

Amazon OpenSearch Service's vector database capabilities explained

by Jon Handler, Dylan Tong, Jianwei Li, and Vamshi Vijay Nakkirtha | on 21 JUN 2023 | in [Amazon OpenSearch Service](#), [Amazon SageMaker](#), [Artificial Intelligence](#), [Customer Solutions](#), [Foundational \(100\)](#), [Intermediate \(200\)](#), [Thought Leadership](#) | [Permalink](#) | [Comments](#) | [Share](#)

[OpenSearch](#) is a scalable, flexible, and extensible open-source software suite for search, analytics, security monitoring, and observability applications, licensed under the Apache 2.0 license. It comprises a search engine, OpenSearch, which delivers low-latency search and aggregations, OpenSearch Dashboards, a visualization and dashboarding tool, and a suite of plugins that provide advanced capabilities like alerting, fine-grained access control, observability, security monitoring, and vector storage and processing. [Amazon OpenSearch Service](#) is a fully managed service that makes it simple to deploy, scale, and operate OpenSearch in the AWS Cloud.

As an end-user, when you use OpenSearch's search capabilities, you generally have a goal in mind—something you want to accomplish. Along the way, you use OpenSearch to gather information in support of achieving that goal (or maybe the information is the original goal). We've all become used to the "search box" interface, where you type some words, and the search engine brings back results based on word-to-word matching. Let's say you want to buy a couch in order to spend cozy evenings with your family around the fire. You go to Amazon.com, and you type "a cozy place to sit by the fire." Unfortunately, if you run that search on Amazon.com, you get items like fire pits, heating fans, and home decorations—not what you intended. The problem is that couch manufacturers probably didn't use the words "cozy," "place," "sit," and "fire" in their product titles or descriptions.

In recent years, machine learning (ML) techniques have become increasingly popular to enhance search. Among them are the use of embedding models, a type of model that can encode a large body of data into an n-dimensional space where each entity is encoded into a vector, a data point in that space, and organized such that similar entities are closer together. An embedding model, for instance, could encode the semantics of a corpus. By searching for the vectors nearest to an encoded document — k-nearest neighbor (k-NN) search — you can find the most semantically similar documents. Sophisticated embedding models can support multiple modalities, for instance, encoding the image and text of a product catalog and enabling similarity matching on both modalities.

A vector database provides efficient vector similarity search by providing specialized indexes like k-NN indexes. It also provides other database functionality like managing vector data alongside other data types, workload management, access control and more. [OpenSearch's k-NN plugin provides core vector database functionality for OpenSearch](#), so when your customer searches for "a cozy place to sit by the fire" in your catalog, you can encode that prompt and use OpenSearch to perform a nearest neighbor query to surface that 8-foot, blue couch with designer arranged photographs in front of fireplaces.

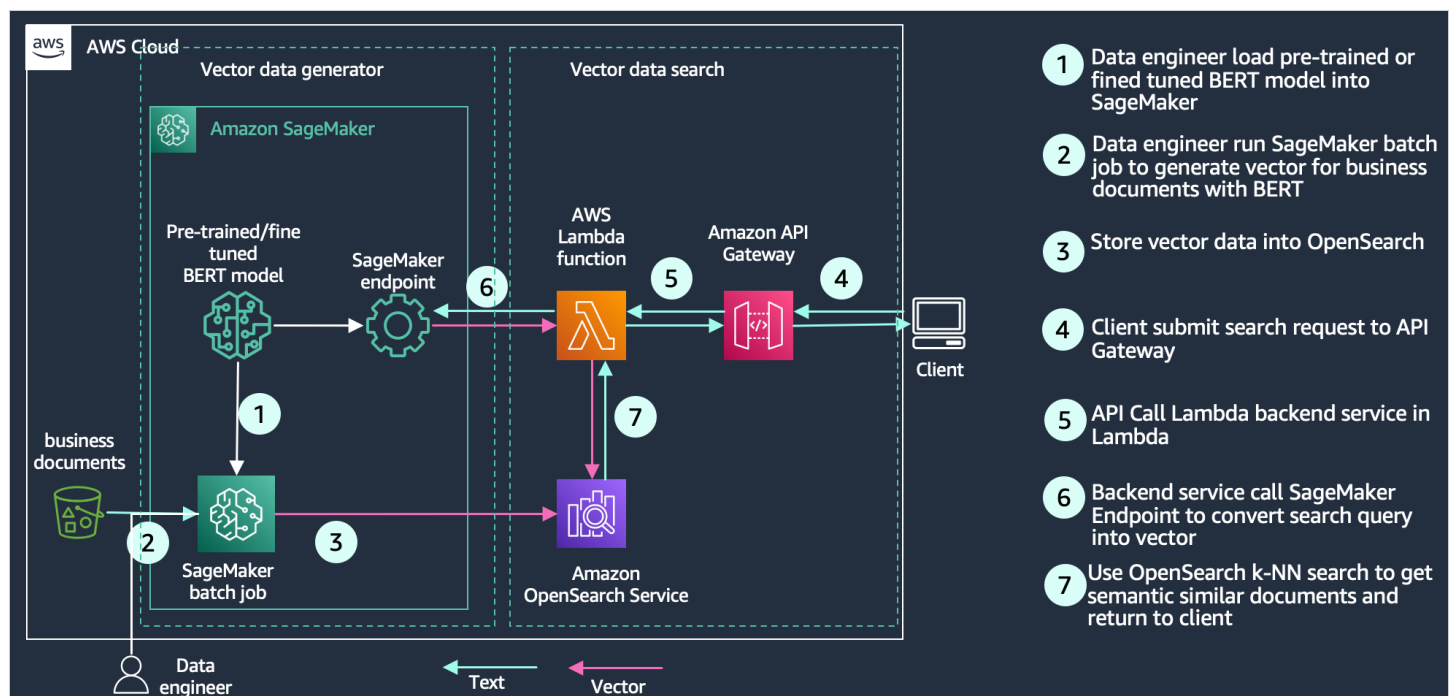
Using OpenSearch Service as a vector database

