Vector Stores, databases, and embeddings in ColdFusion 2025.1

ColdFusion 2025.1 introduces a unified vector store API that lets CFML developers work with embeddings and vector stores to build semantic search, RAG, recommendation, and other AI-driven features.

Introduction

As Generative AI applications evolve, the need to store and retrieve information based on meaning rather than just keyword matching has become critical. ColdFusion 2025.1introduces native support for Vector stores.

This feature is the backbone of RAG (Retrieval-Augmented Generation) systems. It allows you to:

- Store high-dimensional vectors (numerical representations of text).
- Perform semantic similarity searches.
- Filter results using metadata.

- Integrate smoothly with ColdFusion AI services.

Supported providers

ColdFusion provides a unified API for the following providers:

1. InMemory Vector Store (Best for testing/prototyping)
2. Milvus (Open-source, enterprise-grade)
3. Pinecone (Managed SaaS, high scalability)
4. Qdrant (High-performance)
5. Chroma (Lightweight, developer-friendly)

Quick start: Your first Vector Store

The fastest way to understand vector stores in ColdFusion is to use the InMemory Vector Store. This requires no external setup or API keys.

Step 1: Initialize the Store

Use the VectorStore() built-in function. Calling it without arguments creates an in-memory instance.

// Create an in-memory client

client = VectorStore();

Step 2: Add data

Add documents to your store. If you haven't configured an automatic embedding model, you must provide the vector array manually (for this example, we will assume manual vectors).

```
<cfscript>
    docs = [
        {
            "id": "1",
            "text": "ColdFusion is a rapid application development platform.",
            "vector": [0.1, 0.2, 0.9], // Simplified vector
            "metadata": {"category": "tech"}
        },
        {
            "id": "2",
            "text": "Python is popular for Data Science.",
            "vector": [0.8, 0.1, 0.1],
            "metadata": {"category": "data"}
        }
    ];

    // Add documents and get their IDs back
    addedIds = client.addAll(docs);
    writeDump(addedIds);
</cfscript>
```

Step 3: Search

Perform a similarity search using a query vector.

```
<cfscript>
    // Search for vectors similar to [0.1, 0.2, 0.8]
    results = client.search({
        "vector": [0.1, 0.2, 0.8],
        "topK": 1
    });

    writeDump(results);
    // Output will show the "ColdFusion" document due to vector similarity.
</cfscript>
```

Conceptual overview

What is an embedding?

An embedding is a fixed-length list of real numbers representing a piece of data (for example, a text snippet) in a high-dimensional space where semantically similar items lie close together. In ColdFusion, embeddings can be generated implicitly by configuring an embedding model on a vector store client, so most application code only deals with text and metadata while the platform handles numeric vectors and similarity calculations.

```
<cfscript>
    // In-memory store for demo; in production configure a real provider.
    vs = VectorStore( {
        embeddingModel: "text-embedding-3-small",
        dimension     : 384
    } );

    id = vs.add( {
        text     : "ColdFusion supports semantic search with vector stores.",
        metadata: { category = "intro", language = "en" }
    } );

    writeOutput( "<h3>Inserted item ID:</h3>" );
    writeDump( id );
</cfscript>
```

What is a vector store?

A vector store is a specialized datastore for storing and querying high-dimensional vectors using approximate nearest-neighbor indexes and distance metrics like cosine, Euclidean, or dot-product. ColdFusion supports multiple providers, InMemory, Milvus, Pinecone, Qdrant, and Chroma, so you can choose between open-source, self-hosted, or fully managed services.

```
<cfscript>
    milvusClient = VectorStore( {
        provider       : "milvus",
        url            : "https://127.0.0.1:19530",
        apiKey         : "YOUR_API_KEY",
        collectionName : "cf_intro",
        dimension      : 3,
        metricType     : "COSINE",
        indexType      : "IVF"
    } );

    ids = milvusClient.addAll( [
        {
            id      : "cf-intro-1",
            text    : "ColdFusion is a rapid application development platform.",
            vector  : [0.10, 0.20, 0.30],
            metadata: { topic = "cf", kind = "statement" }
        },
        {
            id      : "cf-intro-2",
            text    : "Vector stores enable semantic search.",
            vector  : [0.11, 0.19, 0.29],
            metadata: { topic = "vector-db", kind = "statement" }
        }
    ] );

    results = milvusClient.search( {
        vector   : [0.10, 0.21, 0.29],
        topK     : 2,
        minScore: 0.0
    } );

</cfscript>
```

What is a vector store in ColdFusion?

A vector store in ColdFusion is the unified VectorStore() API that returns a VectorStoreClient object used for all data-plane operations such as add, addAll, search, delete, deleteAll, listCollections, and deleteCollection. This abstraction hides provider-specific SDKs while enabling seamless migration between backends and consistent semantics across the platform.

```
localStore = VectorStore();

prodStore = VectorStore( {
    provider       : "milvus",
    url            : "https://127.0.0.1:19530",
    apiKey         : "YOUR_API_KEY",
    collectionName : "cf_prod",
    dimension      : 384,
    metricType     : "COSINE",
    indexType      : "HNSW"
} );
```

