AWS Database Blog

The role of vector datastores in generative AI applications

by G2 Krishnamoorthy, Rahul Pathak, and Vlad Vlasceanu | on 26 JUL 2023 | in Amazon Aurora, Amazon OpenSearch Service, Amazon RDS, Generative AI, PostgreSQL Compatible, RDS For PostgreSQL | Permalink | Comments | Share

Generative AI has captured our imagination and is transforming industries with its ability to answer questions, write stories, create art, and even generate code. AWS customers are increasingly asking us how they can best take advantage of generative AI in their own businesses. Most have accumulated a wealth of domain-specific data (financial records, health records, genomic data, supply chain, and so on), which provides them with a unique and valuable perspective into their business and broader industry. This proprietary data can be an advantage and differentiator for your generative AI strategy.

At the same time, many customers have also noticed the rise in popularity of vector datastores, or vector databases, used in generative AI applications, and are wondering how these solutions fit in their overall data strategy around generative AI applications. In this post, we describe the role of vector databases in generative AI applications, and how AWS solutions can help you harness the power of generative AI.

Generative AI applications

At the heart of every generative AI application is a large language model (LLM). An LLM is a machine learning (ML) model trained on a large body of content—such as all the content accessible on the internet. LLMs trained on vast amounts of publicly accessible data are considered foundational models (FMs). They can be adapted and fine-tuned for a wide range of use cases. Amazon SageMaker JumpStart provides a variety of pre-trained, open-source, and proprietary foundational models for you to build upon, such as Stability AI's Text2Image model, which can generate photorealistic images using a text prompt, or Hugging Face's Text2Text Flan T-5 model for text generation. Amazon Bedrock, the easiest way to build and scale generative AI applications with FMs, makes models from AI21 Labs, Anthropic, Stability AI, and Amazon Titan accessible via an API.

Although a generative AI application relying purely on an FM will have access to broad real-world knowledge, it needs to be customized to produce accurate results on topics that are domain specific or specialized. Also, hallucinations (results that lack accuracy but look correct with confidence) occur more frequently the more specialized the interaction is. So how can you customize your generative AI application for domain specificity?

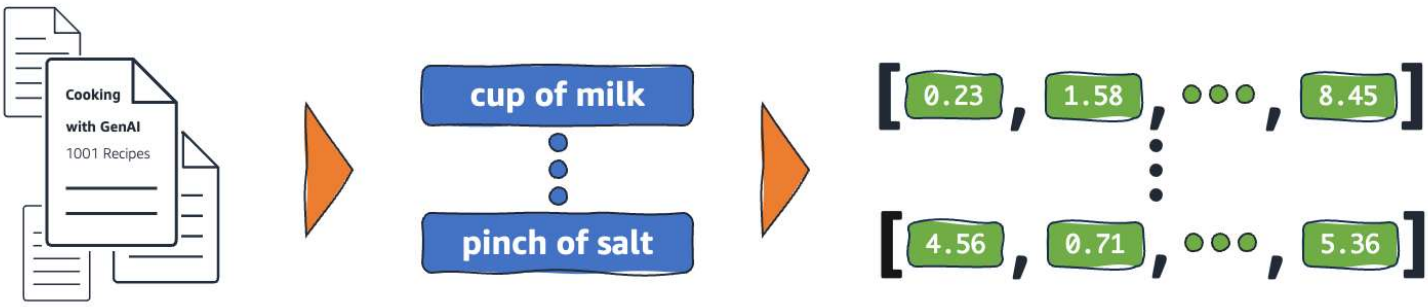
Adding domain specificity using vector datastores

Prompt engineering (also referred to as *in-context learning*) may be the easiest way to ground your generative AI application in your domain-specific context and improve accuracy. Although it won't completely eliminate hallucinations, this technique will scope down the spectrum of semantic meaning to your own domain.

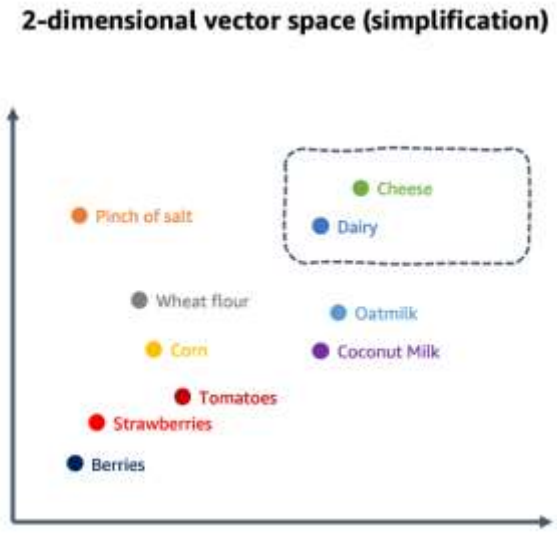
At its core, the FM infers the next token based on a set of input tokens. A *token* in this case refers to any element with semantic meaning, like a word or phrase in text generation. The more contextually relevant inputs you provide, the higher the likelihood that the next token inferred is also contextually relevant. The prompt you query the FM with contains the input tokens, plus as much contextually relevant data as possible.

