ColdFusion RAG Developer Guide

Retrieval-Augmented Generation (RAG) in ColdFusion

This guide helps developers build RAG-based applications using ColdFusion. It uses progressive disclosure: start with the overview and getting started, then expand sections for deeper detail.

Contents

1. Introduction

What is RAG?

Retrieval-Augmented Generation (RAG) combines two capabilities:

- **Retrieval**: Finding relevant pieces of text (chunks) from your documents using semantic search.
- **Generation**: Using a large language model (LLM) to produce an answer based on those chunks and the user's question.

RAG reduces hallucinations and keeps answers grounded in your own content (docs, policies, knowledge bases).

What does ColdFusion provide?

ColdFusion's RAG support gives you:

- **Simple RAG**: A single function call with two required parameters (source and model) and sensible defaults. Ideal for getting started quickly.
- **Standalone document processing**: Load, split, and transform documents without running a full RAG pipeline, useful for testing or custom pipelines.

How RAG fits in the ColdFusion AI stack

RAG in ColdFusion integrates with:

- **Chat models** (e.g., OpenAI, Azure OpenAI, Google Gemini, Anthropic, Mistral, Ollama).
- **Embedding models** for turning text into vectors.
- **Vector stores** (in-memory, Milvus, and Qdrant) for storing and searching embeddings.

You use the same model configuration patterns you already use for `chatModel()` and related AI features.

---

2. Problem Statement

Challenges without RAG

- **LLMs lack your data**: General-purpose models don’t know your internal docs, policies, or product details.
- **Hallucinations**: Models may invent answers when they don’t have the right information.
- **Stale knowledge**: Model knowledge is fixed at training time; your content changes over time.
- **Compliance and control**: You need answers traceable to specific documents and controllable by policy.

What developers need

- **Simplicity**: Build RAG without becoming an AI/ML expert.
- **Speed**: Go from “I have a folder of PDFs” to “I have a Q&A app” in minutes.
- **Flexibility**: Option to customize chunking, retrieval, and guardrails when requirements grow.
- **Enterprise readiness**: Logging, audit trails, security, and integration with existing ColdFusion apps.

ColdFusion RAG is designed to address these needs with a low-friction API and optional advanced configuration.

---

3. Solution

Simple RAG

Aspect	Simple RAG

Entry point	`simpleRAG(source, model, options?)`
	`For example,`
	
	ragService = simpleRAG(
	  expandPath("test.txt"), //source
	  chatModel, //model
	//vectorstore
	  {
	    vectorStore: vectorStoreClient,
	    chunkSize: 500,
	    chunkOverlap: 100,

	recursive: false
	}
	);

Configuration	Minimal; intelligent defaults
Use when	Prototypes, single corpus, standard behavior
Document processing	Built-in (folder/URL → chunks → vectors)

Core flow

1. **Ingest**: Load documents (files/URLs), split into chunks, generate embeddings, store in a vector store. The file formats that will be supported here are: .md, .markdown, .pdf, .doc, .docx, .xls, .xlsx, .ppt, .pptx, .zip, .jar, .war, .ear, .tar, .tar.gz, .tgz, .tar.bz2, .tbz2, .gz, .bz2, .odt, .ods, .odp, .rtf, .html, .htm, .xml, .eml, .msg, .epub .csv, .json, .xml, .rss, .atom, .log, .properties, .props. **But for this release, we recommend using txt files. PDF files might also work in some cases.**
2. **Retrieve**: For each user question, find the most relevant chunks (e.g., by similarity).
3. **Generate**: Send the question plus retrieved chunks to the LLM and return the answer.

ColdFusion hides most of this behind a single call for Simple RAG.

Learn more: Architecture in brief

- **Simple RAG**: One function returns a RAG service object. You call `ingest()` (sync or async) then `ask()` or `chat()`.
- **Document processing**: A separate service (`documentService()`) can load, split, and transform documents. the supported methods in document processing are:
    - load()
    - split()
    - transform()
    - transformSegments()
    - ingest()
    - ingestAsync()
    - transformSegmentsAsync()
    - transformAsync()
    - loadAsync()

---

4. Use Cases

- **Compliance and regulatory documentation search**
  Search policy and regulation docs; get answers that cite specific sections.

- **Enterprise knowledge management and employee self-service**
  Internal wikis, HR docs, and process guides as a question-answering assistant.
- **Customer support and documentation assistant**
  Product docs and FAQs powering support chatbots and help centers.
- **Sales and marketing content intelligence**
  Query over collateral, case studies, and competitive info for consistent messaging.

Learn more: Example scenarios

- **Legal/Compliance**: “What does our policy say about data retention in the EU?”
- **HR**: “What is the process to request remote work?”
- **Support**: “How do I reset my password?”
- **Sales**: “Which case studies mention security certifications?”

---

5. Getting Started with RAG in ColdFusion

Prerequisites

- ColdFusion 2025.0.07
- A chat model (e.g., OpenAI, Azure OpenAI, Gemini, Claude, Mistral) or local model via Ollama.
- Documents to index (PDF, DOCX, TXT, MD, etc.) in a folder or accessible via URL.

