Instituto Superior de Engenharia de Lisboa (ISEL)
Departamento de Engenharia Eletrónica e de Telecomunicações e Computadores (DEETC)

LEIM
Licenciatura em Engenharia Informática e Multimédia
Unidade Curricular de Projeto

Pesquisa Semântica para Registos Criminais

Pedro Miguel Tavares da Silva Pato e Silva (49734)

Orientador

Professor [Doutor] Helder Filipe de Oliveira Bastos

Junho, 2025

Resumo

Motivado pelo interesse na área criminal e pela necessidade de melhorar a recuperação de informação em registos criminais, este projeto desenvolveu uma API de pesquisa semântica baseada em processamento de linguagem natural (*NLP*), capaz de interpretar o contexto das consultas.

O sistema foi desenvolvido em *Python*, com uma interface construída em *Flask* e *React*, e encontra-se totalmente contentorizado com *Docker*, garantindo maior portabilidade e escalabilidade.

Após uma fase de testes, o modelo de embeddings selecionado foi o *medcpt_article*, que obteve uma precisão de 95,1% nos testes realizados especificamente para este projeto.

A arquitetura modular, suportada por contentores, permite uma gestão eficiente dos dados, com funcionalidades como seleção de *buckets* e mecanismos de *healthcheck*. Desafios como a transição para *Docker* e a otimização do *chunking* foram ultrapassados de forma iterativa ao longo do desenvolvimento.

Como trabalho futuro, está prevista a realização de *fine-tuning* do modelo, em colaboração com a Polícia Judiciária, para melhorar ainda mais a precisão e a adaptação ao domínio específico dos registos criminais.

Abstract

Motivated by an interest in the criminal domain and the need to improve information retrieval from criminal records, this project developed a semantic search API based on natural language processing (*NLP*), capable of interpreting the context of user queries.

The system was developed in *Python*, featuring an interface built with *Flask* and *React*, and is fully containerized using *Docker*, ensuring greater portability and scalability.

After a testing phase, the selected embedding model was *medcpt_article*, which achieved an accuracy of 95.1% in tests conducted specifically for this project.

The container-based modular architecture enables efficient data management, with features such as *bucket* selection and *healthcheck* mechanisms. Challenges such as transitioning to *Docker* and optimizing the *chunking* process were overcome iteratively throughout development.

Future work includes performing *fine-tuning* of the model in collaboration with the Portuguese Criminal Police (*Polícia Judiciária*), aiming to further enhance accuracy and domain adaptation to criminal record data.

Agradecimentos

Agradeço ao Professor Helder Filipe de Oliveira Bastos pelas aulas da unidade curricular de Projeto, que, apesar de diferentes, contribuíram para um excelente aproveitamento das aulas, e pela orientação inestimável e apoio contínuo, fundamentais para o sucesso deste projeto. Agradeço à Polícia Judiciária, cuja área de atuação motivou este trabalho e forneceu importante contexto, assim à minha namorada e os meus amigos pela inspiração e discussões técnicas

Dedico este trabalho a todos os que sempre acreditaram em mim, especialmente à minha família e namorada, pelo apoio incondicional ao longo deste percurso.

Índice

Resumo **i**

Abstract **iii**

Agradecimentos **v**

Índice **ix**

Lista de Tabelas **xiii**

Lista de Figuras **xv**

1 Introdução **1**

2 Polícia Judiciária e Pesquisa Semântica **3**

2.1 Introdução à Polícia Judiciária 3

2.2 Registos Criminais na PJ 3

2.3 Desafios na Pesquisa de Registos 4

2.4 Compreensão da Pesquisa Semântica 4

2.4.1 Introdução à Pesquisa Semântica 4

2.4.2 Pesquisa Semântica no Projeto 5

3 Trabalho Relacionado **7**

4 Modelo Proposto **11**

4.1 Requisitos 11

4.2 Fundamentos 12

4.3 Abordagem 12

5 Implementação do Modelo **13**
5.1 Arquitetura 13
5.1.1 Arquitetura da Interface 14
5.1.2 Arquitetura dos Buckets 14
5.1.3 Componentes Principais 15
5.2 Métodos da Pesquisa Semântica 24
5.2.1 Cálculo de Similaridade 24
5.2.2 Escolha do Tamanho de Chunk 24
5.3 Interface 25

6 Validação e Testes **27**
6.1 Corpus, Consultas e Procedimento de Avaliação 27
6.1.1 Consultas Utilizadas 28
6.1.2 Verificação de Robustez a Erros Ortográficos 28
6.2 Gráficos de Desempenho por Modelo 29
6.2.1 medcpt-article 29
6.2.2 medcpt-query 32
6.2.3 nomic-embed 33
6.2.4 all-minilm 35
6.2.5 mxbai-embed 38
6.2.6 avrsfr-embed 39
6.2.7 snowflake-embed2 41
6.3 Comparação de Modelos 42
6.4 Descrição dos Resultados 44
6.4.1 Análise por Quadrantes 44
6.4.2 Conclusões Gerais 45
6.4.3 Limitações e Trabalho Futuro 46
6.5 Script de Teste 47
6.6 Script `test_model.py` 47
6.7 Exemplos de Uso da Interface 47

7 Conclusões e Trabalho Futuro **53**
7.1 Conclusão 53
7.2 Principais Conclusões 54
7.3 Trabalho Futuro 54

A Gestão de Código e Controlo de Versões **57**

B Exemplo de Documento de Teste **59**

C Documentação **61**
C.1 Guia de Utilização 61
C.1.1 Instruções do README 61

Lista de Tabelas

4.1 Requisitos Funcionais 11
4.2 Requisitos Não Funcionais 12

Lista de Figuras

3.1 Arquitetura do Elasticsearch com extensões de NLP. 8
3.2 Crachá da Polícia Judiciária. 8
3.3 Exemplo de pesquisa semântica com BioBERT em relatórios médicos 9
3.4 Interface do sistema ROSS Intelligence para pesquisa jurídica . 10

5.1 Arquitetura geral da API de pesquisa semântica. 14
5.2 Detalhamento da arquitetura da interface em React e integração com o AI Node. 14
5.3 Arquitetura dos serviços do tipo bucket, com processamento e armazenamento independente. 15

6.1 Similaridade K3 - `medcpt-article` 30
6.2 Tempo de processamento - `medcpt-article` 30
6.3 Tempo de consulta K3 - `medcpt-article` 31
6.4 Similaridade K3 - `medcpt-query` 32
6.5 Tempo de processamento - `medcpt-query` 33
6.6 Tempo de consulta K3 - `medcpt-query` 33
6.7 Similaridade K3 - `nomic-embed` 34
6.8 Tempo de processamento - `nomic-embed` 34
6.9 Tempo de consulta K3 - `nomic-embed` 35
6.10 Similaridade K3 - `all-minilm` 36
6.11 Tempo de processamento - `all-minilm` 36
6.12 Tempo de consulta K3 - `all-minilm` 37
6.13 Similaridade K3 - `mxbai-embed` 38
6.14 Tempo de processamento - `mxbai-embed` 39
6.15 Tempo de consulta K3 - `mxbai-embed` 39

6.16 Similaridade K3 - `avrsfr-embed` 40
6.17 Tempo de processamento - `avrsfr-embed` 40
6.18 Tempo de consulta K3 - `avrsfr-embed` 41
6.19 Similaridade K3 - `snowflake-embed2` 41
6.20 Tempo de processamento - `snowflake-embed2` 42
6.21 Tempo de consulta K3 - `snowflake-embed2` 42
6.22 Comparação da similaridade média (K3) entre os modelos para *chunk size* 300. 43
6.23 Comparação do tempo de consulta K3 entre os modelos para *chunk size* 300. 43
6.24 Comparação do tempo de processamento de documentos entre os modelos para *chunk size* 300. 44
6.25 Gráfico de Similaridade vs Tempo comparando a similaridade média K3 e o tempo de consulta K3 para modelos de embeddings com *chunk size* 300. 46
6.26 Tela da interface. 47
6.27 Resultado de uma consulta. 48
6.28 Consulta num só *bucket* com $k = 3$. 49
6.29 Consulta num só *bucket* com $k = 12$. 50
6.30 Resultados da consulta num só *bucket* com $k = 12$. 51
6.31 Demonstração de um *bucket* que falhou. 51
6.32 Seleção do *bucket* que falhou agora indisponível. 52

Capítulo 1

Introdução

A recuperação de informação em registos criminais é um desafio crítico para sistemas judiciais e de segurança pública, devido à complexidade e o volume dos dados existentes.

As abordagens tradicionais baseadas em palavras-chave frequentemente falham em captar o contexto semântico das consultas, o que leva a resultados irrelevantes ou incompletos.

Este projeto, impulsionado por um interesse pessoal na área criminal e pela necessidade de modernizar sistemas de pesquisa, desenvolveu uma API de pesquisa semântica e uma interface dedicada à consulta de relatórios criminais.

A solução recorre a técnicas de processamento de linguagem natural (*NLP*) para interpretar consultas em linguagem natural e recuperar documentos com base na sua similaridade semântica.

A implementação foi realizada com recurso a *Python*[3], *Flask*[2], *React*[4], *Ollama*[13] e *Docker*[1], utilizando o modelo *medcpt_article*[9] selecionado após testes comparativos com sete alternativas.

A arquitetura da API é baseada em microserviços, um modelo onde cada componente (como os buckets ou a interface) operam de forma independente, comunicando via interfaces bem definidas, o que facilita escalabilidade e manutenção.

Os utilizadores podem selecionar qualquer bucket para consulta, sendo estes geridos de forma dinâmica pela aplicação. A API devolve os k resultados mais relevantes, suportada por mecanismos de *healthcheck* que asseguram a robustez e disponibilidade dos serviços.

Este projeto apresentou desafios como a transição para Docker, optimização de chunking e resolução de problemas como resultados duplicados ou incompletos.