Core retrieval concepts

Similarity search and distance metrics

Similarity search finds the vectors nearest to a query vector according to a chosen distance metric such as cosine, Euclidean, or dot-product. The search() API accepts topK to control how many neighbors to return and minScore to set a similarity threshold, with provider-specific defaults configurable at the client level.

```
<cfscript>
    vs = VectorStore( {
        dimension : 3,
        metricType: "COSINE"
    } );

    vs.addAll( [
        { text = "ColdFusion vector stores",   vector = [0.10, 0.20, 0.30],
metadata = { tag = "cf" } },
        { text = "Milvus vector store",     vector = [0.11, 0.19, 0.31], metadata
= { tag = "db" } },
        { text = "Completely unrelated text",  vector = [0.90, 0.10, 0.05],
metadata = { tag = "none" } }
    ] );

    results = vs.search( {
        vector  : [0.10, 0.21, 0.29],
        topK    : 2,
        minScore: 0.0
    } );

</cfscript>
```

Dense vs sparse vectors

The initial integration focuses on dense vectors, where most dimensions are non-zero and produced by modern embedding models. Sparse vectors for lexical or hybrid search can be supported by specific index types in some providers and are considered for future configuration-driven enhancements rather than a separate CF API.

```
<cfscript>
    vs = VectorStore( { dimension: 4, metricType: "COSINE" } );

    ids = vs.addAll( [
        { text = "dense vector example 1", vector = [0.1, 0.3, 0.2, 0.4],
metadata = { type = "dense" } },
        { text = "dense vector example 2", vector = [0.2, 0.1, 0.4, 0.3],
metadata = { type = "dense" } }
    ] );

    res = vs.search( { vector: [0.15, 0.25, 0.25, 0.35], topK: 2, minScore:
0.0 } );

</cfscript>
```

Metadata and filtering semantics

Each vector store item can include a metadata struct, which is attached at write time and used later for filtering, categorization, and context. ColdFusion standardizes metadata filtering with a MongoDB-style syntax supporting operators such as eq, ne, gt, gte, lt, lte, in, nin, and, or, and not, regardless of the underlying provider.

```
<cfscript>
    vs = VectorStore( { dimension: 3, metricType: "COSINE" } );

    vs.addAll( [
        {
            text    : "CF 2026 vector stores documentation.",
            vector  : [0.10, 0.20, 0.30],
            metadata: { product = "coldfusion", year = 2026, channel = "docs" }
        },
        {
            text    : "CF 2025 performance updates.",
            vector  : [0.12, 0.21, 0.28],
            metadata: { product = "coldfusion", year = 2025, channel = "blog" }
        },
        {
            text    : "Generic web development article.",
            vector  : [0.50, 0.60, 0.70],
            metadata: { product = "generic",    year = 2024, channel = "blog" }
        }
    ] );

    filter = {
        and = [
            { product = { eq  = "coldfusion" } },
            { year    = { gte = 2025 } }
        ]
    };

    results = vs.search( {
        vector  : [0.11, 0.22, 0.29],
        topK    : 10,
        minScore: 0.0,
        filter  : filter
    } );


</cfscript>
```

ColdFusion vector store architecture

Unified VectorStore API and client abstraction

VectorStore() is a built-in function that constructs a VectorStoreClient based on a configuration struct or uses an in-memory implementation if no arguments are provided. The client encapsulates all data-plane behavior while configuration can be defined at module, application, or server scope, and reused via aliases.

```
<cfscript>
    function VectorStore( string env = "dev" ) {
        if ( env == "dev" ) {
            return VectorStore();
        }

        return VectorStore( {
            provider       : "milvus",
            url            : "https://127.0.0.1:19530",
            apiKey         : "YOUR_API_KEY",
            collectionName : "cf_env_example",
            dimension      : 384,
            metricType     : "COSINE",
            indexType      : "HNSW"
        } );
    }

</cfscript>
```

Supported providers and in-memory store

ColdFusion supports five vector stores: InMemory, Milvus, Pinecone, Qdrant, and Chroma, all accessed via the same CFML API. The in-memory store is a simple non-persistent implementation for developer testing and prototyping; it uses process memory only and is not suitable for production because data is lost on restart and cannot be scaled independently.

```
<cfscript>
        function createClientForProvider( string providerName ) {
        baseConfig = {
            provider      : providerName,
            collectionName: "cf_switch_demo",
            dimension     : 384,
            metricType    : "COSINE",
            indexType     : "HNSW"
        };

        switch( lcase( providerName ) ) {
            case "milvus":
                baseConfig.url    = "https://127.0.0.1:19530";
                baseConfig.apiKey = "MILVUS_KEY";
                break;
            case "pinecone":
                baseConfig.url    = "https://your-project.svc.pinecone.io";
                baseConfig.apiKey = "PINECONE_KEY";
                break;
        }

        return VectorStore( baseConfig );
    }

    client = createClientForProvider( "milvus" );

</cfscript>
```

Connection and collection configuration model

Configuration is split into connection settings (URL, authentication, retries, timeouts, keep-alive behavior, connection pools) and collection settings (collection/database names, dimension, metric type, index type, field mappings). Collections are generally created automatically when the client is initialized, minimizing boilerplate while still allowing per-provider tuning.

```
<cfscript>
    client = VectorStore( {
        provider          : "milvus",
        url               : "https://127.0.0.1:19530",
        apiKey            : "YOUR_API_KEY",
        databaseName      : "default",
        collectionName    : "cf_config_demo",
        dimension         : 128,
        metricType        : "COSINE",
        indexType         : "IVF",
        idFieldName       : "id",
        textFieldName     : "text",
        vectorFieldName   : "vector",
        metadataFieldName: "metadata",
        callTimeout       : 10000,
        connectTimeout    : 10000,
        maxRetries        : 3,
        retryOnRateLimit : true
    } );

</cfscript>
```

