⍰ Documento Técnico Detalhado: Sistema RAG Jurídico (Versão Atualizada)

1. Visão Geral e Arquitetura do Sistema

O sistema é um **Chatbot de Recuperação Aumentada por Geração (RAG)**, projetado para fornecer respostas precisas, fundamentadas e rastreáveis sobre a legislação do Estado de Santa Catarina[1]. A arquitetura divide o fluxo de trabalho em duas fases críticas: **Ingestão (Offline)** e **Consulta (Online)**, priorizando a **busca híbrida** para otimizar latência e custo[2].

---

2. Fase de Ingestão e Indexação – `src/data_prep.py` (Antigo `processar_leis.py`)

Esta fase é um processo ETL (Extract, Transform, Load) que prepara os dados para busca de alta velocidade, executado uma única vez ou a cada atualização do corpus legal[3].

2.1. Processamento e Normalização (Limpeza de Dados)

O script utiliza o **BeautifulSoup** para extrair o texto de arquivos HTML, seguido por uma rotina de normalização[4]:

- **Extração de Metadados:** Extração do **Ano de Publicação** e **Nome do Arquivo**[5].
- **Remoção de Conteúdo Taxado:** Implementa uma lógica **Regex (Expressão Regular)** para identificar e remover blocos de texto que contenham indicativos de revogação (ex: tags `<s>` ou notações explícitas de revogação), garantindo que apenas o conteúdo potencialmente vigente seja indexado[6].
- **Chunking Estratégico:** O texto é dividido em *chunks* (pedaços) de aproximadamente **1500 caracteres** (`CHUNK_SIZE_LIMIT`), com sobreposição de tokens para garantir continuidade semântica entre os pedaços[7].

2.2. Geração de Embeddings e Gerenciamento da API

O script utiliza o SDK do Google para gerar os vetores[8]. O processo é isolado na função `generate_embeddings_batch` para gerenciamento de erros e limites de taxa[9].

Detalhe Técnico	Implementação	Finalidade
Endpoint/Método	`client.models.embed_content()`	Chamada para gerar vetores. [10]
Modelo	`"text-embedding-004"`	Modelo otimizado para tarefas de recuperação, de alta precisão

Detalhe Técnico	Implementação	Finalidade
		do Gemini. [11]
Parâmetro `contents`	Lista de *strings* de texto (lote de 100).	Otimiza a latência ao enviar múltiplos *chunks* em uma única chamada em lote. [12]
Resiliência	Biblioteca **Tenacity** (`@retry`)	Aplica uma estratégia de *retry* exponencial. Garante que o processo continue em caso de Rate Limit ou falha temporária da API. [13]

2.3. Indexação e Mapeamento

- **FAISS Indexing:** Os vetores gerados são armazenados no índice **FAISS** (`faiss_index.bin`)[14]. O FAISS otimiza consultas de Similaridade de Cosseno (ou Distância Euclidiana) para buscar *k* Nearest Neighbors (kNN)[15].
- **Mapeamento de ID (`documents_map.json`):** Estrutura de busca exata[16]. Associa a chave Regex (`doc_ID:Art_Numero`) ao índice de *chunk* no array de vetores, permitindo acesso direto e rápido[17].

---

3. Fase de Consulta (Online) – `src/rag_service.py` (Antigo `chatbot.py`)

O **`src/rag_service.py`** [18]contém o *back-end* lógico do RAG, centralizado na função `find_document_chunks` (Busca) e `generate_response` (Geração)[19].

3.1. Lógica de Busca Híbrida (`find_document_chunks`)

Rota de Busca	Lógica de Recuperação	Vantagem/Desvantagem
Rota A: ID Match (Regex)	**Busca Determinística:** Utiliza **Regex sofisticado** para extrair o ID da Lei/Artigo. Se houver sucesso, ele ignora o FAISS e recupera o *chunk* pelo `documents_map.json`[20].	**Vantagem:** Latência e custo mínimos, precisão de 100% no artigo exato. **Desvantagem:** Não responde a perguntas conceituais. [21]
Rota B: Fallback FAISS	**Busca Semântica:** Ativada por falha na Rota A. A *query* é vetorizada. O vetor é usado para consultar o `faiss_index.bin` para os *top-K chunks* mais similares semanticamente (recuperação por similaridade de cosseno)[22].	**Vantagem:** Responde a perguntas conceituais. **Desvantagem:** Maior latência (duas chamadas de API), custo maior. [23]

3.2. Chamadas de API de Geração de Conteúdo

A geração da resposta é a chamada final e de maior custo/latência[24].

Detalhe da Chamada	**embed_content (Rota B)**	**generate_content (Rotas A e B)**
Função	`client.models.embed_content(model, contents=[user_query])` [25]	`client.models.generate_content(model, contents=[prompt])` [26]
Modelo Usado	`"text-embedding-004"` [27]	`"gemini-2.5-flash"` (Melhor *trade-off* entre velocidade e qualidade para tarefas RAG). [28]
Payload de Entrada	A *string* exata da pergunta do usuário. [29]	**Prompt Augmentation:** É a concatenação da `SYSTEM_INSTRUCTION` + Contexto Recuperado + Pergunta do Usuário. [30]
Parâmetros	Não se aplica (apenas a *query*). [31]	`temperature=0.0`: Configuração crucial para respostas factuais, minimiza a criatividade e a alucinação. [32]

---

4. Implementação da Lógica de Negócio (Regra 4)

A **Regra 4 (Tratamento de Alterações)** é a principal diferenciação do sistema, garantindo a validade jurídica das informações[33]. Ela é imposta pela **SYSTEM_INSTRUCTION**[34].

Requisito Jurídico	**Mecanismo de Implementação**	**Função da SYSTEM_INSTRUCTION (Regra de Grounding)**
Artigos Alterados	O *chunk* contém o texto atualizado e o metadado da **Lei Alteradora** (ex: "Redação dada pela LC 789").	O LLM é obrigado a citar a **Lei Alteradora** como prova da vigência. [35]
Artigos Não Alterados (Original)	O **`src/data_prep.py`** injetou o **Ano de Publicação** no cabeçalho do *chunk*[36].	O LLM é instruído a usar o **Ano de Publicação** (no cabeçalho) para declarar que "a redação apresentada é a vigente desde [Ano]", superando o problema de artigos que não possuem notas de alteração. [37]

---

5. Estrutura de Arquivos e Deploy

O deploy é otimizado para **Streamlit Community Cloud**, com foco em segurança de credenciais[38].

Arquivo	Propósito	Dependências Críticas
`src/data_prep.py`	Ingestão, ETL e Indexação.	`google-genai`, `faiss-cpu`, `beautifulsoup4`, `tenacity`. [39]
`src/rag_service.py`	Core Logic (Coração do RAG). Contém a `SYSTEM_INSTRUCTION`, a lógica de Busca Híbrida e a chamada de Geração.	`google-genai`, `faiss-cpu`, `numpy`. [40]
`app_web.py`	Interface de Usuário (Streamlit) e Gerenciamento de Sessão/UI.	`streamlit`, **`src/rag_service.py`**. [41]
`requirements.txt`	Lista todas as bibliotecas necessárias para o ambiente de *deployment*.	Usado pelo Streamlit Cloud/Ambiente de Nuvem. [42]
`.streamlit/secrets.toml`	**CRÍTICO PARA DEPLOY.** Arquivo que armazena a `GEMINI_API_KEY` de forma segura.	Acesso via `st.secrets["GEMINI_API_KEY"]` no `app_web.py`. [43]