Minimal Simple RAG example (zero configuration)

```
<cfscript>

   chatModel = chatmodel({

   provider: "openai",

   modelName: "gpt-4o-mini",

   apiKey: application.openaiKey,

   temperature: 0.7

   });
```

```
vectorStoreClient = vectorstore({
provider: "INMEMORY",
embeddingModel: {
    provider: "ollama",
    modelName: "all-minilm",
    baseUrl: "http://localhost:11434"

}

});


ragService = simpleRAG(

expandPath("test.txt"),
chatModel,

{

    vectorStore: vectorStoreClient,
    chunkSize: 500,
        chunkOverlap: 100,

        recursive: false

    }



    );




    ragService.ingest();



    answer = ragService.ask("What is the inflation of year 1999 according to the
document?");

    writeOutput(answer.message);



</cfscript>
```

Typical workflow

1. **Create** the RAG service with simpleRAF`(source, model)` or simpleRAG`(source, model, options).`
2. **Ingest** once (or after doc updates): `future = ragBot.ingest(); result = future.get();`
3. **Query** with `ragBot.ask("question")` for one-off questions or `ragBot.chat("message")` for multi-turn chat.
4. **Inspect** with `ragBot.getStatistics()` (and optionally `getConfiguration()`).

Learn more: Sync vs async ingest

- `ingest()` returns a **Future**. Use `future.get()` to block until indexing is done, or use callbacks/timeouts as needed.

- For large corpora, prefer async so the page doesn’t block; you can show progress or poll `isDone()`.

6. Prerequisites and Environment Setup

ColdFusion and deployment

- **ColdFusion version:** CF2025.1 or later (as supported). RAG APIs are part of the core AI integration.
- **Deployment:** Supported on JEE (Tomcat, etc.) and standalone. For **chat session state** (multi-turn `chat()`), use a cluster-aware or persistent session if you need context across requests.

API keys and credentials

- **Chat model:** You need a valid API key or credential for your provider (OpenAI, Azure OpenAI, Google Gemini, Anthropic, Mistral, or a local Ollama endpoint). Store keys in environment variables or secure config—do not hardcode in source. We recommend using Application.cfc to store the credentials.
- **Embedding model:** When using a custom embedding model (e.g. OpenAI, Ollama), configure the same provider/API key or endpoint as required by that service.
- **Vector stores:** In-memory requires no credentials. For Qdrant, Chroma, Milvus, Pinecone, etc., configure URL, collection/index name, and any API keys per provider. There’s also support for in-memory, Milvus, or Qdrant.

  ```
  <cfscript>

  chatModel = chatmodel({

     provider: "openai",

     modelName: "gpt-4o-mini",

     apiKey: "key",

     temperature: 0.7

  });



  vectorStoreClient = vectorstore({
  ```

```
    provider: "INMEMORY",
    embeddingModel: {
        provider: "ollama",
        modelName: "all-minilm",
        baseUrl: "http://localhost:11434"
    }
});

ragService = rag(
    //getDirectoryFromPath(getCurrentTemplatePath()),
    expandPath("test.txt"),
    chatModel,
    {
        vectorStore: vectorStoreClient,
        chunkSize: 500,
        chunkOverlap: 100,
        recursive: false
    }
);


ragService.ingest();
answer = ragService.ask("Tell me inflation of the year 1999 according to the given document");
writeOutput(answer.message);
```

</cfscript>

Environment variables (recommended)

```
// Example: use environment variables for secrets
modelConfig = {
    provider: "openai",
    model: "gpt-4",
    apiKey: getEnv("OPENAI_API_KEY")  // or your env var name
};
```

---

7. When to Use Simple RAG

Choose	When
Simple RAG (`simplaRAG()`)	Single folder or URL of documents; default chunking and retrieval are fine; you want minimal code (source + model + optional options); prototyping or small/medium corpora.
Document processing only (`documentService()`)	You only need to load, split, or transform documents (e.g. for testing, custom pipelines, or feeding another system); no built-in retrieval or generation.

Re-ingest behavior: Calling `ingest()` again (e.g. after adding or updating documents) typically **re-indexes** from the configured source. Whether the vector store **appends** depends on implementation (e.g. same collection name may replace). Re-run ingest after document changes when you want queries to reflect new content; use a fresh RAG service instance if you changed source or options.

---

8. RAG Functions Reference

This section lists the main APIs used for RAG in ColdFusion, with short descriptions and code samples.

8.1 Simple RAG: `rag()` (create service)

Creates a Simple RAG service instance.

Syntax

```
ragService = simplaRAG(source, model [, options]);
```

Parameters

Parameter	Type	Required	Description
`source`	string or array	Yes	File path, folder path, or URL; or array of paths.
`model`	string or struct	Yes	Model name (e.g. `"gpt-3.5-turbo"`) or chat model config struct.
`options`	struct	No	Chunking, retrieval, vector store, embedding model, etc.