Do ponto de vista da engenharia, o projeto seguiu uma abordagem iterativa, com foco na modularidade, escalabilidade e validação rigorosa. Testes extensivos avaliaram métricas como similaridade, tempo de consulta e tempo de processamento de documentos.

Este trabalho contribui para a modernização de sistemas judiciais, com potencial aplicação em outros domínios, como relatórios médicos ou jornalísticos.

Este relatório está organizado da seguinte forma:

- **Capítulo 2 - Trabalho Relacionado**: Contextualiza o projeto em relação a sistemas de busca semântica e gestão de registos criminais.
- **Capítulo 3 - Modelo Proposto**: Detalha os requisitos, fundamentos tecnológicos e abordagem metodológica.
- **Capítulo 4 - Implementação do Modelo**: Descreve a arquitetura, componentes e desafios técnicos.
- **Capítulo 5 - Validação e Testes**: Apresenta os resultados dos testes e análises comparativas.
- **Capítulo 6 - Conclusões e Trabalho Futuro**: Resume os resultados e sugere direções futuras.

Capítulo 2

Polícia Judiciária e Pesquisa Semântica

2.1 Introdução à Polícia Judiciária

A Polícia Judiciária (PJ), criada em 1945 pelo Decreto-Lei n.º 35042, é o órgão superior de polícia criminal em Portugal, subordinado ao Ministério da Justiça. Investiga crimes graves, como crime organizado, terrorismo, tráfico de drogas, corrupção e crimes financeiros. A sua estrutura inclui diretorias regionais e unidades especializadas, como a Unidade Nacional de Combate ao Cibercrime e Criminalidade Tecnológica e o Laboratório de Polícia Científica.

2.2 Registos Criminais na PJ

Os registos criminais em Portugal, geridos pela Direção-Geral da Administração da Justiça (DGAJ), contêm informações sobre condenações de indivíduos maiores de 16 anos. A PJ acede a bases de dados internas com relatórios detalhados, evidências e perfis de suspeitos, essenciais para investigações. Esses registos, que abrangem crimes complexos como fraude financeira, incluem relatórios textuais, documentos financeiros e evidências forenses.

2.3 Desafios na Pesquisa de Registos

Os sistemas tradicionais de pesquisa baseados em palavras-chave apresentam limitações para a PJ:

- **Dependência de Palavras-Chave**: Exige termos exatos, o que dificulta procuras em documentos extensos.
- **Falta de Contexto**: Não compreende o significado semântico, retornando resultados irrelevantes.
- **Volume de Dados**: Grandes quantidades de dados digitais sobrecarregam as pesquisas tradicionais.
- **Dados Interconectados**: Casos exigem compreender relações entre documentos.

2.4 Compreensão da Pesquisa Semântica

2.4.1 Introdução à Pesquisa Semântica

A pesquisa semântica é uma técnica avançada que visa compreender a intenção e o significado contextual de uma consulta, indo além da correspondência exata de palavras-chave.

Utiliza processamento de linguagem natural (NLP) e AI, interpreta a semântica das palavras, recupera resultados conceptualmente relevantes, sem termos idênticos.

Por exemplo, uma consulta sobre crimes que envolvem computadores pode corresponder a documentos sobre "cibercrime", porque reconhece a similaridade conceptual.

Essa abordagem é valiosa em domínios com dados complexos, como registos criminais, onde documentos extensos e jargões específicos dificultam procuras tradicionais.

A pesquisa semântica usa *embeddings* representações numéricas de texto que capturam significado para medir a similaridade entre as consultas e os documentos.

2.4.2 Pesquisa Semântica no Projeto

Neste projeto, a pesquisa semântica é o foco para a API de recuperação de registos criminais. A API utiliza modelos de linguagem que geram *embeddings* para consultas e documentos, os quais são comparados via similaridade de cosseno, que mede a proximidade semântica entre vetores.

Por exemplo, uma consulta sobre crimes cibernéticos pode retornar documentos relacionados com "crimes digitais", mesmo sem existir correspondência textual exata.

A API está preparada para lidar com erros ortográficos como "actividades"em vez de "atividades"mantendo os resultados relevantes ao focar no conteúdo semântico em vez da forma literal.

Foram utilizados e avaliados diversos modelos de geração de embeddings, todos executados via plataforma Ollama[13], com características distintas:

- `medcpt-article`[9]: especializado para lidar com textos longos e estruturados, como relatórios médicos ou jurídicos.
- `medcpt-query`[10]: variante adaptada para consultas curtas em linguagem natural.
- `nomic-embed-text`[12]: modelo versátil para diversos tipos de textos, com bom equilíbrio entre performance e generalização.
- `all-minilm`[7]: modelo compacto e rápido, apropriado para tarefas simples de similaridade com baixo custo computacional.
- `mxbai-embed`[11]: modelo robusto com foco em precisão semântica, adequado para textos técnicos.
- `avrsfr-embed`[8]: com alta dimensionalidade, voltado para granularidade semântica detalhada.
- `snowflake-embed2`[14]: modelo criado para ambientes empresariais, com foco em dados corporativos.

Este sistema processa documentos, armazena *embeddings* e lida com consultas de forma eficiente, tornando-o adequado para aplicações de pesquisa semântica.

Capítulo 3

Trabalho Relacionado

Neste capítulo abordamos as principais abordagens e tecnologias relacionadas com a pesquisa semântica, com especial foco na sua aplicação a sistemas de recuperação de informação em domínios complexos, como o jurídico e o criminal.

Apresentamos as soluções existentes como o Elasticsearch com extensões de NLP, e discutimos as suas limitações quando aplicadas a textos técnicos e extensos, como os registos da Polícia Judiciária.

Esta contextualização serve de base para justificar a escolha dos modelos de *embeddings* utilizados no projeto, avaliados com o objetivo de oferecer uma alternativa mais eficaz e precisa para este tipo de dados.

A pesquisa semântica tem sido amplamente explorada em sistemas de procura de informação, com aplicações em motores de busca, recomendação de conteúdo e análise documental. No entanto, a sua aplicação a registos criminais continua limitada, devido à complexidade dos dados, à terminologia técnica e à necessidade de contextualização profunda.

O Elasticsearch, uma ferramenta popular de busca textual, permite a integração de extensões baseadas em processamento de linguagem natural (NLP), tais como análise de entidades, vetorização de documentos e ranking semântico.

Estas extensões aumentam a capacidade de recuperação além da simples correspondência de palavras-chave.

Ainda assim, mesmo com essas integrações, o Elasticsearch não é ideal para textos longos e jurídicos, pois não foi concebido para capturar relações semânticas profundas nem lidar com ambiguidade contextual, algo crucial na

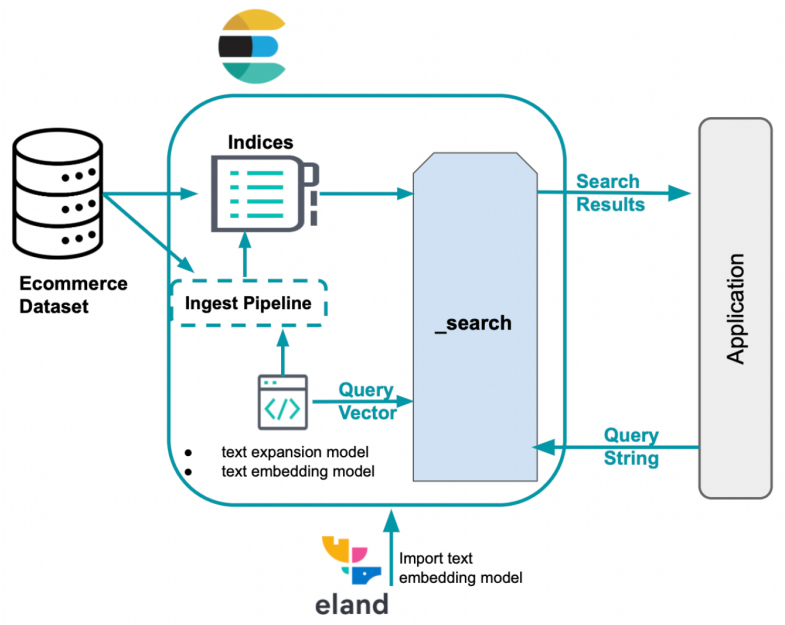

Figura 3.1: Arquitetura do Elasticsearch com extensões de NLP.

análise de relatórios criminais.

Figura 3.2: Crachá da Polícia Judiciária.

Sistemas tradicionais, como os da Polícia Judiciária, dependem de buscas por palavras-chave, limitando a relevância dos resultados.

A API desenvolvida neste projeto distingue-se pelo modelo medcpt-article,

optimizado para textos longos, e pela arquitetura Docker, que permite escalabilidade e gestão dinâmica de dados. Apesar de treinado em relatórios médicos, o medcpt-article demonstrou excelente desempenho em relatórios criminais devido à sua capacidade de captar semântica em textos longos e estruturados. A capacidade de selecionar buckets e os mecanismos de Healthcheck tornam a solução robusta, contribuindo para avanços na procura de informação jurídica.

Aplicações em Outros Domínios

Apesar de o foco deste projeto ser a recuperação de informação em registos criminais, técnicas de pesquisa semântica têm sido amplamente adotadas noutras áreas com dados complexos e linguagem técnica.

Na área da saúde, por exemplo, a pesquisa semântica é utilizada para interpretar consultas clínicas e recuperar documentos médicos relevantes, mesmo quando não há correspondência exata de termos. Modelos como o BioBERT são treinados em textos biomédicos e aplicados em prontuários eletrónicos para apoiar o diagnóstico médico [5].