With OpenSearch Service's vector database capabilities, you can implement semantic search, Retrieval

Augmented Generation (RAG) with LLMs, recommendation engines, and search rich media.

Semantic search

With semantic search, you improve the relevance of retrieved results using language-based embeddings on search documents. You enable your search customers to use natural language queries, like “a cozy place to sit by the fire” to find their 8-foot-long blue couch. For more information, refer to [Building a semantic search engine in OpenSearch](#) to learn how semantic search can deliver a 15% relevance improvement, as measured by [normalized discounted cumulative gain \(nDCG\)](#) metrics compared with keyword search. For a concrete example, our [Improve search relevance with ML in Amazon OpenSearch Service](#) workshop explores the difference between keyword and semantic search, based on a [Bidirectional Encoder Representations from Transformers \(BERT\)](#) model, hosted by [Amazon SageMaker](#) to generate vectors and store them in OpenSearch. The workshop uses product question answers as an example to show how keyword search using the keywords/phrases of the query leads to some irrelevant results. Semantic search is able to retrieve more relevant documents by matching the context and semantics of the query. The following diagram shows an example architecture for a semantic search application with OpenSearch Service as the vector database.

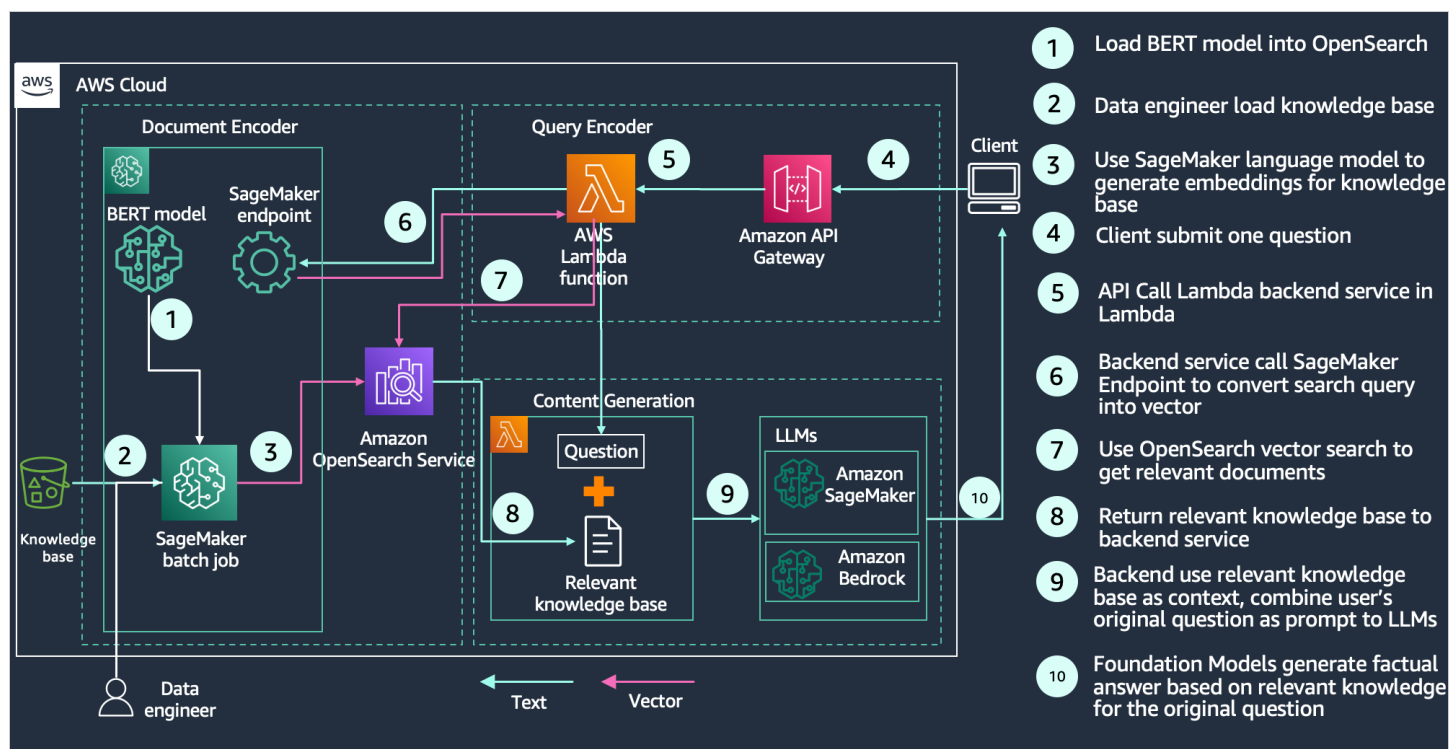


Retrieval Augmented Generation with LLMs

RAG is a method for building trustworthy generative AI chatbots using generative LLMs like OpenAI, ChatGPT, or [Amazon Titan Text](#). With the rise of generative LLMs, application developers are looking for ways to take advantage of this innovative technology. One popular use case involves delivering conversational experiences through intelligent agents. Perhaps you're a software provider with knowledge bases for product information, customer self-service, or industry domain knowledge like tax reporting rules or medical information about diseases and treatments. A conversational search experience provides an intuitive interface for users to sift through information through dialog and Q&A. Generative LLMs on their own are prone to [hallucinations](#)—a

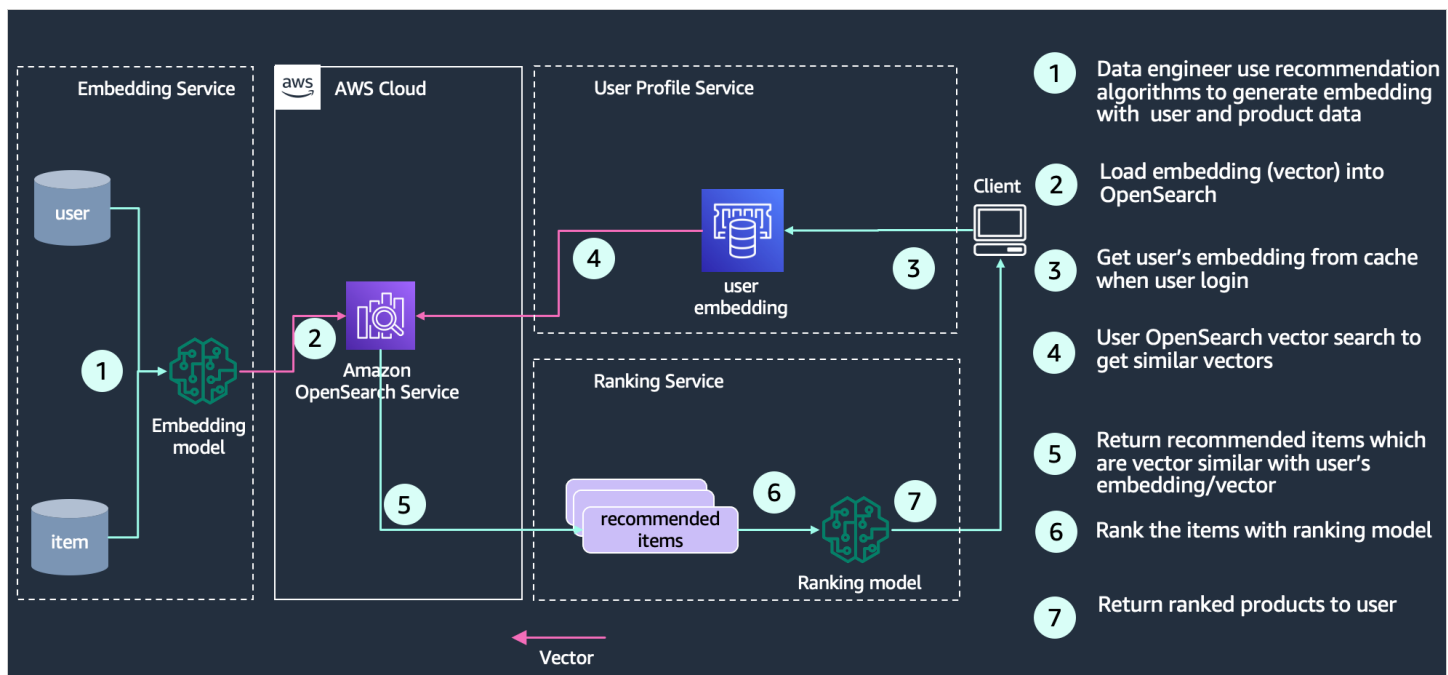
situation where the model generates a believable but factually incorrect response. RAG solves this problem by complementing generative LLMs with an external knowledge base that is typically built using a vector database hydrated with vector-encoded knowledge articles.