Returns: RAG service object (Simple RAG service).

Example: mandatory parameters only

```
ragService = rag("/path/to/docs", "gpt-3.5-turbo");
```

Example: with model struct

```
modelConfig = {
    provider: "openai",
    model: "gpt-3.5-turbo",
    apiKey: getEnv("OPENAI_KEY")
};
ragService = simpleRAG("/docs", modelConfig);
```

8.2 Simple RAG service methods

`ingest()` / async ingest

Indexes documents: load → split → embed → store. Async API returns a Future.

Syntax

```
future = ragService.ingestAsync();
result = future.get();  // block until complete
```

Returns: A Future; `future.get()` returns the ingest result (e.g. stats).

Example

```
result = ragService.ingest();
// If async:
future = ragService.ingestAsync();
result = future.get();
```

Example: PDFs and TXT

```
ragService = rag("/docs/pdfs", "gpt-3.5-turbo");
result = ragService.ingestAsync();  // PDFs parsed and indexed
```

`ask(question [, includeSources])`

One-off question over the indexed content.

Syntax

```
response = ragService.ask(question [, includeSources]);
```

Parameters

Parameter	Type	Required	Description
`question`	string	Yes	User question.

Returns: string (answer text).

Example

```
response = ragService.ask("What is the main topic?");
response = ragService.ask(question = "What is the process?", includeSources =
true);
```

`chat(message)`

Multi-turn conversation with session state/memory.

Syntax

```
response = ragService.chat(message);
```

Example

```
response1 = ragService.chat("Hello");
response2 = ragService.chat("Tell me more");
```

Context is maintained across messages in the same session.

`getStatistics()`

Returns statistics about the indexed corpus.

Syntax

```
stats = ragService.getStatistics();
```

Returns: Struct with fields such as `documents`, `chunks`, `vectors` (exact keys per implementation).

Example

```
answer = ragService.getStatistics();
writeDump(answer);
```

`getConfiguration()`

Returns the current configuration of the RAG service.

Syntax

```
config = ragService.getConfiguration();
```

8.4 Standalone document processing: `documentService()`

Service for loading, splitting, and transforming documents **without** running a full RAG pipeline. Useful for testing or feeding your own ingestion.

Syntax

```
docService = documentService();
```

Example: filesystem loader and recursive splitter

```
<cfscript>
  try {
    docService = documentService();

    // Step 1: Load
    documents = docService.load({
      path: application.getDocumentsDir(),
      pattern: "*.txt"
    });

    // Step 2: Split
    segments = docService.split(documents, {
      chunkSize: 500,
      chunkOverlap: 50
    });

    // Step 3: Transform segments
    function enrichSegment(struct document, required struct segment) {
      segment.metadata.pipeline = "full";
      segment.metadata.processedAt = now();
      return segment;
    }
    enrichedSegments = docService.transformSegments(segments, enrichSegment);

    // Step 4: Ingest
    vectorStoreClient = vectorstore({
      provider: "milvus",
      url: "http://see-lv-a181.corp.adobe.com:19530",
      databaseName: "default",
      collectionName: "dps_test_pipeline_full",
      dimension: 384,
      indexType: "HNSW",
      metricType: "COSINE",
      embeddingModel: {
        provider: "ollama",
        modelName: "all-minilm",
        baseUrl: "http://localhost:11434"
      }
    });
    result = docService.ingest(enrichedSegments, vectorStoreClient);

    if (isArray(documents) && arrayLen(documents) > 0
      && isArray(segments) && arrayLen(segments) > 0
      && isArray(enrichedSegments) && arrayLen(enrichedSegments) > 0
      && enrichedSegments[1].metadata.pipeline == "full"
      && isStruct(result) && result.successfulSegments > 0) {
      writeOutput("Full pipeline load->split->transformSegments->ingest completes successfully");
    } else {
      writeOutput("FAIL: Pipeline did not complete as expected");
    }
  } catch (any e) {
    writeOutput("ERROR: " & e.message);
  }
</cfscript>
```

Example: custom loader and transformer

```
function loadFromDatabase(required struct config) {
    var documents = [];
    var q = queryExecute("SELECT title, content FROM knowledge_base WHERE
status = 'published'");
    for (var i = 1; i <= q.recordCount; i++) {
        arrayAppend(documents, {
            text: q.content[i],
            metadata: { title: q.title[i], source: "database", id: i }
        });
    }
    return documents;
}

function cleanDocument(required struct document) {
    document.text = reReplace(document.text, "\s+", " ", "ALL");
    document.metadata.processedAt = now();
    document.metadata.wordCount = arrayLen(listToArray(document.text, " "));
    return document;
}

customDocService = documentService({
    documentLoader: { type: "custom", implementation: loadFromDatabase },
    documentSplitter: { type: "recursive", params: { chunkSize: 800,
chunkOverlap: 50 } },
    documentTransformer: cleanDocument
});
customResult = customDocService.processDocuments();
```

Document processing methods

Method	Description
`load(config)`	Load documents; returns array of document structs.
`loadAsync(config)`	Async load; returns Future.
`split(documents [, options])`	Split documents into segments; optional splitter options.
`transform(documents, transformerUDF)`	Transform each document with a UDF.
`transformSegments(segments, transformerUDF)`	Transform segments with a UDF.
`ingest(segments, vectorStoreClient [, options])`	Ingest segments into a vector store (when used with a client).
`processDocuments()`	Full pipeline (load, split, transform) when configured.