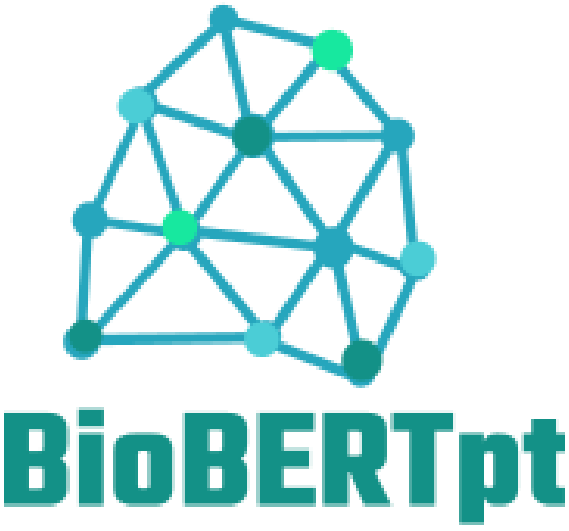

Figura 3.3: Exemplo de pesquisa semântica com BioBERT em relatórios médicos

No setor jurídico, ferramentas como o ROSS Intelligence utilizam modelos baseados em linguagem natural para analisar consultas jurídicas e devolver jurisprudência relevante, mesmo em casos com vocabulário técnico variado. Esta abordagem facilita o trabalho de advogados ao permitir buscas mais inteligentes em bases legais [15].

Figura 3.4: Interface do sistema ROSS Intelligence para pesquisa jurídica

Em ambientes corporativos, soluções como o Microsoft Viva Topics recorrem à semântica para identificar temas recorrentes nos documentos internos, e sugerir conteúdos relacionados, promovendo uma gestão mais eficiente do conhecimento [6].

Estas aplicações reforçam o potencial da abordagem desenvolvida neste projeto para ser generalizada a outras áreas, como saúde, direito ou gestão empresarial, onde a semântica é chave para entender e relacionar grandes volumes de informação textual.

Capítulo 4

Modelo Proposto

Este capítulo apresenta o modelo proposto para a API de pesquisa semântica, delineando os requisitos, fundamentos tecnológicos e a abordagem metodológica, com o objetivo de fornecer uma visão clara dos objetivos do projeto.

4.1 Requisitos

Os requisitos do sistema foram definidos para garantir funcionalidade e robustez, conforme apresentado nas Tabelas 4.1 e 4.2.

Nº	**Requisito**	**Implementado**
1	Pesquisa semântica em linguagem natural	Sim
2	Interface de utilizador para consultas	Sim
3	Seleção de *buckets* para consultas	Sim
4	*Healthcheck* aos *buckets*	Sim
5	Feedback sobre resultados	Sim

Tabela 4.1: Requisitos Funcionais

Nº	Requisito	Implementado
1	Escalabilidade para múltiplos *buckets*	Sim
2	Monitorização de logs	Sim
3	Segurança de dados	Não

Tabela 4.2: Requisitos Não Funcionais

4.2 Fundamentos

A API utiliza técnicas de NLP para processar consultas em linguagem natural, que geram embeddings com o modelo `medcpt_article` via Ollama. Python e Flask formam o backend, enquanto React suporta a interface de utilizador. Docker permite a execução de buckets independentes, e FAISS é usado para indexação e busca de embeddings. A escolha do `medcpt_article` foi baseada em sua capacidade de lidar com textos longos, como relatórios criminais como vamos observar nos resultados do teste.

4.3 Abordagem

O desenvolvimento seguiu uma metodologia iterativa:

- **Prototipagem Local**: Inicialmente, o projeto foi desenvolvido localmente para explorar as técnicas de embeddings e de pesquisa semântica.
- **Otimização de Chunking**: Passou-se de chunking baseado em caracteres para palavras, melhorando a relevância dos resultados.
- **Transição para Docker**: Adotou-se uma arquitetura baseada em contentores para escalabilidade e gestão dinâmica.
- **Seleção de Modelo**: Sete modelos foram testados, com o `medcpt_article` selecionado por sua precisão de 95,1%.

Capítulo 5

Implementação do Modelo

5.1 Arquitetura

A API adota uma arquitetura de microserviços, com Flask para endpoints REST, React para a interface, e Ollama para executar o `medcpt-article` em contentores Docker. Cada *bucket* é um serviço independente, configurado via `docker-compose.yml`, permitindo a adição de dados sem interrupção.

- **Interface**: Desenvolvida em React, permite a seleção de *buckets* e a submissão de consultas em linguagem natural.

- **Buckets**: Serviços independentes que geram e armazenam embeddings dos documentos, processando consultas em paralelo.

- **AI Node**: Coordena o sistema, registra *buckets*, realiza *healthchecks* e encaminha consultas para processamento.

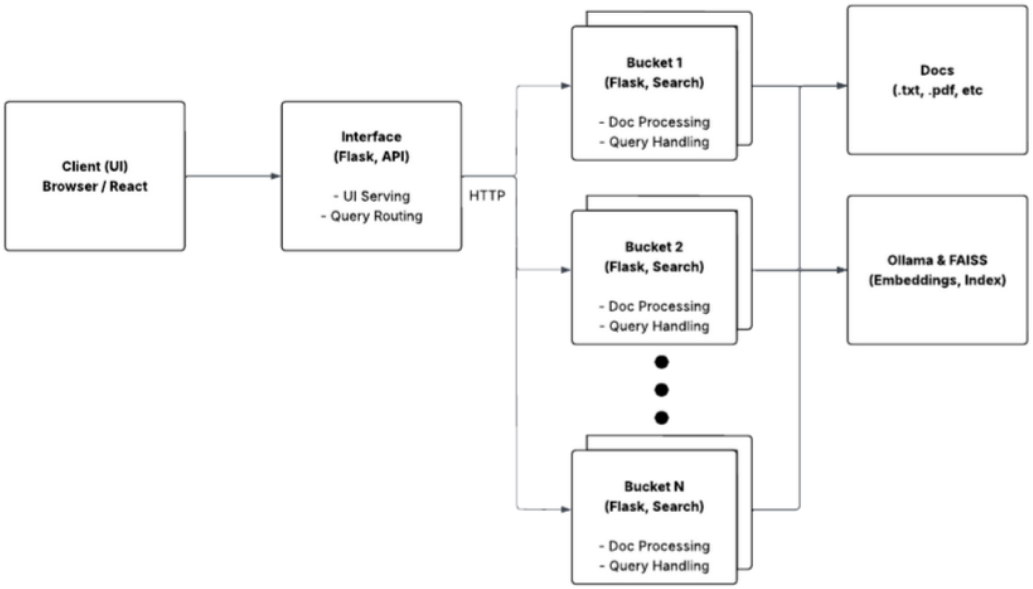

Figura 5.1: Arquitetura geral da API de pesquisa semântica.

5.1.1 Arquitetura da Interface

A interface utiliza componentes React para interação dinâmica, incluindo um seletor de *buckets* e uma área de entrada de texto. O *AI Node* é integrado como ponto central de comunicação, recebendo consultas e retornando respostas processadas, servindo de ponte entre a interface e os *buckets*.

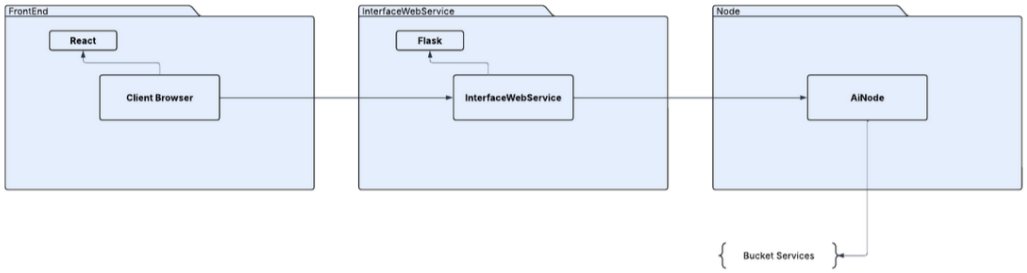

Figura 5.2: Detalhamento da arquitetura da interface em React e integração com o AI Node.

5.1.2 Arquitetura dos Buckets

Os *buckets* são serviços modulares que dividem documentos em *chunks* de 300 palavras, geram embeddings com o modelo `medcpt-article` e os arma-

zenam para futuras consultas. Cada *bucket* funciona de forma independente, permitindo escalabilidade horizontal.

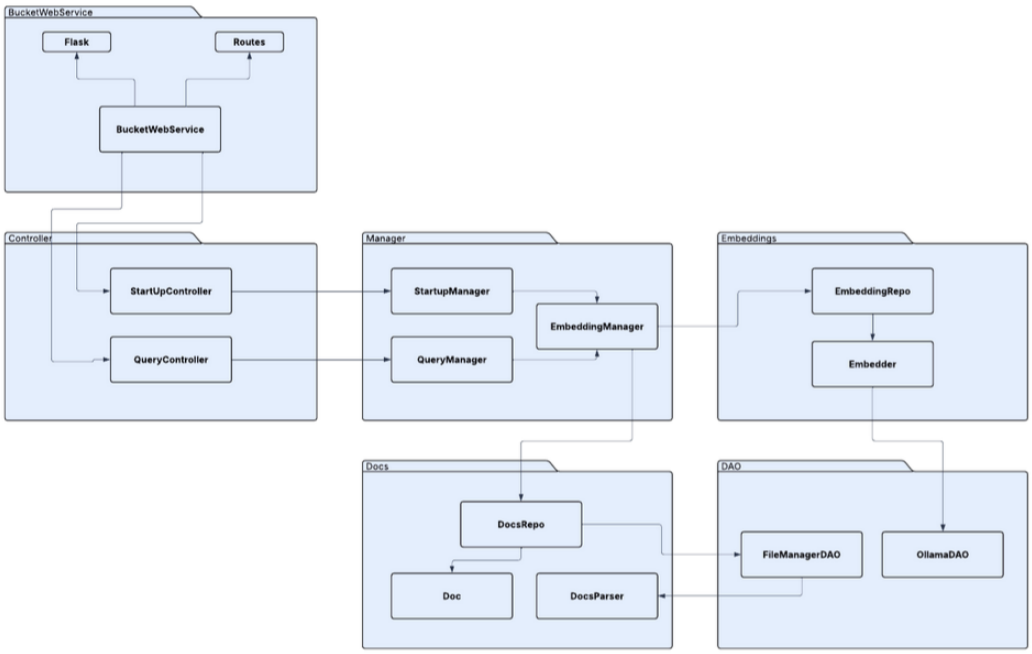

Figura 5.3: Arquitetura dos serviços do tipo bucket, com processamento e armazenamento independente.