As illustrated in the following diagram, the query workflow starts with a question that is encoded and used to retrieve relevant knowledge articles from the vector database. Those results are sent to the generative LLM whose job is to augment those results, typically by summarizing the results as a conversational response. By complementing the generative model with a knowledge base, RAG grounds the model on facts to minimize hallucinations. You can learn more about building a RAG solution in the [Retrieval Augmented Generation module of our semantic search workshop](#).



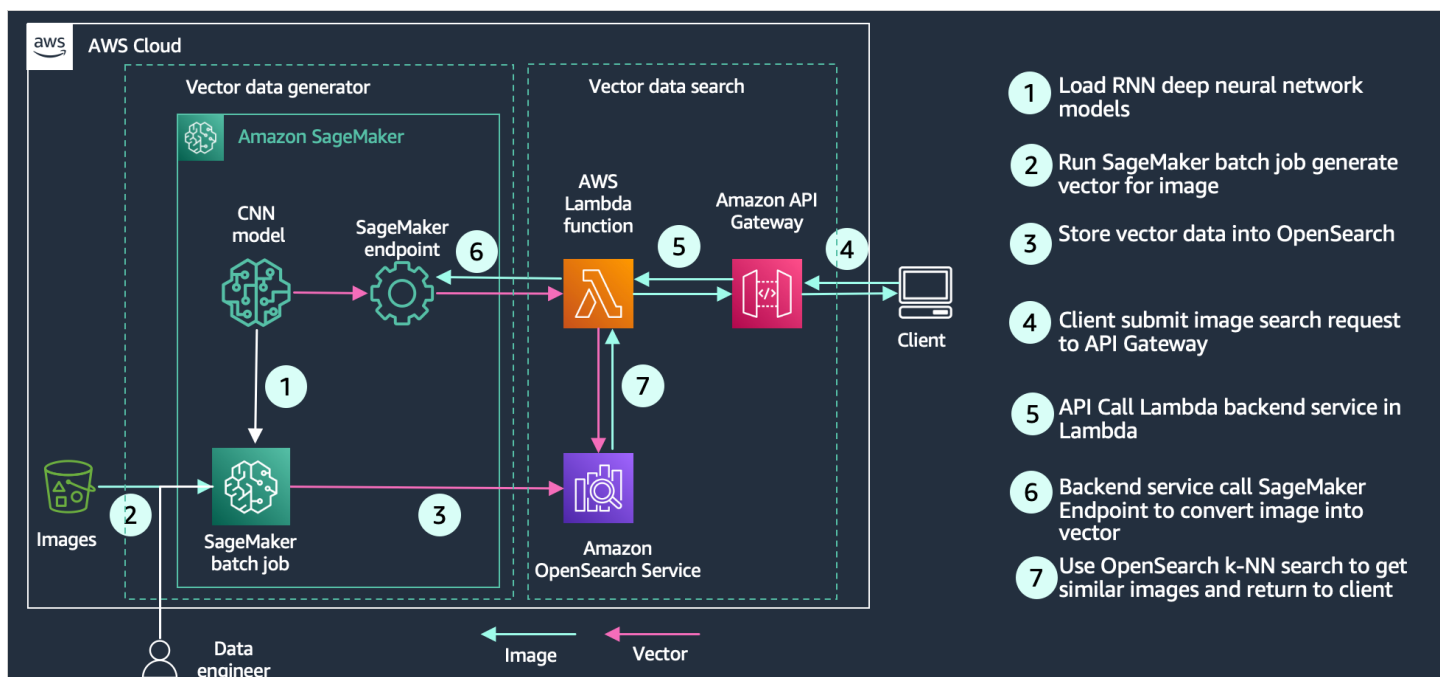
Recommendation engine

Recommendations are a common component in the search experience, especially for ecommerce applications. Adding a user experience feature like "more like this" or "customers who bought this also bought that" can drive additional revenue through getting customers what they want. Search architects employ many techniques and technologies to build recommendations, including [Deep Neural Network \(DNN\)](#) based recommendation algorithms such as the [two-tower neural net model](#), [YoutubeDNN](#). A trained embedding model encodes products, for example, into an embedding space where products that are frequently bought together are considered more similar, and therefore are represented as data points that are closer together in the embedding space. Another possibility is that product embeddings are based on co-rating similarity instead of purchase activity. You can employ this affinity data through calculating the vector similarity between a particular user's embedding and vectors in the database to return recommended items. The following diagram shows an example architecture of building a recommendation engine with OpenSearch as a vector store.



Media search

Media search enables users to query the search engine with rich media like images, audio, and video. Its implementation is similar to semantic search—you create vector embeddings for your search documents and then query OpenSearch Service with a vector. The difference is you use a computer vision deep neural network (e.g. [Convolutional Neural Network](#) (CNN)) such as [ResNet](#) to convert images into vectors. The following diagram shows an example architecture of building an image search with OpenSearch as the vector store.



Understanding the technology

OpenSearch uses approximate nearest neighbor (ANN) algorithms from the [NMSLIB](#), [FAISS](#), and [Lucene](#) libraries to power k-NN search. These search methods employ ANN to improve search latency for large datasets. Of the three search methods the k-NN plugin provides, this method offers the best search scalability for large datasets. The engine details are as follows:

- **Non-Metric Space Library (NMSLIB)** – NMSLIB implements the HNSW ANN algorithm
- **Facebook AI Similarity Search (FAISS)** – FAISS implements both HNSW and IVF ANN algorithms
- **Lucene** – Lucene implements the HNSW algorithm

Each of the three engines used for approximate k-NN search has its own attributes that make one more sensible to use than the others in a given situation. You can follow the general information in this section to help determine which engine will best meet your requirements.