Other methods:

- transformSegmentsAsync()
- transformAsync()
- loadAsync()

8.5 Supporting functions used with RAG

These are used when configuring Simple RAG options:

Function	Purpose
`chatModel(config)`	Create chat model for generation.
`embeddingModel(config)`	Create embedding model for vectorizing text.
`vectorStore(config)`	Create vector store (in-memory, Qdrant, Chroma, etc.).
`getChatModel(config)`	Same idea as chatModel for advanced usage.
`getEmbeddingModel(config)`	Same idea as embeddingModel for advanced usage.

Example: vector store and embedding model for Simple RAG

```
<cfscript>
chatModel = chatmodel({
  provider: "openai",
  modelName: "gpt-4o-mini",
  apiKey: application.openaiKey,
  temperature: 0.7
});
vectorStoreClient = vectorstore({
  provider: "qdrant",
  url: application.vectorDB.qdrant.grpcUrl,
  apiKey: application.vectorDB.qdrant.apiKey,
  collectionName: "test_rag",
  metricType: "COSINE",
  dimension: 384,
  embeddingModel: {
    provider: "ollama",
    modelName: "all-minilm",
    baseUrl: "http://localhost:11434"
  }
});
ragService = simpleRAG(
  expandPath("test.txt"),
  chatModel,
  {
    vectorStore: vectorStoreClient,
    chunkSize: 500,
    chunkOverlap: 100,
    recursive: false
  }
);
ragService.ingest();
answer = ragService.ask("What is the inflation of year 1999 according to the document?");
writeOutput(answer.message);
</cfscript>
```

9. List of All Methods with Function Details

This section lists every method and function with parameters, return values, and code snippets.

Constructor / entry-point functions

1. simpleRAG()

Creates a Simple RAG service instance.

Parameters

Parameter	Type	Required	Description
`source`	string or array	Yes	File path, folder path, or URL to index; or array of paths.
`model`	string or struct	Yes	Model name (e.g. `"gpt-3.5-turbo"`) or chat model config struct (`provider`, `model`, `apiKey` / `credential`).
`options`	struct	Yes	Chunking, retrieval, vector store, embedding model. Keys: `chunkSize`, `chunkOverlap`, `chunkingStrategy`, `maxResults`, `similarityThreshold`, `includeSources`, `vectorStore`, `embeddingModel`, `systemPrompt`, etc.

Returns: RAG service object (Simple RAG service).

Example

```
// With model struct
modelConfig = { provider: "openai", model: "gpt-3.5-turbo", apiKey:
getEnv("OPENAI_KEY") };
ragService = simpleRAG("/docs", modelConfig);

// With options
ragService = simpleRAG("./docs/", chatModel, {
    vectorStore: vectorStore,
    embeddingModel: embeddingModel,
    chunkingStrategy: "hierarchical",
    includeSources: true
});
```

---

2. AiService()

Creates an AI service with optional retrieval augmentor

Parameters

Parameter	Type	Required	Description
`config`	struct	Yes	Configuration object.
`config.chatLanguageModel`	object	Yes	Chat model instance (from `chatModel()` or `getChatModel()`).
`config.retrievalAugmentor`	struct	No	Query transformer, query router (content retrievers), content aggregator.
`content.injector`	struct	No	When present, it controls how retrieved content and the user query are injected into the final prompt sent to the model.
`config.ingestion`	struct	No	Document loader, splitter, embedding store ingestor for ingestion.
`config.contentRetriever`	object	No	Single content retriever (alternative to `retrievalAugmentor` for simple RAG-style).
`config.inputGuardrails`	array	No	Array of UDFs that validate user input; each returns `{ action, reason [, newPrompt] }`.

Returns: AI service object.

Example

```
aiService = getAiService({
    chatLanguageModel: chatModel,
    retrievalAugmentor: {
        queryRouter: {
            routingModel: routingModel,
            contentRetrievers: [
                { type: "embeddingStore", params: { embeddingStore:
technicalVectorStore, embeddingModel: embeddingModel, maxResults: 5 },
description: "Technical docs" },
                { type: "embeddingStore", params: { embeddingStore: 
businessVectorStore, embeddingModel: embeddingModel, maxResults: 5 }, 
description: "Business docs" }
            ]
        }
    }
});
response = aiService.chat("How do I configure SSL?");
```

---

3. documentService()

Creates a standalone document processing service (load, split, transform) without running full RAG.