5.1.3 Componentes Principais

Os componentes principais do sistema foram projetados para fornecer capacidades de pesquisa semântica, que utilizam *embeddings*, uma arquitetura distribuída de *buckets* e uma interface para interação com o utilizador. Abaixo, detalhamos cada componente e suas funcionalidades.

Criação e Armazenamento de *Embeddings*

Os *embeddings* são representações numéricas de texto que capturam o significado semântico, que permite pesquisas baseadas em similaridade. Este processo envolve duas classes principais: `Embedder.py` e `EmbeddingRepo.py`.

`Embedder.py` A classe `Embedder` gera *embeddings* a partir de texto utilizando o serviço Ollama.

- **Divisão de Texto**: Documentos grandes são divididos em *chunks* para garantir que os *embeddings* representem o contexto local de forma eficaz. O método `chunk_content` divide o texto com base no número de palavras, com tamanho e sobreposição configuráveis.

```
def chunk_content(self, content: str,
    chunk_size_words: int = 300, overlap_words: int =
     150) -> List[str]:
     words = content.split()
     chunks = []
     for i in range(0, len(words) - chunk_size_words
        + 1, chunk_size_words - overlap_words):
          chunk_words = words[i:i + chunk_size_words]
          chunk_text = ' '.join(chunk_words)
          chunks.append(chunk_text)
     if len(words) > chunk_size_words and (len(words)
          - i - 1) > overlap_words:
          chunk_words = words[-chunk_size_words:]
          chunks.append(' '.join(chunk_words))
     return chunks
```

Código 1: Método de chunking em `Embedder.py`

- **Geração de *Embeddings***: O método `generate_embedding` envia um *chunk* de texto ao serviço Ollama via `OllamaDAO` e retorna o *embedding* como um array NumPy.

```
def generate_embedding(self, text: str) -> np.
    ndarray:
     return self.ollama_dao.generate_embedding(text)
```

Código 2: Geração de embedding em `Embedder.py`

`EmbeddingRepo.py` A classe `EmbeddingRepo` lida com o armazenamento e recuperação de *embeddings* utilizando o FAISS, uma biblioteca otimizada para pesquisa de similaridade.

- **Configuração do Índice**: Utiliza um índice `IndexFlatIP` da FAISS com dimensão 768 (correspondente ao tamanho do *embedding*) para pesquisa de similaridade por produto interno.

```
self.faiss_index = faiss.IndexFlatIP(768)
```

Código 3: Configuração do índice FAISS

- **Armazenamento de *Embeddings***: O método `save` normaliza os *embeddings* (para garantir magnitude consistente) e os adiciona ao índice FAISS, associando-os aos metadados do documento.

```
def save(self, doc: Doc, chunks: List[str] = None):
    for i, embedding in enumerate(doc.embeddings):
        norm = np.linalg.norm(embedding)
        normalized_embedding = embedding / norm if
           norm != 0 else embedding
        self.faiss_index.add(np.array([
           normalized_embedding]))
        chunk_text = chunks[i] if chunks and i < len
           (chunks) else ""
        self.docs.append((doc, i, chunk_text))
```

Código 4: Armazenamento de embeddings

- **Pesquisa de *Embeddings***: O método `search` encontra os k *embeddings* mais similares a um vetor de consulta, retorna os detalhes do documento e pontuações de similaridade.

```
def search(self, query_vector: np.ndarray, k: int =
   3) -> List[Tuple[Doc, float, int, str]]:
    norm = np.linalg.norm(query_vector)
    normalized_query = query_vector / norm if norm
       != 0 else query_vector
    search_k = min(k * 2, self.faiss_index.ntotal)
    distances, indices = self.faiss_index.search(np.
       array([normalized_query]), search_k)
    similarities = distances[0]
    results = []
    seen_docs = set()
    for j, i in enumerate(indices[0]):
        if len(results) >= k:
            break
        doc, chunk_index, chunk_text = self.docs[i]
        if doc.name not in seen_docs:
            results.append((doc, similarities[j],
               chunk_index, chunk_text))
```

```
                seen_docs.add(doc.name)
    return results
```

Código 5: Pesquisa de embeddings

Gestão de *Buckets*

Os *buckets* são unidades independentes que processam e armazenam *embeddings* de documentos. A sua gestão envolve registo, processamento de documentos e manipulação de ficheiros.

`BucketWebService.py` A classe `BucketWebService` gere o serviço web do *bucket* e sua interação com o *AI Node*.

- **Configuração**: Utiliza variáveis de ambiente para definir o nome, pasta e URLs de comunicação do *bucket*.

```
self.bucket_name = os.getenv("BUCKET_NAME", "
    default_bucket")
self.bucket_folder = os.getenv("BUCKET_FOLDER", "./
    documents")
self.ai_node_url = os.getenv("AI_NODE_URL", "http://
    interface:5000/ai-node")
self.bucket_url = os.getenv("BUCKET_URL", "http://
    bucket1:5000")
```

Código 6: Configuração do bucket

- **Registo**: Os *buckets* registam-se no *AI Node* enviando seu nome e URL, acionando o processamento de documentos após o registo bem-sucedido.

```
def _register_with_ai_node(self, app):
    response = requests.post(
        f"{self.ai_node_url}/register",
        json={"name": self.bucket_name, "url": self.
            bucket_url}
    )
    response.raise_for_status()
    with app.app_context():
        result = self.startup_controller.startup()
    self.set_processing_complete(True)
```

```
    app.processing_complete = True
```

Código 7: Registo do bucket

`StartUpController.py` A classe `StartUpController` processa os documentos no *bucket* durante a inicialização.

- **Processamento de Documentos**: Corre a pasta do *bucket* em busca de tipos de ficheiros suportados (e.g., `.txt`, `.pdf`) e processa-os utilizando o `EmbeddingManager`.

```
def startup(self):
    bucket_folder = os.getenv('BUCKET_FOLDER', './
       documents')
    for root, _, files in os.walk(bucket_folder):
        for filename in files:
            if os.path.splitext(filename)[1].lower()
                in supported_extensions:
                file_path = os.path.join(root,
                   filename)
                self.embedding_manager.
                   process_document(file_path)
    return jsonify({'message': f'Processados {
       document_count} documentos', 'startup_time':
       total_time})
```

Código 8: Processamento de documentos

`FileManagerDAO.py` A classe `FileManagerDAO` lida com o acesso a ficheiros dentro dos *buckets*.

- **Listagem de Ficheiros**: Recupera uma lista de ficheiros suportados do diretório do *bucket* e suas subpastas.

```
def get_docs(self) -> List[Dict[str, str]]:
    docs = []
    for root, _, files in os.walk(self.directory):
        for filename in files:
            if any(filename.lower().endswith(ext)
               for ext in self.SUPPORTED_EXTENSIONS)
               :
```

```
                file_path = os.path.join(root,
                    filename)
                docs.append({
                    "name": filename,
                    "location": os.path.relpath(
                        file_path, self.directory),
                    "content": ""
                })
    return docs
```

Código 9: Listagem de ficheiros

Gestão de Consultas

As consultas são processadas para encontrar documentos semanticamente similares com o uso dos *embeddings*.

`QueryController.py` A classe `QueryController` lida com requisições de consulta via endpoint web.

- **Endpoint de Consulta**: Aceita requisições POST com uma string de consulta e um valor k, retornando resultados classificados.

```
@app.route("/query", methods=["POST"])
def query_endpoint():
    data = request.get_json()
    query_text = data.get("query", "")
    k = int(data.get("k", 3))
    result = self.embedding_manager.process_query(
        query_text, k)
    return jsonify({
        "results": result["results"],
        "query_time": query_time
    })
```

Código 10: Endpoint de consulta

`QueryManager.py` A classe `QueryManager` processa consultas que geram *embeddings* e faz uma pesquisa no repositório.

- **Processamento de Consultas**: Converte a consulta num *embedding* e recupera as k melhores correspondências.

```
def process_query(self, query: str, k: int = 3) ->
   List[Dict[str, any]]:
    query_vector = self.embedding_manager.embedder.
       generate_embedding(query)
    results = self.embedding_repo.search(
       query_vector, k)
    result_list = [
        {
            "name": doc.name,
            "similarity": float(dist),
            "location": doc.location,
            "chunk": chunk_text,
        }
        for doc, dist, chunk_index, chunk_text in
           results
    ]
    return result_list
```

Código 11: Processamento de consultas

Healthcheck e Comunicação

O sistema garante confiabilidade através de *healthchecks* e comunicação eficiente entre *buckets* e a interface.

`AINode.py` A classe `AINode` supervisiona a saúde dos *buckets* e encaminha consultas.