In general, NMSLIB and FAISS should be selected for large-scale use cases. Lucene is a good option for smaller deployments, but offers benefits like smart filtering where the optimal filtering strategy—pre-filtering, post-filtering, or exact k-NN—is automatically applied depending on the situation. The following table summarizes the differences between each option.

	NMSLIB-HNSW	FAISS-HNSW	FAISS-IVF	Lucene-HNSW
Max Dimension	16,000	16,000	16,000	1024
Filter	Post filter	Post filter	Post filter	Filter while search
Training Required	No	No	Yes	No
Similarity Metrics	l2, innerproduct, cosinesimil, l1, linf	l2, innerproduct	l2, innerproduct	l2, cosinesimil
Vector Volume	Tens of billions	Tens of billions	Tens of billions	< Ten million
Indexing latency	Low	Low	Lowest	Low

Query Latency & Quality	Low latency & high quality	Low latency & high quality	Low latency & low quality	High latency & high quality
Vector Compression	Flat	Flat Product Quantization	Flat Product Quantization	Flat
Memory Consumption	High	High Low with PQ	Medium Low with PQ	High

Approximate and exact nearest-neighbor search

The OpenSearch Service k-NN plugin supports three different methods for obtaining the k-nearest neighbors from an index of vectors: approximate k-NN, score script (exact k-NN), and painless extensions (exact k-NN).

Approximate k-NN

The first method takes an approximate nearest neighbor approach—it uses one of several algorithms to return the approximate k-nearest neighbors to a query vector. Usually, these algorithms sacrifice indexing speed and search accuracy in return for performance benefits such as lower latency, smaller memory footprints, and more scalable search. Approximate k-NN is the best choice for searches over large indexes (that is, hundreds of thousands of vectors or more) that require low latency. You should not use approximate k-NN if you want to apply a filter on the index before the k-NN search, which greatly reduces the number of vectors to be searched. In this case, you should use either the score script method or painless extensions.

Score script

The second method [extends the OpenSearch Service score script functionality](#) to run a brute force, exact k-NN search over `knn_vector` fields or fields that can represent binary objects. With this approach, you can run k-NN search on a subset of vectors in your index (sometimes referred to as a *pre-filter search*). This approach is preferred for searches over smaller bodies of documents or when a pre-filter is needed. Using this approach on large indexes may lead to high latencies.

Painless extensions

The third method adds the distance functions as painless extensions that you can use in more complex combinations. Similar to the k-NN score script, you can use this method to perform a brute force, exact k-NN search across an index, which also supports pre-filtering. This approach has slightly slower query performance

compared to the k-NN score script. If your use case requires more customization over the final score, you should use this approach over score script k-NN.