Parameters

Parameter	Type	Required	Description
`config`	struct	Yes	Configuration object.
`config.documentLoader`	struct	No*	Loader config: `type` ("filesystem", "url", "custom"), `params` (e.g. `source`, `recursive`, `pattern`), or `implementation` (UDF) for custom.
`config.documentSplitter`	struct	No*	Splitter config: `type` ("recursive", "hierarchical", "custom"), `params` (e.g. `chunkSize`, `chunkOverlap`), or `implementation` (UDF) for custom.
`config.documentTransformer`	function	No	UDF that receives a document struct `{ text, metadata }` and returns a transformed document struct.

*At least one of `documentLoader` or `documentSplitter` is typically required for useful behavior.

Returns: Document processing service object.

Example

```
docService = documentService({
    documentLoader: {
        type: "filesystem",
        params: { source: "./company-docs/", recursive: true, pattern: 
"*.{pdf,docx,txt,md}" }
    },
    documentSplitter: {
        type: "recursive",
        params: { chunkSize: 1000, chunkOverlap: 100 }
    }
});
loadResult = docService.load({ path: "./company-docs/", recursive: true });
```

Simple RAG service methods (on object returned by `rag()`)

4. ingest()

Indexes documents: load → split → embed → store. Asynchronous; returns a Future.

Parameters: None.

Returns: Future; call `future.get()` to block and get the ingest result (e.g. stats struct).

Example

```
future = ragService.ingest();
result = future.get();  // block until complete

// Optional: timeout
result = future.get(60, "SECONDS");
```

5. ask()

One-off question over the indexed content.

Parameters

Parameter	Type	Required	Description
`question`	string	Yes	The user's question.

Returns: string (answer text).

Example

```
answer = ragService.ask("What is the main topic?");
answerWithRefs = ragService.ask(question = "What is the process?",
includeSources = true);
writeOutput(answer);
```

6. chat()

Multi-turn conversation; maintains session state/memory. Session scope is typically **per request** unless the implementation supports a persistent session or user-scoped context (e.g. via `systemMessage(systemMessage, userId)` or application-managed conversation history). For multi-turn chat across HTTP requests, persist and pass conversation state as required by your API.

Parameters

Parameter	Type	Required	Description
`message`	string	Yes	The user's message in the conversation.

Returns: string (response text).

Example

```
response1 = ragService.chat("Hello");
response2 = ragService.chat("Tell me more about the first topic");
```

---

7. getStatistics()

Returns statistics about the indexed corpus.

Parameters: None.

Returns: Struct with keys such as `documents`, `chunks`, `vectors` (exact keys per implementation).

Example

```
stats = ragService.getStatistics();
writeOutput("Documents: #stats.documents#, Chunks: #stats.chunks#, Vectors:
#stats.vectors#");
```

---

8. getConfiguration()

Returns the current configuration of the RAG service.

Parameters: None.

Returns: Struct (configuration used to create or update the service).

Example

```
config = ragService.getConfiguration();
```

Document processing service methods (on object returned by `documentService()`)

10. load()

Loads documents from the configured or supplied source.

Parameters

Parameter	Type	Required	Description
`config`	struct	Yes	Source configuration. Common keys: `path` (file/folder path), `sourceType` ("filesystem", "url", etc.), `recursive` (boolean), `pattern` (e.g. `"*.pdf"`), `parserType`, `metadata`.

Returns: Array of document structs; each has `text` and `metadata`.

Example

```
documents = docService.load({
    path: "./docs/",
    recursive: true,
    pattern: "*.{txt,md,pdf}"
});
writeOutput("Loaded #arrayLen(documents)# documents.");
```

11. loadAsync()

Asynchronous load; returns immediately with a Future.

Parameters

Parameter	Type	Required	Description
`config`	struct	Yes	Same as `load(config)`: e.g. `path`, `recursive`, `pattern`.

Returns: Future; `future.get()` returns the Array of document structs.

Example

```
future = docService.loadAsync({ path: "./docs/", recursive: true });
// do other work...
documents = future.get();
```

12. split()

Splits documents into text segments (chunks).

Parameters

Parameter	Type	Required	Description
`documents`	array	Yes	Array of document structs from `load()`. Each element: `{ text, metadata }`.
`options`	struct	No	Splitter options: `chunkSize` (number, default 1000), `chunkOverlap` (number, default 100), `splitterType` (e.g. "recursive"), `separators` (array).

Returns: Array of text segment structs; each has `text` and `metadata`.

Example

```
segments = docService.split(documents);
segments = docService.split(documents, { chunkSize: 1500, chunkOverlap: 150
});
```

---

13. transform()

Transforms each document using a UDF.

Parameters

Parameter	Type	Required	Description
`documents`	array	Yes	Array of document structs (`{ text, metadata }`).
`transformerUDF`	function	Yes	UDF that accepts one document struct and returns a transformed document struct with `text` and `metadata`.

Returns: Array of transformed document structs.