- ***Healthcheck***: Verifica o estado dos *buckets* a cada 30 segundos, envia uma requisição GET para o endpoint `/status`. Um *bucket* é marcado como vivo se responder com código 200.

```
def healthcheck(buckets):
    while True:
        for bucket in buckets:
            try:
                r = requests.get(f"{bucket['url']}/
                   status", timeout=2)
                bucket['alive'] = r.status_code ==
                   200
```

```
                bucket['processing_complete'] = r.
                    json().get('processing_complete',
                     False)
            except Exception:
                bucket['alive'] = False
                bucket['processing_complete'] =
                    False
        time.sleep(30)
```

Código 12: Healthcheck dos buckets

- **Encaminhamento de Consultas**: Encaminha consultas de forma assíncrona para os *buckets* selecionados e agrega os resultados.

```
async def forward_query(self, query, k,
   selected_buckets):
    results = {}
    async with httpx.AsyncClient() as client:
        for bucket in self.buckets:
            if bucket['name'] in selected_buckets:
                try:
                    url = f"{bucket['url']}/query"
                    resp = await client.post(url,
                        json={"query": query, "k": k
                        }, timeout=10)
                    if resp.status_code == 200:
                        data = resp.json()
                        results[bucket['name']] =
                            data.get("results", [])
                    else:
                        results[bucket['name']] = [{
                            "error": f"Bucket␣
                            retornou␣estado␣{resp.
                            status_code}"}]
                except Exception as e:
                    results[bucket['name']] = [{"
                        error": f"Bucket␣inacessível:
                        ␣{str(e)}"}]
    return results
```

Código 13: Encaminhamento de consultas

`OllamaDAO`

A classe `OllamaDAO` integra-se com o serviço Ollama para gerar *embeddings*.

- **Inicialização**: Conecta-se ao serviço Ollama com um host e modelo configuráveis.

```
def __init__(self):
    self.model = "nomic-embed-text:v1.5"
    ollama_host = os.getenv("OLLAMA_HOST", "http://
       localhost:11434")
    os.environ["OLLAMA_HOST"] = ollama_host
    try:
        ollama.list()
        logger.info(f"Serviço␣Ollama␣inicializado␣
           com␣o␣modelo␣{self.model}")
    except Exception as e:
        logger.error("Serviço␣Ollama␣indisponível")
        raise RuntimeError("Serviço␣Ollama␣
           indisponível") from e
```

Código 14: Inicialização do OllamaDAO

- **Geração de *Embeddings***: Envia texto ao Ollama e recupera o *embedding* como um array NumPy.

```
def generate_embedding(self, text: str) -> np.
   ndarray:
    try:
        response = ollama.embeddings(model=self.
           model, prompt=text)
        embedding = np.array(response["embedding"],
           dtype=np.float32)
        return embedding
    except Exception as e:
        logger.error(f"Falha␣ao␣gerar␣embedding:␣{
           str(e)}")
        raise
```

Código 15: Geração de embedding via Ollama

Componentes Adicionais

- `DocsParser`: Extrai texto de ficheiros (e.g., `.pdf`, `.docx`) com a bibliotecas PyPDF2 e prepara o conteúdo para *embeddings*.

- `DocsRepo`: Integra `FileManagerDAO` e `DocsParser` para procurar e analisar documentos para criar objetos `Doc` com metadados e conteúdo.
- `EmbeddingManager`: Orquestra o processamento de documentos, coordenando `Embedder` e `EmbeddingRepo` para gerar e armazenar *embeddings*.
- `Interface` (`InterfaceWebService` e `main_interface.py`): Serve um frontend baseado em React e encaminha requisições para *buckets* via *AI Node*, proporcionando uma interface amigável ao utilizador.
- ***Logging***: Registo abrangente em todos os componentes auxilia no monitoramento, depuração e optimização de desempenho.

5.2 Métodos da Pesquisa Semântica

5.2.1 Cálculo de Similaridade

A similaridade entre a consulta e os embeddings dos documentos é calculada com a similaridade de cosseno, que mede o cosseno do ângulo entre dois vetores. A fórmula é:

$$\text{similaridade} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

onde $\mathbf{A}$ é o vetor de embedding da consulta e $\mathbf{B}$ é o vetor de embedding do documento.

Por exemplo, se uma consulta como "crimes cibernéticos"tiver o embedding $\mathbf{A} = [1, 0]$ e um documento com "segurança online"tiver $\mathbf{B} = [0.5, 0.5]$, o cálculo é:

$$\text{similaridade} = \frac{1 \cdot 0.5 + 0 \cdot 0.5}{\sqrt{1^2 + 0^2} \cdot \sqrt{0.5^2 + 0.5^2}} = \frac{0.5}{1 \cdot \sqrt{0.5}} \approx 0.707$$

Este valor indica uma similaridade moderada. A comparação final é feita considerando o chunk com a maior pontuação de similaridade em cada documento e não uma média de todos os chunks.

5.2.2 Escolha do Tamanho de Chunk

O tamanho de chunk de 300 palavras foi escolhido após testes que equilibraram tanto o contexto como a eficiência. Chunks menores (e.g., 200 tokens)

aumentam a granularidade, mas elevam o custo computacional. Chunks maiores (e.g., 500 tokens) reduzem custos, mas perdem especificidade nos embeddings. O valor de 300 palavras, embora não seja exatamente a média dos tamanhos testados, mostrou-se ideal para otimizar pontuações de similaridade e desempenho do sistema.

5.3 Interface

Durante o desenvolvimento, foram enfrentadas diversas dificuldades técnicas, entre as quais:

1. **Transição para Docker**: A migração da implementação local para contentores exigiu ajustes nas configurações de redes e volumes, resolvidos com o apoio do orientador.

2. **Otimização de *Chunking***: A abordagem inicial, baseada no número de caracteres, foi substituída por uma segmentação por palavras, o que melhorou a relevância dos resultados.

3. **Filtro de Duplicados**: A filtragem de documentos repetidos limitava o número de resultados. A solução passou por aumentar o número de candidatos retornados pela FAISS antes de aplicar o filtro.

Capítulo 6

Validação e Testes

6.1 Corpus, Consultas e Procedimento de Avaliação

Os testes de desempenho dos modelos de embeddings foram realizados num ambiente local, recorrendo à infraestrutura pessoal do autor, com as seguintes especificações técnicas:

- **Processador**: 13$^{\text{th}}$ Gen Intel(R) Core(TM) i7-13700KF
- **Placa Gráfica**: NVIDIA GeForce RTX 4070 Ti
- **Sistema Operativo**: Windows 11
- **Memória RAM**: 32 GB

O corpus de teste foi constituído por **100 documentos fictícios de registos criminais** em português, criados para simular situações realistas enfrentadas por autoridades judiciais. Os documentos abrangem diversos tipos de crimes e entidades envolvidas. Um exemplo destes documentos encontra-se no Apêndice B, com o título "*A extorsão se torna mortal: violenta retenção dos lobos vermelhos sobre as empresas de Vilkor*".

Foram utilizadas **12 queries** distintas em linguagem natural (ver Secção 6.1.1) para testar a capacidade e robustez semântica dos modelos de embeddings.

A avaliação foi feita com base na métrica de **similaridade média K3**, que calcula a média da similaridade entre a query e os três documentos mais relevantes retornados.

Além da similaridade, foram também analisados os tempos médios de **processamento de documentos** (indexação) e de **tempo de consulta K3**, considerando três tamanhos distintos de *chunk*: 200, 300 e 500 palavras.

6.1.1 Consultas Utilizadas

As consultas foram elaboradas em português, abrangendo tópicos variados e refletindo diferentes tipos de investigações criminais e contextos temáticos:

- "Quais são os relatórios recentes sobre atividades de gangues armadas em Vilkor, Zakovia?"
- "Quais são os relatórios mais recentes sobre roubos de arte envolvendo os Ghost Shadows em Ravenska, Zakovia?"
- "Quais incidentes envolvendo os esquemas de proteção dos Blood Ravens ocorreram perto de Baron's Peak em 2023?"
- "Quais são os casos de contrabando de armas relatados em Sokovia em 2024?"
- "Quais atividades criminosas dos Night Vipers foram documentadas em Krov, Zakovia?"
- "Quais são as políticas de turismo sustentável em Ravenska, Zakovia?"

6.1.2 Verificação de Robustez a Erros Ortográficos

Para verificar a robustez dos modelos face a erros ortográficos, o script `test_model.py` repetiu as queries com variações propositadamente incorretas (por exemplo, atividddades em vez de atividades). As pontuações de similaridade permaneceram altas, com uma queda média de apenas 0.85 para 0.80, demonstrando boa tolerância semântica.

- "Quais são os relatórrios resentes sobbre atividddades de ganguues armadaz em Vilkorr, Zakovia?"
- "Quais são os relatórios mais recntes sobre rouboos de arte enolvendo os Ghoost Shadws em Ravennska, Zakovia?"

- "Quais incidntes envovendo os esquemmas de proteçção dos Blod Ravns ocoreram perto de Baroon?s Peek em 2023?"
- "Quais são os casso de contrabbando de armaz relatadoz em Sokovia em 2024?"
- "Quais atvidades criminossas dos Nigt Viipers foram documntadas em Kroov, Zakovia?"
- "Quais são as polícas de turismmo sustntável em Ravennska, Zakovia?"

6.2 Gráficos de Desempenho por Modelo

Esta seção apresenta os resultados dos modelos testados `medcpt-article`, `medcpt-query`, `nomic-embed`, `all-minilm`, `mxbai-embed`, `avrsfr-embed`, `snowflake-embed2` em documentos em português, com base nas métricas de similaridade média (K3), tempo de processamento de documentos e tempo de consulta K3, para diferentes tamanhos de *chunk* (200, 300, 500 palavras). As figuras de cada modelo são apresentadas individualmente, seguidas de suas análises.

6.2.1 medcpt-article

O modelo `medcpt-article` apresentou a maior similaridade K3 (0.947, 0.951, 0.949 para *chunk sizes* 200, 300, 500), indicando recuperação de documentos altamente relevante. O tempo de processamento de documentos diminui com o aumento do *chunk size* (7.38s, 4.64s, 1.94s), devido à redução de *chunks* (4.7, 3.03, 1.22). O tempo de consulta K3 é estável (0.018-0.020s). Comparado ao `medcpt-query`, tem ligeira vantagem em similaridade, com tempos de processamento semelhantes.