Vector search algorithms

The simple way to find similar vectors is to use [k-nearest neighbors \(k-NN\)](#) algorithms, which compute the distance between a query vector and the other vectors in the vector database. As we mentioned earlier, the score script k-NN and painless extensions search methods use the exact k-NN algorithms under the hood. However, in the case of extremely large datasets with high dimensionality, this creates a scaling problem that reduces the efficiency of the search. Approximate nearest neighbor (ANN) search methods can overcome this by employing tools that restructure indexes more efficiently and reduce the dimensionality of searchable vectors. There are different ANN search algorithms; for example, locality sensitive hashing, tree-based, cluster-based, and graph-based. OpenSearch implements two ANN algorithms: Hierarchical Navigable Small Worlds (HNSW) and Inverted File System (IVF). For a more detailed explanation of how the HNSW and IVF algorithms work in OpenSearch, see blog post "[Choose the k-NN algorithm for your billion-scale use case with OpenSearch](#)".

Hierarchical Navigable Small Worlds

The HNSW algorithm is one of the most popular algorithms out there for ANN search. The core idea of the algorithm is to build a graph with edges connecting index vectors that are close to each other. Then, on search, this graph is partially traversed to find the approximate nearest neighbors to the query vector. To steer the traversal towards the query's nearest neighbors, the algorithm always visits the closest candidate to the query vector next.

Inverted File

The IVF algorithm separates your index vectors into a set of buckets, then, to reduce your search time, only searches through a subset of these buckets. However, if the algorithm just randomly split up your vectors into different buckets, and only searched a subset of them, it would yield a poor approximation. The IVF algorithm uses a more elegant approach. First, before indexing begins, it assigns each bucket a representative vector. When a vector is indexed, it gets added to the bucket that has the closest representative vector. This way, vectors that are closer to each other are placed roughly in the same or nearby buckets.

Vector similarity metrics

All search engines use a similarity metric to rank and sort results and bring the most relevant results to the top. When you use a plain text query, the similarity metric is called TF-IDF, which measures the importance of the terms in the query and generates a score based on the number of textual matches. When your query includes a vector, the similarity metrics are spatial in nature, taking advantage of proximity in the vector space. OpenSearch supports several similarity or distance measures:

- **Euclidean distance** – The straight-line distance between points.
- **L1 (Manhattan) distance** – The sum of the differences of all of the vector components. L1 distance measures

how many orthogonal city blocks you need to traverse from point A to point B.

- **L-infinity (chessboard) distance** – The number of moves a King would make on an n-dimensional chessboard. It's different than Euclidean distance on the diagonals—a diagonal step on a 2-dimensional chessboard is 1.41 Euclidean units away, but 2 L-infinity units away.
- **Inner product** – The product of the magnitudes of two vectors and the cosine of the angle between them. Usually used for natural language processing (NLP) vector similarity.
- **Cosine similarity** – The cosine of the angle between two vectors in a vector space.
- **Hamming distance** – For binary-coded vectors, the number of bits that differ between the two vectors.

Advantage of OpenSearch as a vector database

When you use OpenSearch Service as a vector database, you can take advantage of the service's features like usability, scalability, availability, interoperability, and security. More importantly, you can use OpenSearch's search features to enhance the search experience. For example, you can use [Learning to Rank](#) in OpenSearch to integrate user clickthrough behavior data into your search application and improve search relevance. You can also combine OpenSearch text search and vector search capabilities to search documents with keyword and semantic similarity. You can also use other fields in the index to filter documents to improve relevance. For advanced users, you can use a hybrid scoring model to combine [OpenSearch's text-based relevance score, computed with the Okapi BM25 function](#) and its vector search score to improve the ranking of your search results.

Scale and limits

OpenSearch as vector database support billions of vector records. Keep in mind the following calculator regarding number of vectors and dimensions to size your cluster.

Number of vectors

OpenSearch VectorDB takes advantage of the sharding capabilities of OpenSearch and can scale to billions of vectors at single-digit millisecond latencies by sharding vectors and scale horizontally by adding more nodes. The number of vectors that can fit in a single machine is a function of the off-heap memory availability on the machine. The number of nodes required will depend on the amount of memory that can be used for the algorithm per node and the total amount of memory required by the algorithm. The more nodes, the more memory and better performance. The amount of memory available per node is computed as

`memory_available = (node_memory - jvm_size) * circuit_breaker_limit`, with the following parameters:

- **node_memory** – The total memory of the instance.
- **jvm_size** – The OpenSearch JVM heap size. This is set to half of the instance's RAM, capped at approximately 32 GB.
- **circuit_breaker_limit** – The native memory usage threshold for the circuit breaker. This is set to 0.5.