Example

```
function cleanDoc(required struct doc) {
    doc.text = reReplace(doc.text, "\s+", " ", "ALL");
    doc.metadata.processedAt = now();
    return doc;
}
transformed = docService.transform(documents, cleanDoc);
```

---

14. transformSegments()

Transforms each text segment using a UDF.

Parameters

Parameter	Type	Required	Description
`segments`	array	Yes	Array of segment structs (`{ text, metadata }`).
`transformerUDF`	function	Yes	UDF that accepts one segment struct and returns a transformed segment struct.

Returns: Array of transformed segment structs.

Example

```
function addMetadata(required struct seg) {
    seg.metadata.enhanced = true;
    return seg;
}
transformedSegments = docService.transformSegments(segments, addMetadata);
```

---

15. ingest() (document processing)

Ingests text segments into a vector store (embed and store).

Parameters

Parameter	Type	Required	Description
`segments`	array	Yes	Text segment structs from `split()`.
`vectorStoreClient`	object	Yes	Vector store client (e.g. from `getVectorStore()` or equivalent) with embedded embedding model.
`options`	struct	No	e.g. `batchSize` (number, default 100), `continueOnError` (boolean, default true).

Returns: Struct with keys such as `totalSegments`, `successfulSegments`, `failedSegments`, `durationMs`, `successRate`, `isFullySuccessful`, `errors`.

Example

```
vectorStoreClient = getVectorStore({
    provider: "inmemory",
    embeddingModel: { provider: "ollama", modelName: "all-minilm", baseUrl:
"http://localhost:11434" }
});
result = docService.ingest(segments, vectorStoreClient, { batchSize: 50,
continueOnError: true });
```

```
writeOutput("Ingested #result.successfulSegments# of #result.totalSegments#
segments.");
```

---

16. ingestAsync() (document processing)

Asynchronous ingestion; returns a Future.

Parameters: Same as `docService.ingest(segments, vectorStoreClient [, options])`.

Returns: Future; `future.get()` returns the result Struct.

Example

```
future = docService.ingestAsync(segments, vectorStoreClient, { batchSize: 50
});
result = future.get();
```

Supporting functions (used when configuring RAG)

21. chatModel()

Creates a chat model for generation.

Parameters

Parameter	Type	Required	Description
`config`	struct	Yes	Provider config. Common keys: `provider` (e.g. "openai", "azure", "gemini", "anthropic", "mistral", "ollama"), `model` (e.g. "gpt-3.5-turbo", "gpt-4"), `apiKey` or `credential`, and provider-specific options (e.g. `baseUrl` for Ollama).

Returns: Chat model object.

Example

```
chatModel = chatModel({
    provider: "openai",
    model: "gpt-4",
    apiKey: "sk-your-api-key"
});
```

---

22. embeddingModel()

Creates an embedding model for vectorizing text.

Parameters

Parameter	Type	Required	Description
`config`	struct	Yes	Provider config. Common keys: `provider` (e.g. "openai", "ollama"), `model` (e.g. "text-embedding-3-small", "text-embedding-3-large"), `apiKey`, and provider-specific options.

Returns: Embedding model object.

Example

```
embeddingModel = embeddingModel({
    provider: "openai",
    model: "text-embedding-3-small",
    apiKey: "sk-your-api-key"
});
```

---

23. vectorStore()

Creates a vector store for storing and searching embeddings.

Parameters

Parameter	Type	Required	Description
`config`	struct	Yes	Store config. Common keys: `provider` or `type` (e.g. "inmemory", "qdrant", "chroma", "milvus", "pinecone"), `url`, `collection`, `host`, `port`, and provider-specific options (e.g. API keys).

Returns: Vector store object.

Example

```
vectorStore = vectorStore({
    provider: "qdrant",
    url: "http://localhost:6333",
    collection: "my_docs"
});
```

```
(!future.isDone()) {future.get(60,
"SECONDS");future.get();future.cancel(true);
```

10. Data Structures and Return Values

This section defines the **shapes** of structs and arrays used as inputs and return values so you can work with them reliably in code.

Document struct

Used by `documentService().load()`, `transform()`, and as elements in arrays returned by `load()`.

Key	Type	Description
`text`	string	The document body (extracted or loaded text).
`metadata`	struct	Optional. Source info and custom fields: e.g. `source` (file path or URL), `fileName`, `lastModified`, or application-defined keys.

Example: `{ text: "Document content here.", metadata: { source: "./docs/readme.md", fileName: "readme.md" } }`

Text segment struct

Used by `documentService().split()`, `transformSegments()`, `ingest()`, and as elements in arrays returned by `split()`.

Key	Type	Description
`text`	string	The chunk text.
`metadata`	struct	Optional. Often inherits document metadata plus segment-specific keys (e.g. chunk index, position).

Example: `{ text: "A paragraph of content.", metadata: { source: "./docs/readme.md", chunkIndex: 1 } }`

Ingest result (Simple RAG: `ragService.ingest()` → `future.get()`)

Key	Type	Description
(implementation-specific)	—	Often includes counts (e.g. documents processed, chunks created, vectors stored) and timing. Use `getStatistics()` for current corpus statistics after ingest.