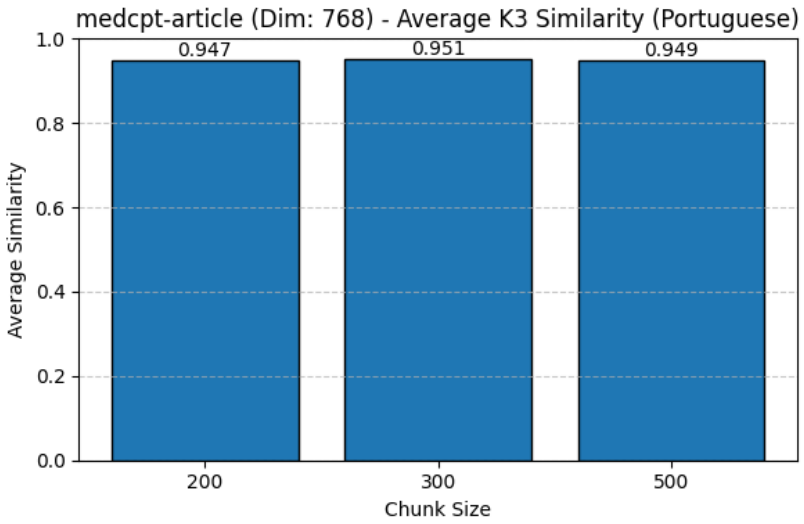

Figura 6.1: Similaridade K3 - `medcpt-article`

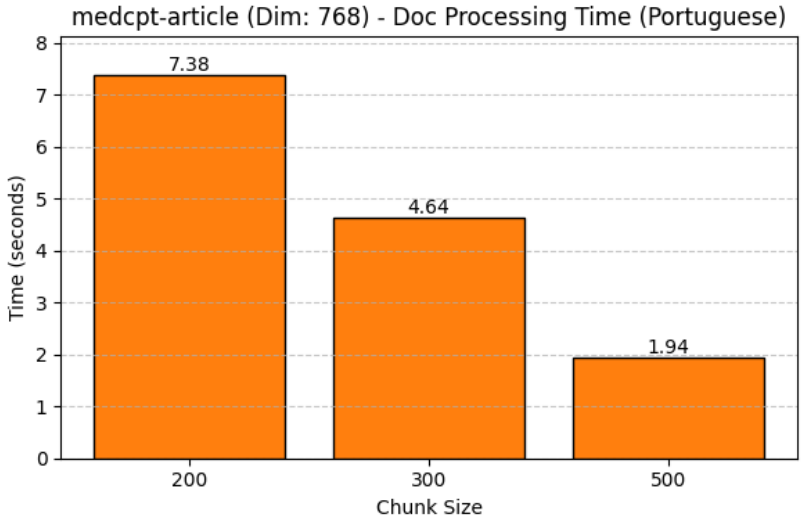

Figura 6.2: Tempo de processamento - `medcpt-article`

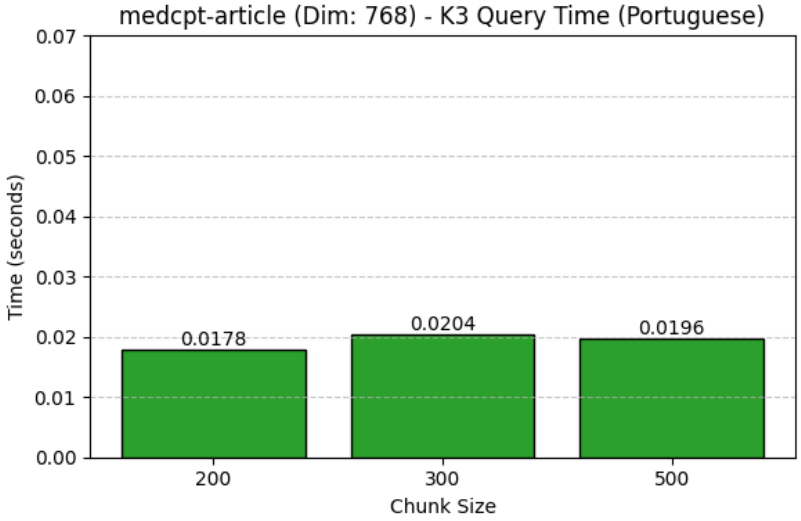

Figura 6.3: Tempo de consulta K3 - `medcpt-article`

6.2.2 medcpt-query

O `medcpt-query` exibiu Similaridade Alta K3 (0.916, 0.918, 0.912 para *chunk sizes* 200, 300, 500), ligeiramente inferior ao `medcpt-article`. O tempo de processamento de documentos é eficiente (7.55s, 4.14s, 1.70s), similar ao `medcpt-article`, e o tempo de consulta K3 é rápido (0.016-0.018s). Supera o `nomic-embed` em similaridade e precisão, mantendo tempos de consulta competitivos.

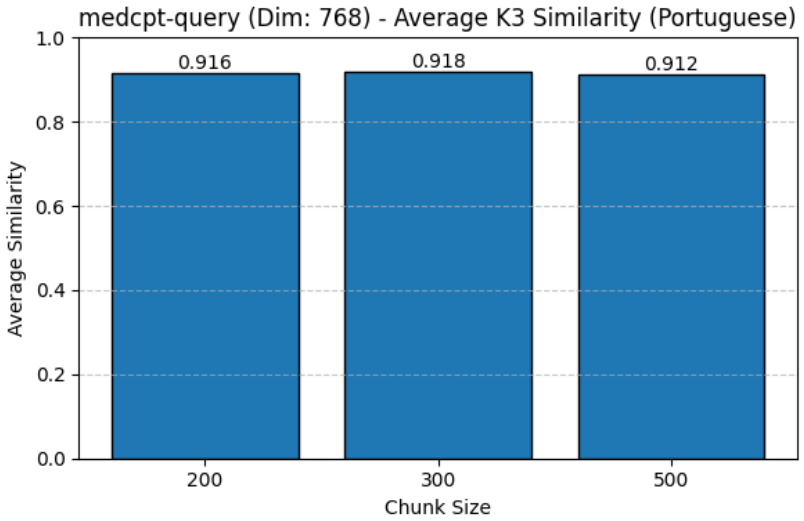

Figura 6.4: Similaridade K3 - `medcpt-query`

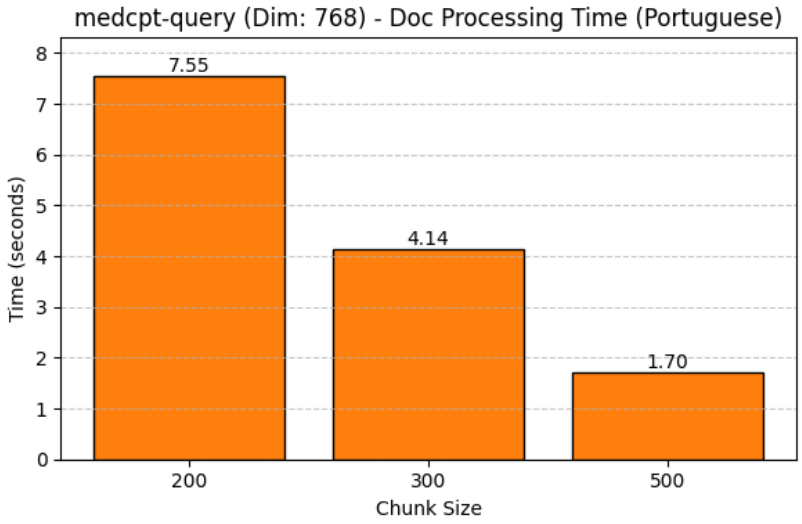

Figura 6.5: Tempo de processamento - `medcpt-query`

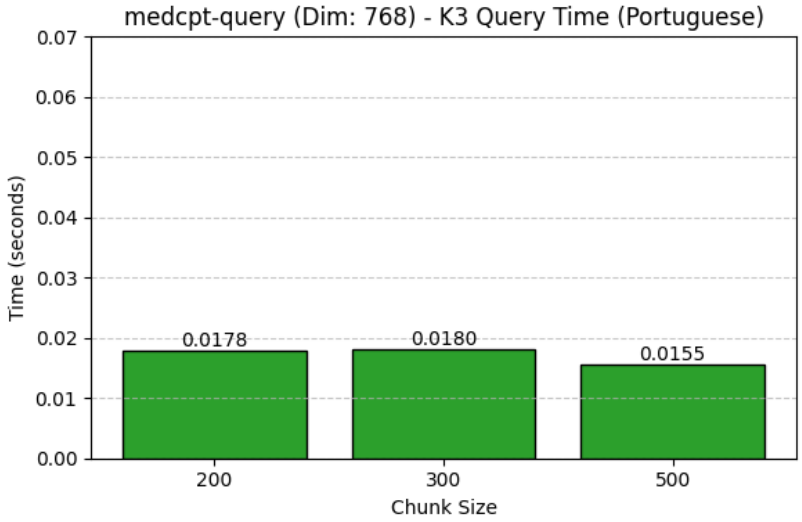

Figura 6.6: Tempo de consulta K3 - `medcpt-query`

6.2.3 nomic-embed

O `nomic-embed` alcançou similaridade K3 moderada (0.757, 0.722, 0.696), inferior aos `medcpt`. O tempo de processamento de documentos é razoável (7.52s, 6.99s, 4.30s), mas ligeiramente superior ao `medcpt-query` para *chunk size* 500. O tempo de consulta K3 é o mais rápido (0.017-0.018s). Compa-

rado ao `mxbai-embed`, tem melhor similaridade e tempos de consulta mais curtos.