Total cluster memory estimation depends on total number of vector records and algorithms. HNSW and IVF have different memory requirements. You can refer to [Memory Estimation](#) for more details.

Number of dimensions

OpenSearch's current dimension limit for the vector field `knn_vector` is 16,000 dimensions. Each dimension is represented as a 32-bit float. The more dimensions, the more memory you'll need to index and search. The number of dimensions is usually determined by the embedding models that translate the entity to a vector. There are a lot of options to choose from when building your `knn_vector` field. To determine the correct methods and parameters to choose, refer to [Choosing the right method](#).

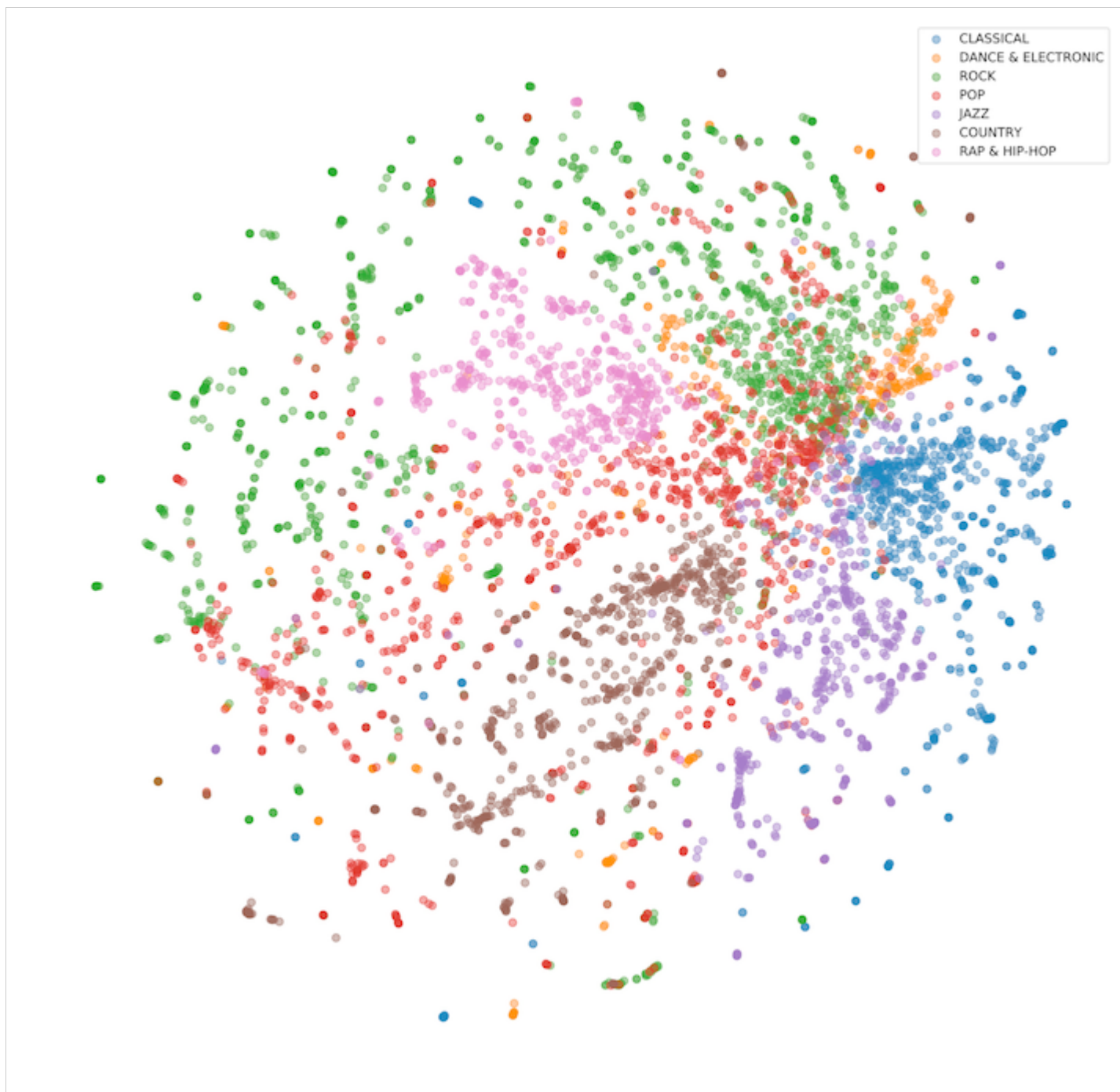
Customer stories:

Amazon Music

Amazon Music is always innovating to provide customers with unique and personalized experiences. One of Amazon Music's approaches to music recommendations is a remix of a classic Amazon innovation, [item-to-item collaborative filtering](#), and vector databases. Using data aggregated based on user listening behavior, Amazon Music has created an embedding model that encodes music tracks and customer representations into a vector space where neighboring vectors represent tracks that are similar. 100 million songs are encoded into vectors, indexed into OpenSearch, and served across multiple geographies to power real-time recommendations. OpenSearch currently manages 1.05 billion vectors and supports a peak load of 7,100 vector queries per second to power Amazon Music recommendations.

The item-to-item collaborative filter continues to be among the most popular methods for online product recommendations because of its effectiveness at scaling to large customer bases and product catalogs. OpenSearch makes it easier to operationalize and further the scalability of the recommender by providing scale-out infrastructure and k-NN indexes that grow linearly with respect to the number of tracks and similarity search in logarithmic time.

The following figure visualizes the high-dimensional space created by the vector embedding.



Brand protection at Amazon

Amazon strives to deliver the world's most trustworthy shopping experience, offering customers the widest possible selection of authentic products. To earn and maintain our customers' trust, we strictly prohibit the sale of counterfeit products, and we continue to invest in innovations that ensure only authentic products reach our customers. Amazon's brand protection programs build trust with brands by accurately representing and completely protecting their brand. We strive to ensure that public perception mirrors the trustworthy experience we deliver. Our brand protection strategy focuses on four pillars: (1) Proactive Controls (2) Powerful Tools to Protect Brands (3) Holding Bad Actors Accountable (4) Protecting and Educating Customers. Amazon OpenSearch

Service is a key part of Amazon's Proactive Controls.

In 2022, Amazon's automated technology scanned more than 8 billion attempted changes daily to product detail pages for signs of potential abuse. Our proactive controls found more than 99% of blocked or removed listings before a brand ever had to find and report it. These listings were suspected of being fraudulent, infringing, counterfeit, or at risk of other forms of abuse. To perform these scans, Amazon created tooling that uses advanced and innovative techniques, including the use of advanced machine learning models to automate the detection of intellectual property infringements in listings across Amazon's stores globally. A key technical challenge in implementing such automated system is the ability to search for protected intellectual property within a vast billion-vector corpus in a fast, scalable and cost effective manner. Leveraging Amazon OpenSearch Service's scalable vector database capabilities and distributed architecture, we successfully developed an ingestion pipeline that has indexed a total of 68 billion, 128- and 1024-dimension vectors into OpenSearch Service to enable brands and automated systems to conduct infringement detection, in real-time, through a highly available and fast (sub-second) search API.

Conclusion

Whether you're building a generative AI solution, searching rich media and audio, or bringing more semantic search to your existing search-based application, OpenSearch is a capable vector database. OpenSearch supports a variety of engines, algorithms, and distance measures that you can employ to build the right solution. OpenSearch provides a scalable engine that can support vector search at low latency and up to billions of vectors. With OpenSearch and its vector DB capabilities, your users can find that 8-foot-blue couch easily, and relax by a cozy fire.

About the Authors



Jon Handler is a Senior Principal Solutions Architect at Amazon Web Services based in Palo Alto, CA. Jon works closely with OpenSearch and Amazon OpenSearch Service, providing help and guidance to a broad range of customers who have search and log analytics workloads that they want to move to the AWS Cloud. Prior to joining AWS, Jon's career as a software developer included four years of coding a large-scale, eCommerce search engine. Jon holds a Bachelor of the Arts from the University of Pennsylvania, and a Master of Science and a Ph. D. in Computer Science and Artificial Intelligence from Northwestern University.



Jianwei Li is a Principal Analytics Specialist TAM at Amazon Web Services. Jianwei provides consultant service for customers to help customer design and build modern data platform. Jianwei has been working in big data domain as software developer, consultant and tech leader.



Dylan Tong is a Senior Product Manager at Amazon Web Services. He leads the product



initiatives for AI and machine learning (ML) on OpenSearch including OpenSearch's vector database capabilities. Dylan has decades of experience working directly with customers and creating products and solutions in the database, analytics and AI/ML domain. Dylan holds a BSc and MEng degree in Computer Science from Cornell University.



Vamshi Vijay Nakkirtha is a Software Engineering Manager working on the OpenSearch Project and Amazon OpenSearch Service. His primary interests include distributed systems. He is an active contributor to various plugins, like k-NN, GeoSpatial, and dashboard-maps.

Comments