Ingest result (document processing: `docService.ingest()`)

Key	Type	Description
`totalSegments`	number	Total segments passed to ingest.
`successfulSegments`	number	Segments successfully embedded and stored.
`failedSegments`	number	Segments that failed (e.g. embedding or store error).
`durationMs`	number	Total processing time in milliseconds.
`successRate`	number	Success percentage (0–100).
`isFullySuccessful`	boolean	True if all segments succeeded.
`isPartiallySuccessful`	boolean	True if at least one succeeded.
`isCompleteFailure`	boolean	True if none succeeded.
`errors`	array	Optional. Error messages or details for failures.

Values returned in ingest:

avgSegmentsPerDocument	1.00
documentsLoaded	1
initialized	YES
pipelineBuilt	YES
segmentsCreated	1
segmentsFailed	0
segmentsIngested	1
status	completed
timestamp	1771332403894
totalDurationMs	85
totalTimeMs	85
totalTimeSec	0.09

Load result (`docService.load()`)

Returns an **array** of document structs. Some implementations may wrap it in a struct with keys such as `documents`, `documentsLoaded`, `processingTimeMs`; check your API. When the return is a struct, it may look like:

Key	Type	Description
`documents`	array	Array of document structs.
`documentsLoaded`	number	Count of documents loaded.
`processingTimeMs`	number	Optional. Load duration in ms.

Metadata

absoluteDirectoryPath

absoluteFilePath
fileName
fileSize
sourceType

Split result (`docService.split()`)

Returns an **array** of text segment structs. When wrapped in a result struct:

Key	Type	Description
`segments`	array	Array of text segment structs.
`segmentsCreated`	number	Count of segments.
`averageSegmentLength`	number	Optional. Average character length per segment.

For example,

absoluteDirectoryPath	
absoluteFilePath	
chunkIndex	0
chunkOverlap	100
chunkSize	253
documentIndex	0
endOffset	253
fileNametest.txt	
fileSize	253
globalSegmentIndex	0
index	0
sourceType	file
splitTimestamp	{ts '2026-02-17 18:22:15'}
splitterType	recursive
startOffset	0
totalChunks	1
text	

Simple RAG `getStatistics()` result

Key	Type	Description

`documents`	number	Number of documents indexed.
`chunks`	number	Number of chunks/segments.
`vectors`	number	Number of vectors stored (typically same as chunks).

Exact keys may vary by implementation; use the returned struct in a way that tolerates extra or missing keys if needed.

For example,

avgSegmentsPerDocument	1.00
documentsLoaded	1
initialized	YES
pipelineBuilt	YES
segmentsCreated	1
segmentsFailed	0
segmentsIngested	1
status	completed
timestamp	1771332846083
totalDurationMs	443
totalTimeMs	444
totalTimeSec	0.44

11. Supported Document Formats

Format	Extensions	Parser
Text	.txt, .text	Text Parser
Markdown	.md, .markdown	Markdown Parser
PDF	.pdf	PDF Parser
Office	.doc, .docx, .xls, .xlsx, .ppt, .pptx	POI Parser
Archives	.zip, .jar, .war, .ear, .tar, .tar.gz, .tgz, .gz, .bz2	ZIP Document Parser
OpenDocument / HTML / XML / Email	.odt, .ods, .odp, .rtf, .html, .htm, .xml, .eml, .msg, .epub	Tika Parser
CSV	.csv	CSV Parser
JSON	.json	JSON Parser
XML	.xml	XML Parser
Feeds	.atom, .rss	Feed Parser
Properties	.properties, .props	Properties Parser

Logs	.log	Log Parser
Custom	—	Custom parser via code

Unsupported or corrupted files are skipped during ingest; processing continues for valid files.

Note: Some of the specified formats will be supported in future releases.

---

12. Configuration Reference

Simple RAG options (high level)

Option	Type	Default	Description
`chunkSize`	number	1000	Max characters per chunk.
`chunkOverlap`	number	200	Overlap between chunks.
`chunkingStrategy`	string	"recursive"	e.g. "recursive", "hierarchical".
`maxResults`	number	5	Max chunks to retrieve per query.
`vectorStore`	object	in-memory	Vector store instance.
`embeddingModel`	object	model-based default	Embedding model.
`batchSize`	number	10	Batch size for loading.

Document load config (documentService / load)

When calling `docService.load(config)` or configuring a document loader, `config` can include:

Key	Type	Required	Description
`path`	string	Yes (filesystem)	File or directory path.
`sourceType`	string	No	`"filesystem"`, `"url"`, or as supported; often auto-detected from `path` or `url`.
`recursive`	boolean	No	If true, load from subdirectories. Default: false.
`pattern`	string	No	Glob or regex to filter files (e.g. `"*.pdf"`, `"*.{txt,md}"`).
`patternType`	string	No	`"glob"` (default) or `"regex"`.
`includePatterns`	array	No	Multiple inclusion patterns; file must match at least one.