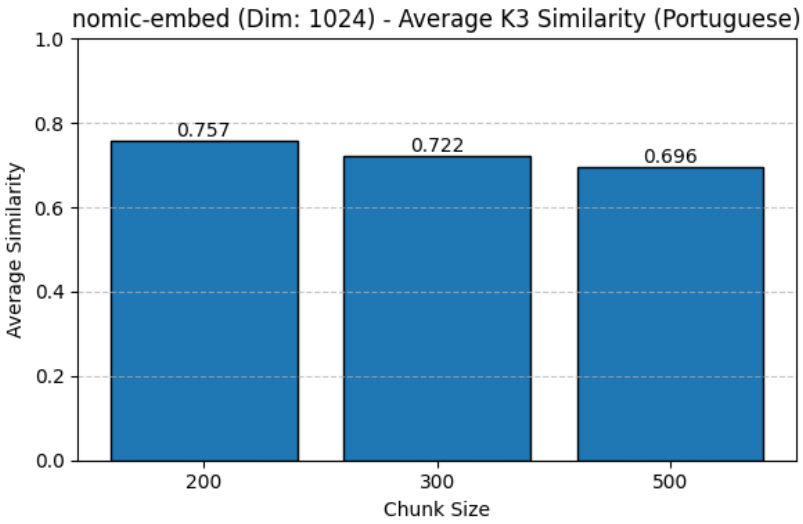

Figura 6.7: Similaridade K3 - `nomic-embed`

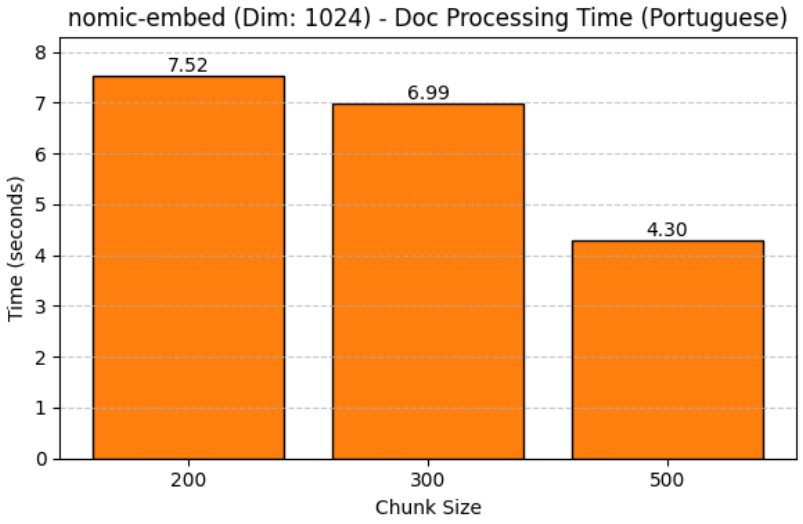

Figura 6.8: Tempo de processamento - `nomic-embed`

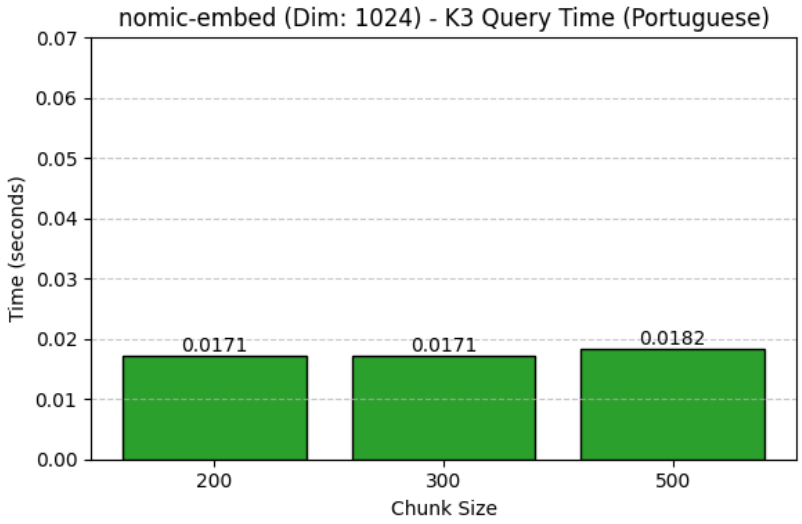

Figura 6.9: Tempo de consulta K3 - `nomic-embed`

6.2.4 all-minilm

O `all-minilm` apresentou a menor similaridade K3 (0.622, 0.607, 0.582), indicando recuperação menos eficaz. O tempo de processamento é competitivo para *chunk size* 500 (1.22s), mas elevado para 200 (9.77s). O tempo de consulta K3 é o mais baixo (0.013-0.015s), mas a Similaridade Baixa limita sua utilidade. Comparado ao `snowflake-embed2`, tem melhor similaridade.

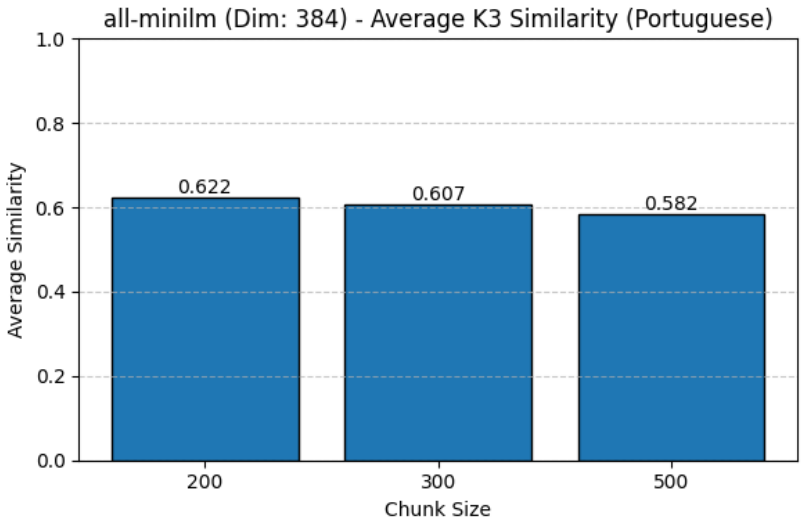

Figura 6.10: Similaridade K3 - `all-minilm`

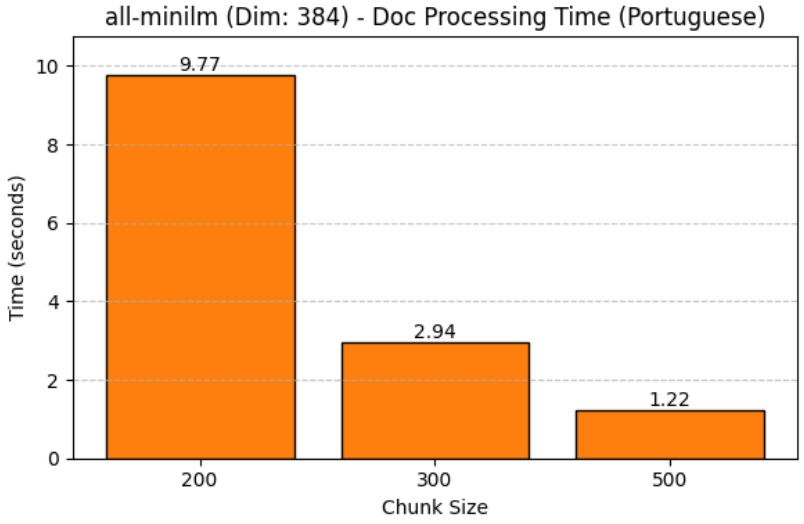

Figura 6.11: Tempo de processamento - `all-minilm`

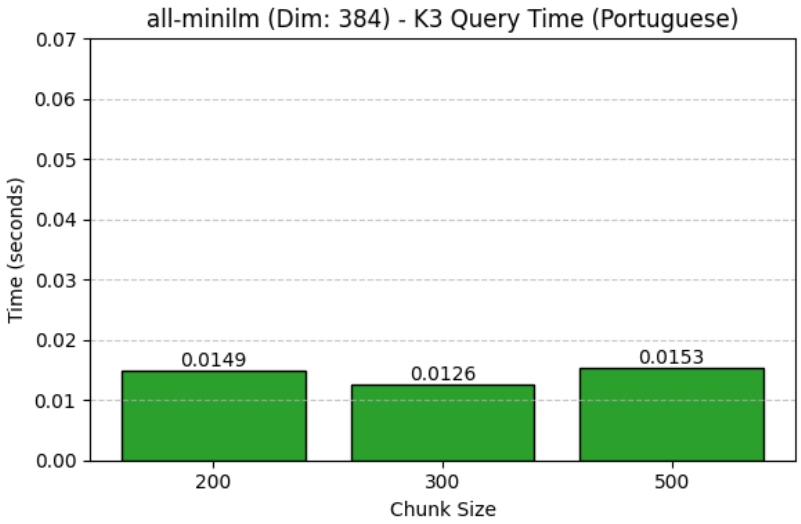

Figura 6.12: Tempo de consulta K3 - `all-minilm`

6.2.5 mxbai-embed

O `mxbai-embed` obteve similaridade K3 razoável (0.769, 0.748, 0.717), inferior aos `medcpt`. O tempo de processamento é elevado para *chunk size* 200 (14.10s), mas reduz para 500 (3.69s). O tempo de consulta K3 (0.029-0.030s) é mais lento que `nomic-embed` e `medcpt`. Comparado ao `avrsfr-embed`, tem melhor similaridade, mas tempos de processamento mais longos para *chunk size* 200.

