imagine you ask an agent, "Find the top three customer complaints from last month about our new product and summarize them in an email to the support team." A simple chatbot can't do this. An AI agent, however, could be designed to:

**Step 1:** Connect to your company's customer support database (a **tool**).

**Step 2:** Search for the relevant complaints (using its **knowledge**).

**Step 3:** Use its LLM brain to analyze and summarize the findings (**reasoning**).

**Step 4:** Draft an email and send it (another **tool**).

**Key Capabilities Explained Simply**

Vertex AI Agent Builder provides several key features to make these powerful agents a reality.

**1. Grounding: Making the AI More Accurate and Trustworthy**

A major challenge with AI models is that they can sometimes make up facts (this is called "hallucination") or have outdated information. **Grounding** is the process of "fact-checking" the AI's responses against a reliable source of truth.

* **Grounding with Google Search:** The agent can use the power of Google Search to find up-to-date, public information to answer questions accurately.
* **Grounding on Your Data (with RAG):** This is extremely powerful. You can connect the agent to your own private data—like internal documents, product manuals, or databases. This technique, called **Retrieval-Augmented Generation (RAG)**, means the agent first *retrieves* relevant information from your data and then uses it to *generate* a fact-based answer. This ensures the responses are specific to your business and highly accurate.

**2. Tools: Giving the AI Superpowers**

This allows the AI model to connect to the outside world. By using **function calling** and **extensions**, you can give the model the ability to interact with other software and APIs. This turns the AI from a simple text generator into an active assistant that can perform tasks on your behalf.

**3. Orchestration: Managing Complex Tasks**

When an agent needs to perform a multi-step task, something needs to act as the "conductor" to manage the process. **Orchestration** tools help coordinate the flow of information between the LLM, the various tools, and the user.

* **LangChain on Vertex AI:** For developers, this provides a popular open-source framework to build complex chains of logic and create sophisticated agents with code.

**4. Building for Everyone**

Vertex AI Agent Builder supports different skill levels:

* **Low-Code/No-Code:** You can use a visual interface to design conversational agents without writing code.
* **Code-First:** Developers can use libraries like **LangChain** or the **Firebase Genkit plugin** to build custom applications with full control, integrating Google's AI models directly into their code.

In short, Vertex AI Agent Builder is a comprehensive suite of tools designed to help you build reliable, powerful, and secure AI agents that are grounded in real data and can interact with the world to get things done.

vertex Search

Concise

Vertex AI Search is a managed Google platform that helps you build powerful search and recommendation features for your websites and apps.

A key capability is **"grounding,"** which connects an AI model to a specific data source, like Google Search or your own private documents. This makes the AI's answers more factual and trustworthy by preventing it from making things up ("hallucinating").

To use your own data, you create a "data store" using Vertex AI Agent Builder.

Key features include:

* **Smart Search:** Understands natural language, not just keywords.
* **AI Summaries:** Can summarize search results and have conversations.
* **Recommendations:** Suggests similar content to users.

**What is Vertex AI Search?**

Think of Vertex AI Search as a powerful, ready-to-use service from Google that lets you build incredibly smart search bars and recommendation features (like a "you might also like" section) for your website or mobile app. Instead of just matching keywords, it uses advanced AI, including Large Language Models, to understand what your users actually mean. Google manages all the complicated technology behind the scenes, so you can focus on creating a great experience for your users.

**The Most Important Feature: Grounding**

A major challenge with AI models is that they can sometimes make up facts that sound believable but aren't true. This is called a **"hallucination."**

**Grounding** is the solution to this problem. It works by connecting the AI model to a specific, reliable source of information. This forces the AI to "ground" its answers in facts from that source, which dramatically reduces the chances of it inventing content.

In simple terms, grounding helps the AI by:

* **Stopping it from making things up.**
* **Anchoring its answers to real, specific information.**
* **Making the AI's responses more reliable and trustworthy.**

You can ground a model in two main ways:

1. **With Google Search:** You can connect the model to the entire public internet. This gives it a vast range of up-to-date knowledge on almost any topic.
2. **With Your Own Data:** This is extremely powerful for businesses. You can have the AI ground its answers in your own private information, like your company's product manuals, internal documents, or website content. To do this, you first need to put your data into a **"data store"** using a tool called Vertex AI Agent Builder.