Work with embeddings in ColdFusion

Configure embedding models on a client

Specifying an embedding model when creating a client lets ColdFusion generate embeddings automatically for both stored content and text queries. Explicit vectors still take precedence if provided, giving flexibility for mixed or migration scenarios.

```
<cfscript>
    client = VectorStore( {
        provider       : "milvus",
        url            : "https://127.0.0.1:19530",
        apiKey         : "YOUR_API_KEY",
        collectionName : "cf_embed_demo",
        dimension      : 384,
        metricType     : "COSINE",
        indexType      : "IVFFLAT",
        embeddingModel : "text-embedding-3-small"
    } );

    ids = client.addAll( [
        { text = "ColdFusion integrates with vector stores.", metadata =
{ section = "overview" } },
        { text = "Vector stores power semantic search and RAG.", metadata =
{ section = "concepts" } }
    ] );

</cfscript>
```

Text-only workflows (implicit embeddings)

With an embedding model configured, applications can use text-only search() calls, and the platform embeds the query and performs similarity search behind the scenes. This flow is ideal for knowledge bases, chatbots, and RAG implementations where requests and content are primarily textual.

```
<cfscript>
   client = VectorStore( {
        provider       : "milvus",
        url            : "https://127.0.0.1:19530",
        apiKey         : "YOUR_API_KEY",
        collectionName : "cf_text_search_demo",
        dimension      : 384,
        metricType     : "COSINE",
        indexType      : "IVFFLAT",
        embeddingModel : "text-embedding-3-small",
        topK           : 5,
        minScore       : 0.4
    } );

    client.addAll( [
        { text = "ColdFusion vector stores provide a unified API over multiple 
providers.", metadata = { topic = "cf-vector" } },
        { text = "Retrieval-Augmented Generation uses a vector store as a 
retriever.",      metadata = { topic = "rag" } }
    ] );

    results = client.search( {
        text: "How does ColdFusion support RAG?"
    } );

</cfscript>
```

Explicit vector workflows

For offline workflows or external embedding pipelines, applications can supply explicit numeric vectors for items and queries. In this model, ColdFusion does not call an embedding model and assumes vector dimensionality and semantics are handled by the caller.

```
<cfscript>
    vs = VectorStore( {
        dimension : 3,
        metricType: "COSINE"
    } );

    ids = vs.addAll( [
        {
            id      : "precomp-1",
            text    : "Pre-computed embedding for CF documentation.",
            vector  : [0.10, 0.20, 0.30],
            metadata: { source = "batch-job", kind = "doc" }
        },
        {
            id      : "precomp-2",
            text    : "Pre-computed embedding for AI article.",
            vector  : [0.40, 0.50, 0.60],
            metadata: { source = "batch-job", kind = "article" }
        }
    ] );

    results = vs.search( {
        vector  : [0.11, 0.19, 0.29],
        topK    : 2,
        minScore: 0.0
    } );
</cfscript>
```

Data operations in a vector store

Add and batch items

add() inserts a single item and returns its ID, while addAll() accepts an array of items and returns an array of IDs. Batching large writes reduces network overhead and works in conjunction with internal batchSize parameters to respect provider limits such as gRPC message sizes.

```
<cfscript>
    vs = VectorStore( { dimension: 3, metricType: "COSINE" } );

    id1 = vs.add( {
        text    : "Single insert example.",
        vector  : [0.10, 0.20, 0.30],
        metadata: { batch = "single" }
    } );

    ids = vs.addAll( [
        { text = "Batch insert 1.", vector = [0.11, 0.21, 0.31], metadata =
{ batch = "multi" } },
        { text = "Batch insert 2.", vector = [0.12, 0.22, 0.32], metadata =
{ batch = "multi" } }
    ] );

</cfscript>
```

Delete by ID, filter, or full clear

delete(id) removes a single item by its ID, while deleteAll() supports three modes: delete by list of IDs, delete by metadata filter, or delete all items when called with no arguments. Filter-based deletion is useful when IDs are not tracked externally and you want to purge items based on tags, version, or environment.

Query vector stores

Text-based similarity search

Text-based similarity search uses the client's configured embedding model to convert the query into a vector, then finds the nearest neighbors in the store. Defaults for topK and minScore can be set on the client for consistent semantics across your app.

```
<cfscript>
    client = VectorStore( {
        provider       : "milvus",
        url            : "https://127.0.0.1:19530",
        apiKey         : "YOUR_API_KEY",
        collectionName : "cf_text_query_demo",
        dimension      : 384,
        metricType     : "COSINE",
        indexType      : "IVFFLAT",
        embeddingModel : "text-embedding-3-small",
        topK           : 3,
        minScore       : 0.3
    } );

    client.addAll( [
        { text = "ColdFusion has native support for vector stores.", metadata =
{ section = "intro" } },
        { text = "Vector stores are key for RAG systems.",        metadata =
{ section = "rag" } }
    ] );

    results = client.search( {
        text: "How do vector stores help RAG?"
    } );


</cfscript>
```