The contextual data typically comes from your internal databases or data lakes, the systems that host your domain-specific data. Although you can enrich the prompt by simply appending additional domain-specific data from these data stores, vector datastores help you engineer your prompts with semantically relevant inputs. This method is called Retrieval Augmented Generation (RAG). In practice, you will likely engineer a prompt with both contextually personalized data, like user profile information, and semantically similar data.

For generative AI usage, your domain-specific data must be encoded as a set of elements, each expressed internally as a *vector*. The vector contains a set of numeric values across a set of dimensions (array of numbers). The following figure illustrates an example of transforming context data into semantic elements and then vectors.



These numeric values are used to map elements in relation to each other in a multi-dimensional vector space. When the vector elements are semantic (they represent a form of meaning), the proximity becomes an indicator for contextual relationship. Used in this way, such vectors are referred to as *embeddings*. For example, the semantic element for “Cheese” may be put in proximity to the semantic element for “Dairy” in a multi-dimensional space representing the data domain context of groceries or cooking. Depending on your specific domain context, a semantic element may be a word, phrase, sentence, paragraph, whole document, image, or something else entirely. You split your domain-specific dataset into meaningful elements that can be related to each other. For example, the following figure illustrates a simplified vector space for a context on cooking.



As a result, to produce the relevant context for the prompt, you need to query a database and find elements that are closely related to your inputs in the vector space. A *vector datastore* is a system that allows you to store and query vectors at scale, with efficient nearest neighbor query algorithms and appropriate indexes to improve data retrieval. Any database management system that has these vector-related capabilities can be a vector datastore. Many commonly used database systems offer these vector capabilities along with the rest of their functionality. One advantage of storing your domain-specific datasets in a database with vector capabilities is that your vectors will be located close to the source data. You can enrich vector data with additional metadata, without having to query external databases, and you can simplify your data processing pipelines.

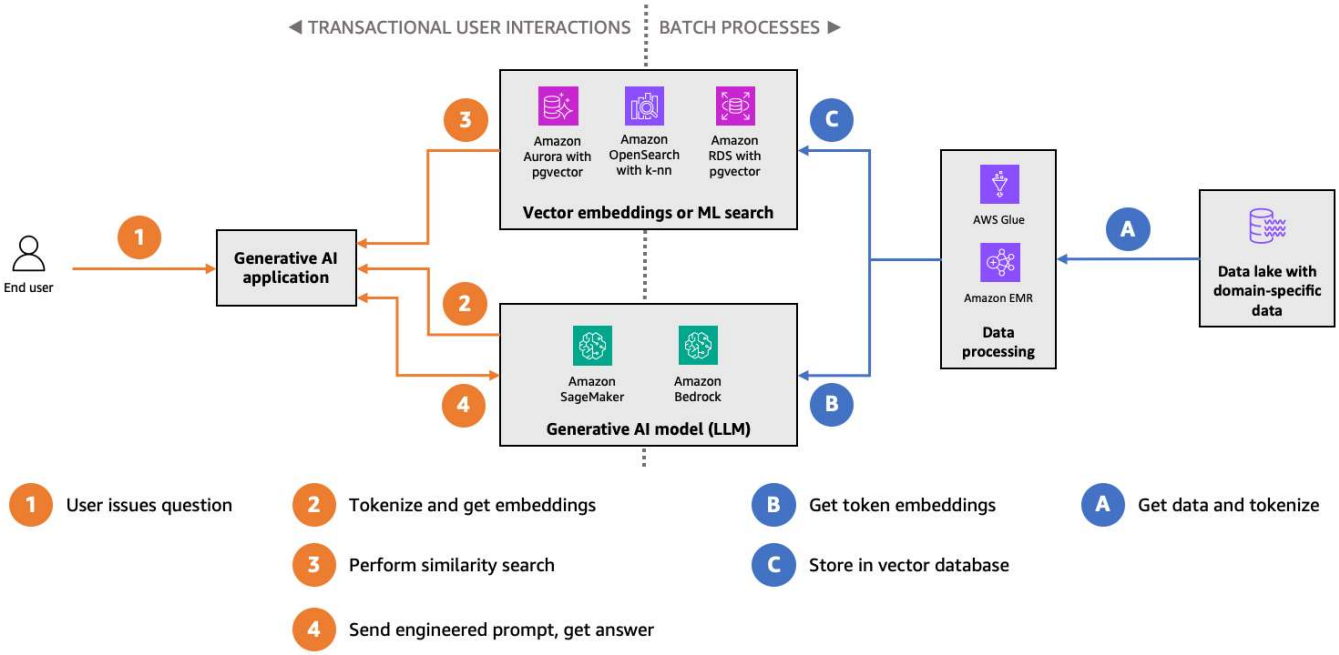
To help you get started with vector datastores quickly, today we announced the vector engine for Amazon OpenSearch Serverless, which provides a simple API for storing and querying billions of embeddings, once it becomes generally available. However, we think that in the fullness of time, all AWS databases will have vector capabilities, because that simplifies your operations and data integration. Additionally, the following options are available for more advanced vector datastore needs:

- An Amazon Aurora PostgreSQL-Compatible Edition relational database, with the pgvector open-source vector similarity search extension
- Amazon OpenSearch Service, a distributed search and analytics service, with the k-NN (k-nearest neighbor) plugin, and vector engine for Amazon OpenSearch Serverless
- An Amazon Relational Database Service (Amazon RDS) for PostgreSQL relational database, with the pgvector extension

Embeddings should be stored close to your source data. As a result, where you store your data today, as well as familiarity with these database technologies, scale in terms of vector dimensions, number of embeddings, and performance needs will determine which option is right for you. Before we dive deeper into more specific guidance for these options, let's first understand how RAG works and how you apply vector datastores in RAG.

Using vector datastores for RAG

You can use embeddings (vectors) to improve the accuracy of your generative AI application. The following diagram illustrates this data flow.



You take your domain-specific dataset (the right side of the preceding figure, depicted in blue), split it into semantic elements, and use the FM to compute the vectors for these semantic elements. Then you store these vectors in a vector datastore, which will enable you to perform similarity search.

In your generative AI application (left side of the preceding figure, depicted in orange), you take the end-user-provided question, split it into semantic elements (tokenization) using the same algorithm that was used on your dataset, and query the vector datastore for the nearest neighbors in the vector space for the input elements. The store will provide you with contextually similar semantic elements that you then add to your engineered prompt. This process will further ground the LLM into your domain-specific context, increasing the likelihood that the LLM output is accurate and relevant to that context.

Performing similarity searches in your vector datastore, in the critical path of end-users, uses concurrent read queries. Batch processes to populate the vector datastore with embeddings and keep up with data changes are mostly data writes to the vector datastore. Aspects of this usage pattern along with previously mentioned considerations, like familiarity and scale, determine which service—Aurora PostgreSQL-Compatible, OpenSearch Service, the vector engine for OpenSearch Serverless, or Amazon RDS for PostgreSQL—is right for you.

Vector datastore considerations

The usage pattern we described also leads to some unique and important considerations for vector datastores.

The volume of domain-specific data you wish to use and the process you use to split up that data into semantic elements will determine the number of embeddings your vector datastore needs to support. As your domain-specific data grows and changes over time, your vector datastore also has to accommodate that growth. This has impact on indexing efficiency and performance at scale. It's not uncommon for domain-specific datasets to result in hundreds of millions—even billions—of embeddings. You use a tokenizer to split the data, and the Natural Language Toolkit

(NLTK) provides several general purpose tokenizers you can use. But you can use alternatives, too. Ultimately, the right tokenizer depends on what a semantic element in your domain-specific dataset is—as previously mentioned, it could be a word, phrase, paragraph of text, entire document, or any subdivision of your data that holds independent meaning.

The number of dimensions for the embedding vectors is another important factor to consider. Different FMs produce vectors with different numbers of dimensions. For example, the all-MiniLM-L6-v2 model produces vectors with 384 dimensions, and Falcon-40B vectors have 8,192 dimensions. The more dimensions a vector has, the richer the context it can represent—up to a point. You will eventually see diminishing returns and increased query latency. This eventually leads to the curse of dimensionality (objects appear sparse and dissimilar). To perform semantic similarity searches, you generally need vectors with dense dimensionality, but you may need to reduce the dimensions of your embeddings for your database to handle such searches efficiently.