`excludePatterns`	array	No	Files matching any of these are excluded.
`parserType`	string	No	Override parser: `"text"`, `"pdf"`, `"docx"`, etc.
`metadata`	struct	No	Custom metadata to attach to loaded documents.
`url`	string	Yes (url)	Document or base URL when `sourceType` is `"url"`.
`timeout`	number	No	Request timeout in ms for URL loading.

---

13. Provider Configuration Reference

These tables summarize common keys for chat models, embedding models, and vector stores. Required keys and supported values depend on the specific provider; treat this as a quick reference and check provider docs for details.

Chat model `config` (chatModel)

Key	Type	Description
`provider`	string	One of: `"openai"`, `"azure"`, `"gemini"`, `"anthropic"`, `"mistral"`, `"ollama"`, or as supported.
`model`	string	Model name (e.g. `"gpt-3.5-turbo"`, `"gpt-4"`, `"claude-3-opus-20240229"`, `"llama2"` for Ollama).
`apiKey` or `credential`	string	API key or credential for the provider (not needed for local Ollama).
`baseUrl`	string	Optional. Override endpoint (e.g. Ollama `"http://localhost:11434"`).

Embedding model `config` (embeddingModel)

Key	Type	Description
`provider`	string	e.g. `"openai"`, `"ollama"`, or as supported.
`model`	string	Model name (e.g. `"text-embedding-3-small"`, `"text-embedding-3-large"`, `"nomic-embed-text"` for Ollama).
`apiKey`	string	API key for cloud providers.
`baseUrl`	string	Optional. Override endpoint (e.g. Ollama).

Vector store `config` (vectorStore)

Key	Type	Description
`provider` or `type`	string	One of: `"inmemory"`, `"qdrant"`, `"chroma"`, `"milvus"`, `"pinecone"`, or as supported.
`url`	string	Store URL (e.g. Qdrant `"http://localhost:6333"`).

`collection`	string	Collection or index name.
`host` / `port`	string / number	Alternative to `url` for some stores (e.g. Chroma).
`apiKey`	string	Required by some cloud stores (e.g. Pinecone).
(provider-specific)	—	e.g. `environment`, `project`, `namespace`; check provider docs.

---

15. Troubleshooting and FAQ

"No documents indexed" or error when calling ask() / chat()

- **Cause:** You called `ask()` or `chat()` before ingest completed or before any documents were successfully indexed.
- **Fix:** Call `ingest()` first (and `future.get()` if using the async API). Wait until ingest completes before querying. If the source directory is empty or all files were skipped (unsupported format, permissions, or corruption), ingest may complete with zero documents—check `getStatistics()` and fix the source or filters.

Ingest is very slow or times out

- **Cause:** Large corpus, big files, or slow embedding/vector store.
- **Fix:** Use async ingest and a longer timeout: `future.get(300, "SECONDS").` Consider increasing `batchSize` in options, or splitting the source into smaller batches. For very large documents, check Limits and Considerations.

Dimension mismatch between embedding model and vector store

- **Cause:** The embedding model's output dimension (e.g. 1536 for `text-embedding-3-small`) does not match the dimension the vector store expects (e.g. after changing the embedding model or reusing an existing collection).
- **Fix:** Use an embedding model that matches the store's expected dimension, or create a new collection/index with the correct dimension. Each model has its own embedding dimension, and that dimension should be used as-is. Instead of selecting a model to match an existing dimension, the vector store dimension should be configured to match the chosen model.

Query returns irrelevant or empty answers

- **Cause:** Chunks may be too large/small, similarity threshold too high, or documents not well aligned with the question.
- **Fix:** Tune `chunkSize`, `chunkOverlap`, and `chunkingStrategy`, to see which chunks were retrieved. Consider re-ingesting after changing chunking options.

How do I re-index after adding or updating documents?

- Call `ingest()` again on the same RAG service instance. Behavior (append) depends on implementation; typically the same source and collection will update or replace the indexed content. Ensure ingest completes before querying.

Chat() context not maintained across requests

---

16. Limits and Considerations

- **Document size and count:** Very large single documents (e.g. hundreds of MB) or very large corpora may cause long ingest times, timeouts, or high memory use.
- **PDFs:** Image extraction from PDFs may not be supported; only text is typically indexed. Password-protected or corrupted PDFs may be skipped or throw; handle errors and check ingest results.
- **In-memory vector store:** Default in-memory store does not persist across restarts. Use a persistent store (Qdrant, Chroma, etc.) for production or multi-instance deployments.
- **API rate limits and quotas:** Cloud chat and embedding providers enforce rate limits and quotas. For high throughput, use batching, backoff, and consider local models (e.g. Ollama) where appropriate.
- **Logging and audit:** Implementations may provide RAG-specific logging (e.g. component execution, UDF timing) and audit trails (e.g. who called which API, ingestion events). Enable and configure these per your security and compliance requirements. Exceptions in RAG appear normally as would in exception log files in cfusion/logs.