Vector-based similarity search

Vector-based queries send a numeric vector directly to the search() method, bypassing any embedding model. This mode is common when embeddings are pre-computed elsewhere or when testing low-level behavior.

```
<cfscript>
    // In-memory store for demo; in production configure a real provider.
    vs = VectorStore( {
        embeddingModel: "text-embedding-3-small",
        dimension      : 384
    } );

    id = vs.add( {
        text     : "ColdFusion supports semantic search with vector stores.",
        metadata: { category = "intro", language = "en" }
    } );

    writeOutput( "<h3>Inserted item ID:</h3>" );
    writeDump( id );
</cfscript>



<cfscript>
    // 1. Define the Vector Store Client (assuming it's already configured and
connected)
    // This client configuration must match the dimension of your search vector
    vsClient = VectorStore({
        provider: "milvus",
        url: "https://milvus.example.com",
        apiKey: "YOUR_API_KEY",
        collectionName: "knowledge_base",
        dimension: 1536, // IMPORTANT: Must match the vector dimension
        // NOTE: embeddingModel is omitted here, requiring the vector parameter
in search()
    });
    // 2. Define the Query Vector
    // This vector would typically be generated by an external embedding model
    // based on the user's search query text.
    queryVector = [
        0.015, 0.022, 0.031, // ... 1533 more float values ...
        0.045, 0.051, 0.063
    ];

    // 3. Perform the Search
    results = vsClient.search({
        // Use the 'vector' parameter for a raw vector search
        vector: queryVector,

        // Optional: Specify the number of results to retrieve
        topK: 5,

        // Optional: Set a minimum similarity score threshold (0.0 to 1.0)
        minScore: 0.8,

        // Optional: Apply metadata filtering to narrow the search scope
        filter: {
            category: "API_Reference",
            status: { notEquals: "deprecated" }
        }
    });
</cfscript>
```

Top-K, score thresholds, and result shapes

topK and minScore tune the balance between recall and precision: higher topK returns more candidates, while higher minScore filters weaker matches. Search results are arrays of item structs with fields like id, text, metadata, and optionally vector, depending on provider and configuration.

```
<cfscript>
    vs = VectorStore( { dimension: 3, metricType: "COSINE" } );

    vs.addAll( [
        { text = "High similarity sample.",   vector = [0.10, 0.20, 0.30],
metadata = { scoreHint = "high" } },
        { text = "Medium similarity sample.", vector = [0.20, 0.10, 0.25],
metadata = { scoreHint = "medium" } },
        { text = "Low similarity sample.",    vector = [0.90, 0.05, 0.10], 
metadata = { scoreHint = "low" } }
    ] );

    query = [0.11, 0.21, 0.29];

    strictResults = vs.search( {
        vector  : query,
        topK    : 3,
        minScore: 0.7
    } );

    relaxedResults = vs.search( {
        vector  : query,
        topK    : 3,
        minScore: 0.0
    } );

</cfscript>
```

Collection management

What is a collection?

A **collection** in the AI Vector store is the main logical container that holds your vectors and documents, similar to how a table works in a relational database or an index in a search engine. Each collection has a unique name or ID (for example, `support_kb`) and is created with a fixed embedding dimension (such as 768 or 1536), which all stored vectors must match.

Within a collection, you store many documents, and each document has a unique `id`, its `text` (or other payload), an `embedding` (the numeric vector), and optional `metadata` (such as tenant, language, or tags). All similarity searches run within a specific collection, when you query `support_kb`, only documents in that collection are considered. Collections are important because they give you isolation and structure: you can separate different content domains (support articles vs. product docs), enforce different access or tenancy rules, and tune indexing or model choices per collection. In short, a collection is your named, isolated bucket of vectors and documents that you query as a unit.

Create a collection

When you initialize a client with collection parameters, ColdFusion automatically creates the collection if it does not already exist in the underlying vector store. This hides provider-specific DDL while giving you ready-to-use collections for data operations.

```
<cfscript>
    vs = VectorStore( { dimension: 3, metricType: "COSINE" } );

    vs.addAll( [
        { text = "Vector-based query example 1.", vector = [0.10, 0.20, 0.30], 
metadata = { label = "a" } },
        { text = "Vector-based query example 2.", vector = [0.40, 0.50, 0.60], 
metadata = { label = "b" } }
    ] );

    queryVector = [0.11, 0.19, 0.31];

    results = vs.search( {
        vector   : queryVector,
        topK     : 2,
        minScore: 0.0
    } );

    client = VectorStore( {
        provider      : "milvus",
        url           : "https://127.0.0.1:19530",
        apiKey        : "YOUR_API_KEY",
        collectionName: "cf_auto_create",
        dimension     : 3,
        metricType    : "COSINE",
        indexType     : "IVF"
    } );

    id = client.add( {
        text    : "First document in auto-created collection.",
        vector  : [0.10, 0.20, 0.30],
        metadata: { createdBy = "demo" }
    } );
</cfscript>
```

List and delete collections

listCollections() returns the names of available collections, while deleteCollection(name) drops a collection and its associated vector data. These APIs are useful for environment setup/teardown, integration tests, and admin tooling.

```
<cfscript>
    client = VectorStore( {
        provider      : "milvus",
        url           : "https://127.0.0.1:19530",
        apiKey        : "YOUR_API_KEY",
        collectionName: "cf_temp_collection",
        dimension     : 3,
        metricType    : "COSINE",
        indexType     : "IVF"
    } );

    client.add( {
        text     : "Temporary data.",
        vector   : [0.10, 0.20, 0.30],
        metadata: { type = "temp" }
    } );

    collectionsBefore = client.listCollections();

    client.deleteCollection( "cf_temp_collection" );

    collectionsAfter = client.listCollections();

</cfscript>
```