Another consideration is whether you need exact similarity search results. Indexing capabilities in vector datastores will speed up similarity search considerably, but they will also use an approximate nearest neighbor (ANN) algorithm to produce results. ANN algorithms provide performance and memory efficiencies in exchange for accuracy. They can't guarantee that they return the exact nearest neighbors every time.

Finally, consider data governance. Your domain-specific datasets likely contain highly sensitive data, such as personal data or intellectual property. With your vector datastore close to your existing domain-specific datasets, you can extend your access, quality, and security controls to your vector datastore, simplifying operations. In many cases, it won't be feasible to strip away such sensitive data without affecting the semantic meaning of the data, which in turn reduces accuracy. Therefore, it's important to understand and control the flow of your data through the systems that create, store, and query embeddings.

Using Aurora PostgreSQL or Amazon RDS for PostgreSQL with pgvector

Pgvector, an open-source, community-supported PostgreSQL extension, is available both in Aurora PostgreSQL and Amazon RDS for PostgreSQL. The extension expands PostgreSQL with a vector data type called `vector`, three query operators for similarity searching (Euclidian, negative inner product, and cosine distance), and the ivfflat (inverted file with stored vectors) indexing mechanism for vectors to perform faster approximate distance searches. Although you can store vectors with up to 16,000 dimensions, only 2,000 dimensions can be indexed to improve similarity search performance. In practice, customers tend to use embeddings with fewer dimensions. The post Building AI-powered search in PostgreSQL using Amazon SageMaker and pgvector is a great resource to dive deeper into this extension.

You should strongly consider using Aurora PostgreSQL with the pgvector extension for your vector datastore if you are already heavily invested in relational databases, especially PostgreSQL, and have a lot of expertise in that space. Also, highly structured domain-specific datasets are a more natural fit for relational databases. Amazon RDS for PostgreSQL can also be a great choice if you need to use specific community versions of PostgreSQL. Similarity search queries (reads) can also scale horizontally subject to the maximum number of read replicas supported by Aurora in a single DB cluster (15) and Amazon RDS in a replication chain (15).

Aurora PostgreSQL also supports Amazon Aurora Serverless v2, an on-demand, auto scaling configuration that can adjust the compute and memory capacity of your DB instances automatically based on load. This configuration simplifies operations because you no longer have to provision for peak or perform complex capacity planning in most use cases.

Amazon Aurora Machine Learning (Aurora ML) is a feature you can use to make calls to ML models hosted in Amazon SageMaker via SQL functions. You can use it to make calls to your FMs to generate embeddings directly from your database. You can package these calls into stored procedures or integrate them with other PostgreSQL capabilities, such that the vectorization process is completely abstracted away from the application. With the batching capabilities built into Aurora ML, you may not even need to export the initial dataset from Aurora in order to transform it to create the initial set of vectors.

Using OpenSearch Service with the k-NN plugin and the vector engine for OpenSearch Serverless

The k-NN plugin expands OpenSearch, an open-source, distributed search and analytics suite, with the custom `knn_vector` data type, enabling you to store embeddings in OpenSearch indexes. The plugin also provides three methods to perform k-nearest neighbor similarity searches: Approximate k-NN, Script Score k-NN (exact), and the Painless extensions (exact). OpenSearch includes the Non-Metric Space Library (NMSLIB) and Facebook AI Research's FAISS library. You can use different search algorithms for distance to find the best one that meets your needs. This plugin is also available in OpenSearch Service, and the post Amazon OpenSearch Service's vector database capabilities explained is a great resource to dive deeper into these features.

Due to the distributed nature of OpenSearch, it's a great choice for vector datastores with a very large number of embeddings. Your indexes scale horizontally, allowing you to handle more throughput for storing embeddings and performing similarity searches. It's also a great choice for customers who want to have deeper control over the method and algorithms used to perform searches. Search engines are designed for low-latency, high throughput querying, trading off transactional behavior to achieve that.

OpenSearch Serverless is an on-demand serverless configuration that removes the operational complexities of provisioning, configuring, and tuning OpenSearch domains. You simply start by creating a collection of indexes and start populating your index data. The newly announced vector engine for OpenSearch Serverless is offered as a new vector collection type, along with search and time series collections. It gives you an easy way to get started working with vector similarity search. It provides an easy-to-operate pairing for Amazon Bedrock to integrate prompt engineering into your generative AI applications, without needing advanced expertise in ML or vector technology. With the vector engine, you're able to query vector embeddings, metadata, and descriptive text easily within a single API call, resulting in more accurate search results while reducing complexity in your application stack.