**Other Key Features of Vertex AI Search**

* **Smart Search (Semantic Search):** It understands the *meaning* behind a search, not just the exact words. It can handle synonyms, correct spelling mistakes, and suggest what you might be looking for as you type.
* **AI Summaries and Conversations:** You can enable features where the AI provides a quick summary of the search results. Users can even have a conversation with the search engine, asking follow-up questions to narrow down what they're looking for.
* **Recommendations:** The AI can understand the content a user is looking at and automatically suggest other similar or relevant items, keeping them engaged.
* **Easy to Use:** You can set up and manage your search apps through the Google Cloud console (a web-based interface) or by using APIs if you are a developer who wants to integrate it directly into your code.

**Types of Search Apps You Can Build**

Vertex AI Search isn't just a one-size-fits-all tool. You can create different types of specialized search applications:

* **Generic Search:** Perfect for a company website or a private database of documents, allowing users to easily find the content you want them to see.
* **Media Search:** Specially designed for searching through catalogs of movies, videos, or music.
* **Healthcare Search:** A secure and specialized search for querying medical records stored in a specific format (called FHIR), allowing users to search through clinical data and related files like PDFs or images.

Reasoning Engine

**What is Reasoning Engine?**

Think of Reasoning Engine as a service from Google Cloud that makes it much easier to build and run your own powerful AI "agents." An AI agent is more than just a chatbot; it's a smart application that can reason, plan, and use external tools to complete complex tasks.

Reasoning Engine is built on top of **LangChain**, a popular open-source framework for creating these kinds of AI applications. Google's service takes care of all the complicated backend work, so you can focus on building the agent itself.

**Key Benefits of Using Reasoning Engine**

* **Deep Integration with Google's AI:** It's designed to work perfectly with Google's best AI models, like Gemini. It also seamlessly uses advanced features like **Function Calling**, which is how the AI model decides to use a tool.
* **Security and Scalability:** You don't have to worry about setting up servers, managing security, or figuring out how to handle lots of users. Google's managed service handles all of that for you automatically, including creating the necessary containers, setting up authentication, and scaling up or down as needed.
* **Simplified Deployment:** If you already know how to use LangChain, you're ready to use Reasoning Engine. You build your application using familiar LangChain methods, and then you can deploy it to Google's powerful, production-ready environment with a single command.

**How Does an Agent Work? (The System Flow)**

* Here’s a step-by-step look at what happens when you use an agent built with Reasoning Engine:
* **User Asks a Question:** A user makes a request, like "What is Reasoning Engine and how does it relate to LangChain?"
* **Agent Creates a Prompt:** The agent takes the user's question and formats it into a detailed prompt for the AI model (the "brain," like Gemini).
* **Model Thinks and Decides:** The AI model analyzes the prompt and decides if it can answer directly or if it needs help from one of its "tools."
* **Agent Uses a Tool (If Needed):** A **tool** is just a custom Python function you write that can connect to an external service (like a database, a search engine API, or your company's internal software). If the model decides it needs more information, it uses **Function Calling** to tell the agent which tool to use and what information to look for.
  + Example: For the question above, the model might decide to use a wikipedia\_search tool to get the latest definitions for "Reasoning Engine" and "LangChain."
* **Tool Returns Results:** The agent runs the Python function, gets the results (e.g., the text from the Wikipedia page), and sends them back to the AI model.
* **Model Generates the Final Answer:** Now, with this new, factual information from the tool, the model generates a comprehensive and accurate answer for the user. This process of using external data helps prevent the AI from making things up (a problem called "hallucination").

**How to Build and Deploy an Agent**

Here is a simplified overview of the development process:

**Set Up Your Environment:**

* Enable the necessary Google Cloud services (like Vertex AI).
* Install the Vertex AI SDK for Python with the specific extras for Reasoning Engine.
* Initialize the SDK in your code, telling it your project ID and where to store temporary files.

# Install the required libraries

# pip install google-cloud-aiplatform[langchain,reasoningengine]

import vertexai

# Initialize the SDK with your project details

vertexai.init(

    project="your-gcp-project-id",

    location="your-region",

    staging\_bucket="gs://your-storage-bucket-name"

)

1. **Define Your Model:**

model = "gemini-1.5-pro-001"

**Define Your Tools:**

* Write standard Python functions that will act as your tools. The key is to include a clear description in the function's docstring, as this is what the AI model reads to understand what the tool does and when to use it.
* **Create and Test the Agent:**
  + Use the LangChain templates provided by the Vertex AI SDK to combine your model and your tools into an agent.
  + You can then test it on your local machine to make sure it behaves as you expect.
* **Deploy the Agent:**
  + Once you're happy with how it works, you use a single command from the SDK to deploy your agent.
  + Reasoning Engine automatically packages your code and tools into a container and deploys it to Google's fully managed, scalable, and secure runtime environment.

Bottom of Form