Environment and multi-tenant patterns

By parameterizing collection names and other configuration values, you can isolate environments (dev/QA/prod) or tenants (per customer) on separate collections in the same backend. This keeps data boundaries clear while still sharing infrastructure resources.

```
<cfscript>
    function getEnvCollectionName( string env ) {
        return "cf_docs_" & lcase( env );
    }

    env      = "dev";
    collName = getEnvCollectionName( env );

    client = VectorStore( {
        provider      : "milvus",
        url           : "https://127.0.0.1:19530",
        apiKey        : "YOUR_API_KEY",
        collectionName: collName,
        dimension     : 3,
        metricType    : "COSINE",
        indexType     : "IVF"
    } );

    client.add( {
        text    : "Env-specific document for " & env,
        vector  : [0.10, 0.20, 0.30],
        metadata: { env = env }
    } );

</cfscript>
```

Error handling and observability

VectorStoreException hierarchy

All vector store operations use a common exception hierarchy, with VectorStoreException as the base and specific subclasses such as:

- VectorStoreItemAdditionException
- VectorStoreItemDeletionException
- VectorStoreItemSearchException
- FieldValidationException
- UnsupportedMetricTypeException
- UnsupportedIndexTypeException
- VectorStoreConnectionException,
- VectorDimensionMismatchException
- VectorStoreInvalidFilterException

These exceptions carry type, message, root cause, and stack trace to support diagnosis.

```
<cfscript>
```

```
    try {
        vs = VectorStore( { dimension: 3, metricType: "COSINE" } );
        vs.add( {
            text     : "Bad vector dimension.",
            vector   : [0.10, 0.20],
            metadata: { test = "dimensionError" }
        } );
    }
    catch( any e ) {
        if ( findNoCase( "VectorDimensionMismatchException", e.type ) ) {
            writeOutput( "<p>Vector dimension mismatch detected.</p>" );
        } else if ( findNoCase( "VectorStore", e.type ) ) {
            writeOutput( "<p>General VectorStore-related error.</p>" );
        }
    }

</cfscript>
```

Logging

Provider-specific logs (for Milvus, Pinecone, Chroma, Qdrant) capture backend errors, retries, and connection details, while in-memory and factory-level messages are written to application.log. Application-level logging can wrap vector operations to record failures and correlate them with CFML requests.

```
<cfscript>
    // Configure logging for vector operations
    import org.apache.logging.log4j.*;

    logger = LoggerFactory.getLogger("VectorOperations");

    try {
        startTime = now();

        vs = VectorStore({
            provider      : "milvus",
            url           : "https://127.0.0.1:19530",
            apiKey        : "YOUR_API_KEY",
            dimension     : 3,
            metricType    : "COSINE"
        });

        logger.info("VectorStore client created successfully");
        // Add document with logging
        try {
            vs.add({
                text     : "Sample document for logging demonstration",
                vector   : [0.1, 0.2, 0.3],
                metadata : { source = "demo", version = 1 }
            });
            logger.info("Document added successfully");
        } catch (VectorStoreItemAdditionException e) {
            logger.error("Failed to add document: " & e.message, e);
            logger.debug("Error details: " & serializeJSON(e));
        }

        // Search with logging
        try {
            results = vs.search({
                text     : "sample search",
                limit    : 5,
                threshold: 0.7
            });
            logger.info("Search completed. Found " & arrayLen(results) & "
results");
        } catch (VectorStoreItemSearchException e) {
            logger.error("Search failed: " & e.message, e);
        }

        elapsed = dateDiff("s", startTime, now());
        logger.info("VectorStore operation completed in " & elapsed & "
seconds");

    } catch (VectorStoreConnectionException e) {
        logger.error("Connection error: " & e.message, e);
        logger.warn("Vector store may be unavailable");
    } catch (any e) {
        logger.error("Unexpected error during vector operations: " & e.message,
e);
    }

    // Application logs in application.log contain all operation details
</cfscript>
```

Common failure scenarios and mitigation

Typical issues include invalid configuration (wrong URL/API key), wrong vector dimension, unsupported metric or index type, and malformed filters, each mapping to a specific exception type. Validating configuration up front and catching vector store exceptions at boundaries helps maintain robustness in production.

```
<cfscript>
    try {
        // Validate configuration before client creation
        requiredParams = ['url', 'apiKey', 'dimension', 'metricType'];
        requiredValues = [url, apiKey, dimension, metricType];

        for (param in requiredParams) {
            if (!isDefined(requiredValues[arrayFind(requiredParams, param)])  ||
requiredValues[arrayFind(requiredParams, param)] == "") {
                throw(type="ConfigurationException", message="Missing required
parameter: " & param);
            }
        }

        // Verify valid metric type
        validMetrics = ['COSINE', 'EUCLIDEAN', 'DOTPRODUCT'];
        if (!arrayContains(validMetrics, metricType)) {
            throw(type="UnsupportedMetricTypeException", message="Unsupported
metric type: " & metricType);
        }

        // Verify valid index type
        validIndexTypes = ['IVF', 'HNSW', 'FLAT'];
        if (!arrayContains(validIndexTypes, indexType)) {
            throw(type="UnsupportedIndexTypeException", message="Unsupported
index type: " & indexType);
        }

        vs = VectorStore({
            provider     : "milvus",
            url          : url,
            apiKey       : apiKey,
            dimension    : dimension,
            metricType   : metricType,
            indexType    : indexType
        });
        writeOutput("<p>VectorStore configured successfully.</p>");

    } catch (VectorStoreDimensionMismatchException e) {
        writeOutput("<p>Vector dimension mismatch error: " & e.message & "</p>");
        // Log and handle gracefully

    } catch (UnsupportedMetricTypeException e) {
        writeOutput("<p>Unsupported metric type error: " & e.message & "</p>");
        // Log and handle gracefully

    } catch (ConfigurationException e) {
        writeOutput("<p>Configuration error: " & e.message & "</p>");
        // Log and handle gracefully

    } catch (any e) {
        writeOutput("<p>Unexpected VectorStore error: " & e.message & "</p>");
        // Log exception details for debugging
    }
</cfscript>
```