Vectors in OpenSearch with the k-NN plugin support up to 16,000 dimensions when using the `nmslib` and `faiss` engines, and 1,024 dimensions with the Lucene engine. Lucene provides the core search and analytics capabilities of OpenSearch, along with vector search. OpenSearch uses a custom REST API for most operations, including similarity searches. It enables greater flexibility when interacting with OpenSearch indexes, while allowing you to reuse skills for building distributed web-based applications.

OpenSearch is also a great option if you need to combine semantic similarity search with keyword search use cases. Prompt engineering for generative AI applications involves both retrieval of contextual data and RAG. For example, a customer support agent application may build a prompt by including previous support cases with the same keywords, as well as support cases that are semantically similar, so the recommended solution is grounded in the appropriate context.

The Neural Search plugin (experimental) enables the integration of ML language models directly into your OpenSearch workflows. With this plugin, OpenSearch automatically creates vectors for the text provided during ingestion and search. It then seamlessly uses the vectors for search queries. This can simplify similarity search tasks used in RAG.

Additionally, if you prefer a fully managed semantic search experience on domain-specific data, you should consider Amazon Kendra. It provides out-of-the-box semantic search capabilities for state-of-the-art ranking of documents and passages, eliminating the overhead of managing text extraction, passage splitting, getting embeddings, and managing vector datastores. You can use Amazon Kendra for your semantic search needs and package the results into your engineered prompt, thereby maximizing the benefits of RAG with the least amount of operational overhead. The post Quickly build high-accuracy Generative AI applications on enterprise data using Amazon Kendra, LangChain, and large language models provides deeper guidance for this use case.

Finally, Aurora PostgreSQL and Amazon RDS for PostgreSQL with pgvector, the vector engine for OpenSearch Serverless, and OpenSearch Service with k-NN are supported in LangChain. LangChain is a popular Python framework for developing data-aware, agent-style applications based on LLMs.

Summary

Embeddings should be stored and managed close to your domain-specific datasets. Doing so allows you to combine them with additional metadata without using additional, external data sources. Your data is also not static, but changes over time, and storing the embeddings near your source data simplifies your data pipelines for keeping the embeddings up to date.

Aurora PostgreSQL and Amazon RDS for PostgreSQL with pgvector, as well as the vector engine for OpenSearch Serverless and OpenSearch Service with the k-NN plugin, are great choices for your vector datastore needs, but which solution is right for you will ultimately depend on your use case and priorities. If your database of choice doesn't have vector capabilities, the options discussed in this post span the spectrum of familiarity with SQL and NoSQL and are straightforward to pick up without a lot of operational overhead. No matter which option you choose, your vector datastore solution needs to sustain the concurrent throughput dispatched by the application. Validate your solution at scale with a full set of embeddings, so the similarity search response latencies meet your expectations.

At the same time, prompt engineering used in conjunction with foundational models provided by SageMaker JumpStart and Amazon Bedrock will enable you to build innovative generative AI solutions to delight your customers, without having to invest in significant ML skills.

On a final note, keep in mind technology is evolving rapidly in this space, and although we will make every effort to update our guidance as things change, the recommendations in this post may not be universally applicable.

Get started building generative AI applications on AWS today! Discover the tools and features AWS offers to help you innovate faster, and reinvent customer experiences.

About the authors

G2 Krishnamoorthy is VP of Analytics, leading AWS data lake services, data integration, Amazon OpenSearch Service, and Amazon QuickSight. Prior to his current role, G2 built and ran the Analytics and ML Platform at Facebook/Meta, and built various parts of the SQL Server database, Azure Analytics, and Azure ML at Microsoft.

Rahul Pathak is VP of Relational Database Engines, leading Amazon Aurora, Amazon Redshift, and Amazon QLDB. Prior to his current role, he was VP of Analytics at AWS, where he worked across the entire AWS database portfolio. He has co-founded two companies, one focused on digital media analytics and the other on IP-geolocation.

Vlad Vlasceanu is the Worldwide Tech Leader for Databases at AWS. He focuses on accelerating customer adoption of purpose-built databases, and developing prescriptive guidance mechanisms to help customers select the right databases for their workloads. He is also the leader of the database expert community within AWS, where he helps develop Solutions Architects' database skills and enables them to deliver the best database solutions for their customers.

Comments