Boudica RAG System Technical Overview

1. Executive Summary

The Boudica RAG (Retrieval-Augmented Generation) system is a high-performance, C++ based information retrieval layer integrated directly into the Boudica LM (Language Model) inference engine. It allows the model to access private, domain-specific data securely, reducing hallucinations and enabling role-based access to information without retraining the model.

2. Architecture

2.1 Core Components

- **Storage Layer**: PostgreSQL 15+ with `pgvector` extension.
  - Table: `boudislm.rag_corpus` stores text chunks, vector embeddings, metadata, and ACL tags.
  - Table: `boudislm.api_keys` stores user credentials and access permissions.
- **Ingestion Engine**: `boudica_rag_builder`
  - Handles PII sanitization (Redaction of emails, IPs, etc.).
  - Performs intelligent chunking (Sliding window with overlap).
  - Extracts metadata and security categories from directory structures.
- **Retrieval Engine**: `RAGRetriever` (C++ Library)
  - **Keyword Search**: `tsvector` based full-text search with ranking.
  - **Vector Search**: Cosine similarity search using embeddings (Hybrid mode).
  - **Filtering**: Mandatory SQL-level filtering based on user `allowed_categories`.
- **API Layer**: `boudica_cgi`
  - Standard OpenAI-compatible endpoint (`/v1/chat/completions`).
  - Injects retrieved context into the system prompt automatically.

3. Security & Authentication (RBAC)

The system implements a strict Role-Based Access Control (RBAC) mechanism at the retrieval level. Access is not determined by the model, but by the retrieval engine filtering data *before* the model sees

it.

3.1 Category-Based Access

Every document chunk in the RAG corpus is assigned a `category` (e.g., “sales”, “engineering”, “hr”). * By default, documents ingest their category from their parent directory name (e.g., `/data/sales/Q3_report.pdf` -> `sales`). * Categories can be explicitly overridden during ingestion.

3.2 User Management

Users are authenticated via API Keys. Each key is bound to a specific list of `allowed_categories`.

Example: * **User “Bob” (Sales)**: `allowed_categories = {"sales", "marketing"}`. * Bob *cannot* retrieve chunks tagged “engineering”. * **User “Marianne” (Dev)**: `allowed_categories = {"engineering", "sales"}`. * Marianne can access technical docs and sales briefs.

3.3 Managing Users (CLI)

Use the `boudica_keymanager` tool to manage access:

```
# Create a Sales user
./build/boudica_keymanager create "Bob" --categories "sales"


# Create a Super-user/Dev
./build/boudica_keymanager create "Marianne" --categories
        "sales,development"


# Update categories for existing user
./build/boudica_keymanager set-categories <API_KEY_HASH>
        "hr,finance"
```

4. Workflows

4.1 Data Ingestion

There are two methods to populate the RAG system.

Method A: Bulk Ingestion from Training Data

Ingest data that is already part of the model’s training set (useful for grounding the model on data it has already seen but needs exact references for).

```
# reads from boudislm.training_corpus table
./build/boudica_rag_builder config.json --chunk-size 512
```

Method B: Direct File Ingestion (Independent)

Add new documents immediately without retraining or DB import.

```
# Ingest limits.txt into the 'engineering' category
./build/boudica_rag_builder config.json \
    --input-file ./md/technical_documents/limits.txt \
    --category engineering
```

Note: If `--category` is omitted, the tool attempts to infer it from the file's parent folder name.

4.2 Inference (Chat)

When a user makes a request to the chat API, the following sequence occurs: 1. **Auth**: API Key is validated. `allowed_categories` are loaded. 2. **Retrieval**: The user's query is sent to `RAGRetriever`. 3. **Filtration**: The SQL query enforces `WHERE category = ANY(user_allowed_categories)`. 4. **Augmentation**: Top-K chunks are prepended to the system prompt. 5. **Generation**: The model generates an answer based on the augmented prompt.

5. Capabilities

5.1 Hybrid Search

The system supports fetching documents based on: * **Semantic Meaning**: Mapping query embedding to document embeddings (understanding intent). * **Keyword Match**: Exact matching of technical terms (e.g., error codes, specific product names).

5.2 Context Awareness

- **Automatic Context Window Management**: The retriever limits results to fit within the `config_1b_fp16_infer.json` "max_context_tokens" setting, ensuring the prompt never overflows the model's capacity.
- **Source Citations**: Retrieved chunks include `source_url` or `source_path` metadata, allowing the frontend to display citations.

5.3 Performance

- Written in pure C++17 for minimal latency.
- Direct memory integration with the inference server (no HTTP overhead for retrieval steps).
- Uses `libpqxx` for efficient connection pooling to PostgreSQL.