Performance and scalability

Batching and large documents

To avoid gRPC message size and memory issues when ingesting large content (for example, 100 MB documents), the spec recommends chunking documents and embedding them in batches, with a batch size default around 1000 items. This approach avoids latency spikes and memory pressure while sustaining throughput.

```
<cfscript>
    // Function to chunk large documents
    function chunkDocument(document, chunkSize = 1000) {
        chunks = [];
        words = listToArray(document, " ");
        currentChunk = "";

        for (word in words) {
            if (len(currentChunk) + len(word) + 1 > chunkSize) {
                arrayAppend(chunks, trim(currentChunk));
                currentChunk = word;
            } else {
                currentChunk = currentChunk & " " & word;
            }
        }

        if (len(currentChunk) > 0) {
            arrayAppend(chunks, trim(currentChunk));
        }

        return chunks;
    }

    // Large document ingestion with batch processing
    vs = VectorStore({
        provider     : "milvus",
        url          : "https://127.0.0.1:19530",
        apiKey       : "YOUR_API_KEY",
        dimension    : 768,
        metricType   : "COSINE"
    });

    largeDocument = "[Very large text content here - 100+ MB document...]";
    batchSize = 1000;  // Default batch size to avoid gRPC issues

    // Chunk the document
    documentChunks = chunkDocument(largeDocument);
    writeOutput("<p>Document chunked into " & arrayLen(documentChunks) & " 
chunks.</p>");

    // Process in batches
    for (i = 1; i <= arrayLen(documentChunks); i += batchSize) {
        batchEnd = min(i + batchSize - 1, arrayLen(documentChunks));
        batch = arraySlice(documentChunks, i, batchEnd - i + 1);

        try {
            batchItems = [];
            for (chunk in batch) {
                // Generate embedding for each chunk
                embedding = generateEmbedding(chunk);
                arrayAppend(batchItems, {
                    text      : chunk,
                    vector    : embedding,
                    metadata : { chunkIndex = i, totalChunks =
arrayLen(documentChunks) }
                });
            }

            vs.add(batchItems);
            writeOutput("<p>Batch " & ceiling(i / batchSize) & "
processed.</p>");

        } catch (VectorStoreItemAdditionException e) {
            writeOutput("<p>Error processing batch: " & e.message & "</p>");
        }

        sleep(100);
    }

    writeOutput("<p>Document ingestion completed.</p>");
</cfscript>
```

Index types and tuning

Index types determine how a vector store organizes and searches high-dimensional data for fast, scalable similarity queries. The choice of index type affects ingestion speed, memory usage, and query latency. ColdFusion's vector store API supports explicit index type selection and validates your choice, throwing an UnsupportedIndexTypeException if an invalid value is provided.

Common Index Types

IVF (Inverted File Index)

- Purpose: Fast approximate nearest neighbor search, ideal for large datasets.
- How it works: Divides the vector space into clusters ("lists"). During search, only a subset of clusters is probed, reducing computation.

HNSW (Hierarchical Navigable Small World)

- Purpose: Excellent for low-latency queries, especially with large datasets.
- How it works: Builds a graph structure for efficient navigation and search.

IVF_FLAT

- Purpose: Brute-force search, compares every vector.

- How it works: No indexing, just linear scan.
- Use case: Small datasets, testing, or when absolute accuracy is needed.

How to choose an Index Type

- IVF: Use for large-scale ingestion and when you need fast, approximate search.
- HNSW: Use for real-time, low-latency queries and when you want high recall.
- IVF_FLAT: Use for small datasets or when you need exact results.

```
<cfscript>
    // Index type tuning based on workload requirements
    // IVF: Fast approximate nearest neighbor search, good for large datasets
    function createIVFIndex(configuration) {
        return VectorStore({
            provider      : configuration.provider,
            url           : configuration.url,
            apiKey        : configuration.apiKey,
            dimension     : configuration.dimension,
            metricType    : configuration.metricType,
            indexType     : "IVF",
            indexParams   : {
                nlist         : 1024,      // Number of clusters
                nprobe        : 32         // Clusters to probe during search
            }
        });
    }

    // HNSW: Hierarchical navigable small-world, excellent for low-latency
queries
    function createHNSWIndex(configuration) {
        return VectorStore({
            provider      : configuration.provider,
            url           : configuration.url,
            apiKey        : configuration.apiKey,
            dimension     : configuration.dimension,
            metricType    : configuration.metricType,
            indexType     : "HNSW",
            indexParams   : {
                M             : 16,        // Max connections per element
                efConstruction: 200,       // Size of dynamic list during
insertion
```

```
                ef            : 64         // Size of dynamic list during search
            }
        });
    }

    // Select index type based on workload
    config = {
        provider  : "milvus",
        url       : "https://127.0.0.1:19530",
        apiKey    : "YOUR_API_KEY",
        dimension : 768,
        metricType: "COSINE"
    };

    // For high-throughput ingestion scenarios
    if (workloadType == "ingestion-heavy") {
        try {
            vs = createIVFIndex(config);
            writeOutput("<p>IVF index created for high-throughput
ingestion.</p>");
        } catch (UnsupportedIndexTypeException e) {
            writeOutput("<p>IVF index not supported: " & e.message & "</p>");
        }
    }
    // For low-latency query scenarios
    else if (workloadType == "query-latency-sensitive") {
        try {
            vs = createHNSWIndex(config);
            writeOutput("<p>HNSW index created for low-latency queries.</p>");
        } catch (UnsupportedIndexTypeException e) {
            writeOutput("<p>HNSW index not supported: " & e.message & "</p>");
        }
    }
</cfscript>
```

Latency, throughput, and retry policies

Performance tuning includes setting retry policies (max retries, exponential backoff), call and connection timeouts, keep-alive behavior, and connection pooling on providers that support it. The spec emphasizes profiling under expected load and using provider features to maintain sub-second query latency.

```
<cfscript>
```

```
// Performance tuning with retry policies and connection pooling
vs = VectorStore({
    provider            : "milvus",
    url                 : "https://127.0.0.1:19530",
    apiKey              : "YOUR_API_KEY",
    dimension           : 768,
    metricType          : "COSINE",

    // Connection timeout configuration
    connectionTimeout  : 5000,     // 5 seconds
    callTimeout         : 10000,    // 10 seconds for API calls
    keepAlive           : true,     // Keep connection alive

    // Retry policy with exponential backoff
    retryPolicy         : {
        maxRetries      : 3,
        backoffMultiplier: 2.0,
        initialDelayMs  : 100,
        maxDelayMs      : 5000
    },

    // Connection pooling
    connectionPooling  : {
        corePoolSize    : 5,
        maxPoolSize     : 20,
        maxIdleTime     : 60000    // 60 seconds
    }
});

// Function to execute search with retry and timing profiling
function performSearch(vs, query, maxRetries = 3) {
    startTime = getTickCount();
    retryCount = 0;

    while (retryCount < maxRetries) {
        try {
            results = vs.search({
                text      : query,
                limit     : 10,
                threshold : 0.7
            });

            elapsed = getTickCount() - startTime;
```

```
                writeOutput("<p>Search completed in " & elapsed & "ms (" &
retryCount & " retries)</p>");
                return results;

            } catch (VectorStoreConnectionException e) {
                retryCount += 1;
                if (retryCount < maxRetries) {
                    delay = 100 * pow(2, retryCount - 1);  // Exponential backoff
                    sleep(delay);
                } else {
                    writeOutput("<p>Search failed after " & maxRetries & "
retries: " & e.message & "</p>");
                    throw e;
                }
            }
        }
    }

    // Execute search with configured retry and timeout settings
    try {
        results = performSearch(vs, "semantic search query");
        writeOutput("<p>Retrieved " & arrayLen(results) & " results within sub-
second latency.</p>");
    } catch (any e) {
        writeOutput("<p>Search operation failed: " & e.message & "</p>");
    }
</cfscript>
```

Advanced topics

Hybrid search (dense + sparse) concepts

Hybrid search combines dense semantic vectors with sparse lexical features, either via a single hybrid index or separate indexes merged at query time. Because not all supported vector stores have first-class hybrid capabilities, ColdFusion defers direct hybrid support and will treat it as a future configuration-driven enhancement.

```
<cfscript>
    // Hybrid search implementation combining dense and sparse vectors
    // Note: Current ColdFusion treats this as application-level hybrid search

    VectorStore = VectorStore({
        provider     : "milvus",
        url          : "https://127.0.0.1:19530",
        apiKey       : "YOUR_API_KEY",
        dimension    : 768,
        metricType   : "COSINE"
    });

    // Function to generate sparse lexical features (BM25-like scoring)
    function generateSparseFeatures(text) {
        words = listToArray(lcase(text), " ");
        sparseVector = {};

        for (word in words) {
            word = trim(word);
            if (len(word) > 2) {  // Skip very short words
                if (isDefined(sparseVector[word])) {
                    sparseVector[word] += 1;
                } else {
                    sparseVector[word] = 1;
                }
            }
        }

        return sparseVector;
    }

    // Function to execute hybrid search
    function hybridSearch(query, denseWeight = 0.7, sparseWeight = 0.3) {
        // Dense vector search
        denseResults = VectorStore.search({
            text      : query,
            limit     : 20,
            threshold : 0.5
        });

        // Sparse lexical search
        sparseFeatures = generateSparseFeatures(query);

        // Merge and rank results
        mergedResults = [];
        scoreMap = {};

        // Apply dense score weight
        for (result in denseResults) {
            scoreMap[result.id] = (result.score * denseWeight);
        }

        // Apply sparse score weight
        for (word in sparseFeatures) {
            for (result in denseResults) {
                if (findNoCase(word, result.text)) {
                    if (isDefined(scoreMap[result.id])) {
                        scoreMap[result.id] += (0.1 * sparseWeight);
                    } else {
                        scoreMap[result.id] = (0.1 * sparseWeight);
                    }
                }
            }
        }

        // Sort by combined score
        sortedScores = structSort(scoreMap, "numeric", "desc");

        return {
            results      : denseResults,
            hybridScores : sortedScores,
            denseWeight  : denseWeight,
            sparseWeight : sparseWeight
        };
    }
</cfscript>
```

Asynchronous and concurrent ingestion patterns

Current Java clients do not yet offer asynchronous APIs, so asynchronous ingestion is managed at the CF level using patterns like cfthread while still relying on synchronous client calls. The spec notes async support as a future enhancement, and recommends batching and connection tuning in the meantime.

```
<cfscript>
        vs = VectorStore({
            provider      : "milvus",
            url           : "https://127.0.0.1:19530",
            apiKey        : "YOUR_API_KEY",
            dimension     : 768,
            metricType    : "COSINE"
        });
        // Queue for managing asynchronous ingestion tasks
        ingestionQueue = [];
        queueMutex = createObject("java",
"java.util.concurrent.locks.ReentrantReadWriteLock");

        // Function to add items to ingestion queue
        function queueIngestionTask(items) {
            lock = queueMutex.writeLock();
            lock.lock();
            try {
                arrayAppend(ingestionQueue, items);
            } finally {
                lock.unlock();
            }
        }

        // Background worker thread for concurrent ingestion
        cfthread(name="vectorIngestionWorker") {
            while (true) {
                lock = queueMutex.readLock();
                lock.lock();
                hasItems = arrayLen(ingestionQueue) > 0;
                lock.unlock();
                if (hasItems) {
                lock = queueMutex.writeLock();
                lock.lock();
                try {
                    if (arrayLen(ingestionQueue) > 0) {
                        batch = arrayShift(ingestionQueue);

                        try {
                            vs.add(batch);
                            writeOutput("<p>Batch ingested asynchronously.</p>");
                        } catch (VectorStoreItemAdditionException e) {
                            writeOutput("<p>Async ingestion failed: " & e.message
& "</p>");
                            // Re-queue for retry
                            arrayPrepend(ingestionQueue, batch);
                            sleep(5000);
                        }
                    }
                } finally {
                    lock.unlock();
                }
            } else {
                sleep(100);  // Check queue every 100ms
            }
        }
    };

    // Main thread: queue documents concurrently
    for (i = 1; i <= 10; i++) {
        cfthread(name="ingestionTask_#i#") {
            documentBatch = [];
            for (j = 1; j <= 100; j++) {
                arrayAppend(documentBatch, {
                    text     : "Document " & (i * 100 + j),
                    vector   : [0.1, 0.2, 0.3],
                    metadata : { batch = i, index = j }
                });
            }
            queueIngestionTask(documentBatch);
            writeOutput("<p>Batch " & i & " queued for ingestion.</p>");
        }
    }

</cfscript>
```

Integrate vector stores into RAG and AI workflows

In Retrieval-Augmented Generation, the vector store holds embeddings for your knowledge base and acts as the retriever that supplies context snippets to LLM prompts. ColdFusion's vector store feature is designed to plug into platform AI services so embeddings, retrieval, and model calls can be orchestrated from CFML much like frameworks such as LangChain do in other ecosystems.

```
<cfscript>
// Initialize vector store for knowledge base
    VectorStore = VectorStore({
        provider      : "milvus",
        url           : "https://127.0.0.1:19530",
        apiKey        : "YOUR_API_KEY",
        dimension     : 768,
        metricType    : "COSINE"
    });
    // Initialize LLM service
    llmService = createObject("component", "com.example.LLMService").init("gpt-
4");

    // RAG: Retrieval-Augmented Generation function
    function ragWorkflow(userQuery) {
        // Step 1: Retrieve relevant context from vector store
        retrievedContext = VectorStore.search({
            text      : userQuery,
            limit     : 5,
            threshold : 0.6
        });

        // Step 2: Construct context from results
        contextSnippets = [];
        for (doc in retrievedContext) {
            arrayAppend(contextSnippets, doc.text);
        }

        context = arrayToList(contextSnippets, chr(10) & chr(10));

        // Step 3: Build augmented prompt with context
        systemPrompt = "You are a helpful assistant. Use the following context to
answer questions accurately.";
        augmentedPrompt = systemPrompt & chr(10) & chr(10) & "Context:" & chr(10)
& context & chr(10) & chr(10) & "Question: " & userQuery;

        // Step 4: Call LLM with augmented prompt
        try {
            llmResponse = llmService.generateCompletion({
                prompt      : augmentedPrompt,
                maxTokens   : 500,
                temperature : 0.7
            });

            // Step 5: Return response with source references
            return {
                answer    : llmResponse.text,
                sources   : retrievedContext,
                timestamp : now()
            };
        } catch (any e) {
            return {
                error   : e.message,
                answer  : "Unable to process query at this time.",
                sources : []
            };
        }
    }

    // Execute RAG workflow
    userQuestion = "What is the capital of France?";
    ragResult = ragWorkflow(userQuestion);

    writeOutput("<h3>Question:</h3><p>" & userQuestion & "</p>");
    writeOutput("<h3>Answer:</h3><p>" & ragResult.answer & "</p>");
    writeOutput("<h3>Sources Retrieved:</h3><ul>");
    for (source in ragResult.sources) {
        writeOutput("<li>" & source.text & " (Score: " &
numberFormat(source.score, "0.00") & ")</li>");
    }
    writeOutput("</ul>");
</cfscript>
```