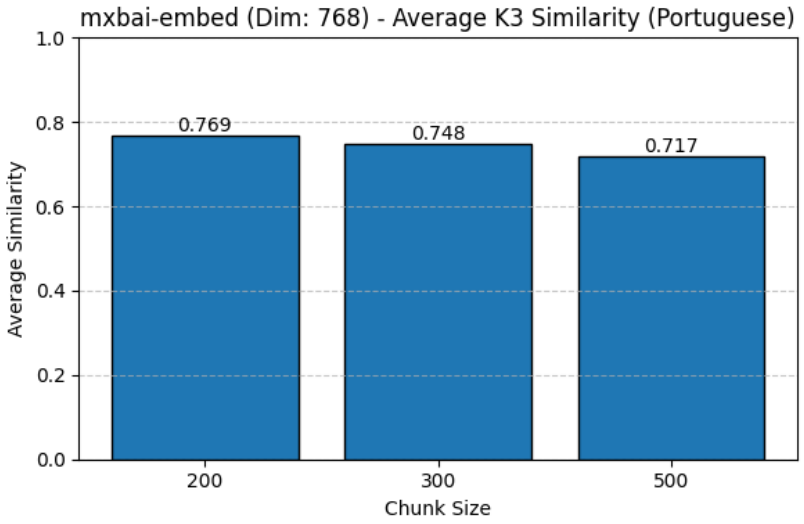

Figura 6.13: Similaridade K3 - `mxbai-embed`

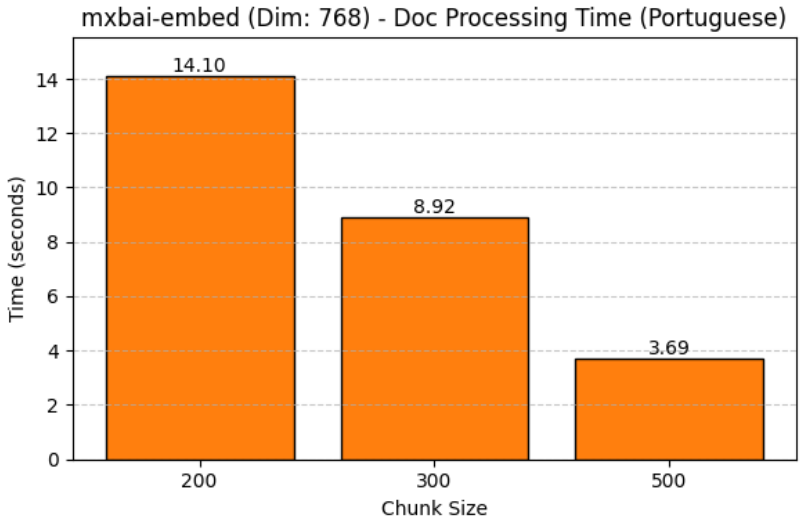

Figura 6.14: Tempo de processamento - `mxbai-embed`

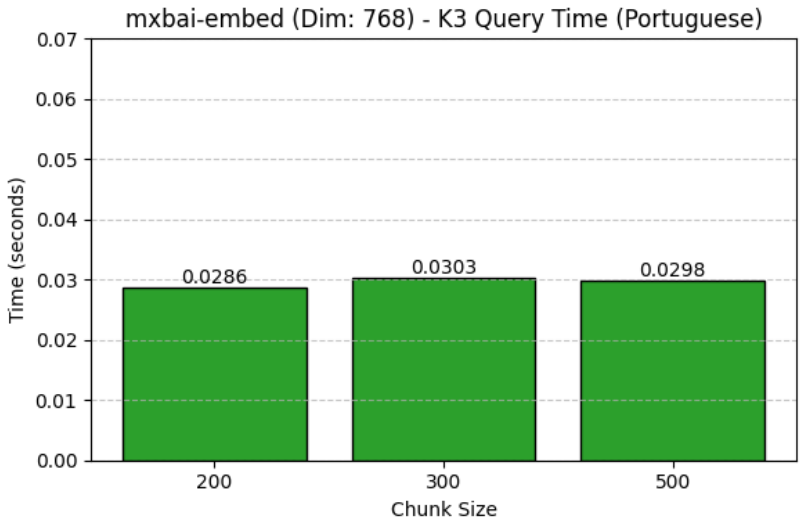

Figura 6.15: Tempo de consulta K3 - `mxbai-embed`

6.2.6 avrsfr-embed

O `avrsfr-embed` apresentou similaridade K3 baixa (0.623, 0.619, 0.605). O tempo de processamento é o mais alto (43.65s, 40.96s, 28.63s), devido ao tamanho do embedding (4096). O tempo de consulta K3 (0.029-0.030s) é semelhante ao `mxbai-embed`. Comparado ao `snowflake-embed2`, tem melhor

similaridade, mas tempos de processamento superiores.

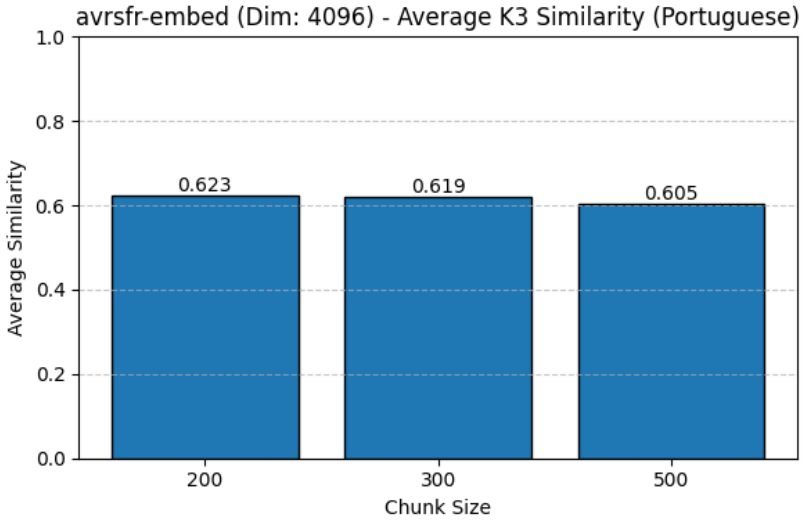

Figura 6.16: Similaridade K3 - `avrsfr-embed`

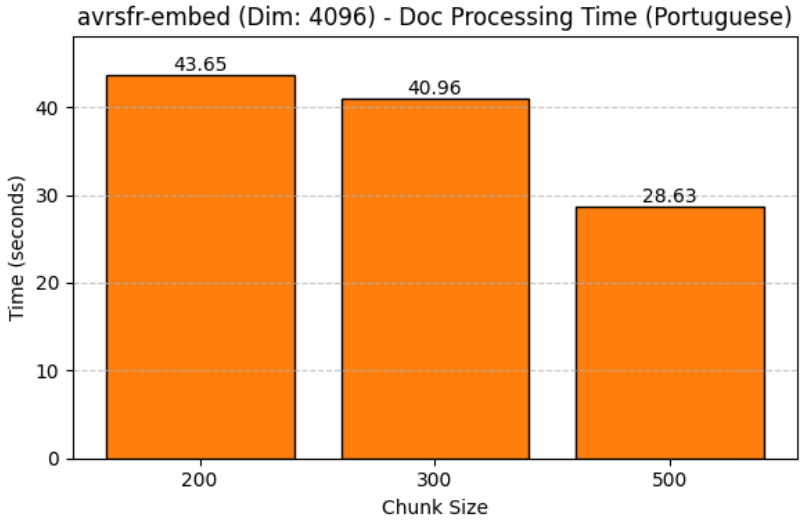

Figura 6.17: Tempo de processamento - `avrsfr-embed`

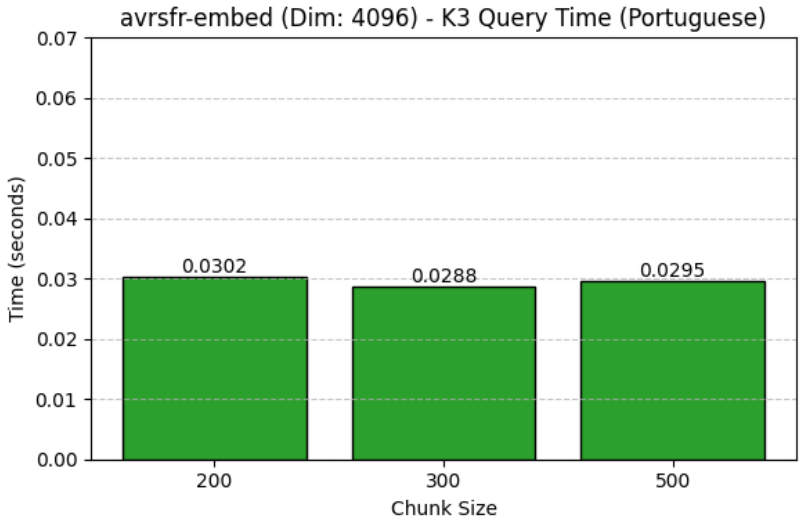

Figura 6.18: Tempo de consulta K3 - `avrsfr-embed`

6.2.7 snowflake-embed2

O `snowflake-embed2` teve o pior desempenho em similaridade K3 (0.570, 0.552, 0.388), inadequado para recuperação de documentos. O tempo de processamento é elevado (24.85s, 15.26s, 8.58s), e o tempo de consulta K3 é o mais lento (0.051-0.063s). É consistentemente inferior em todas as métricas.

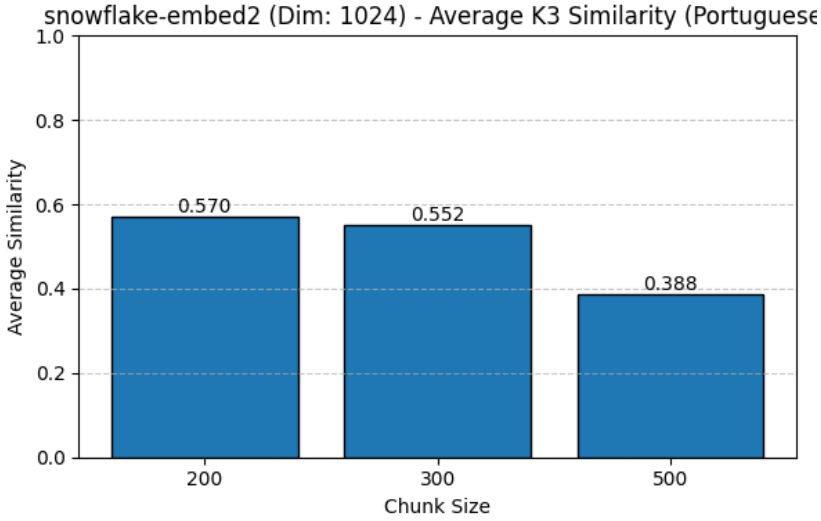

Figura 6.19: Similaridade K3 - `snowflake-embed2`

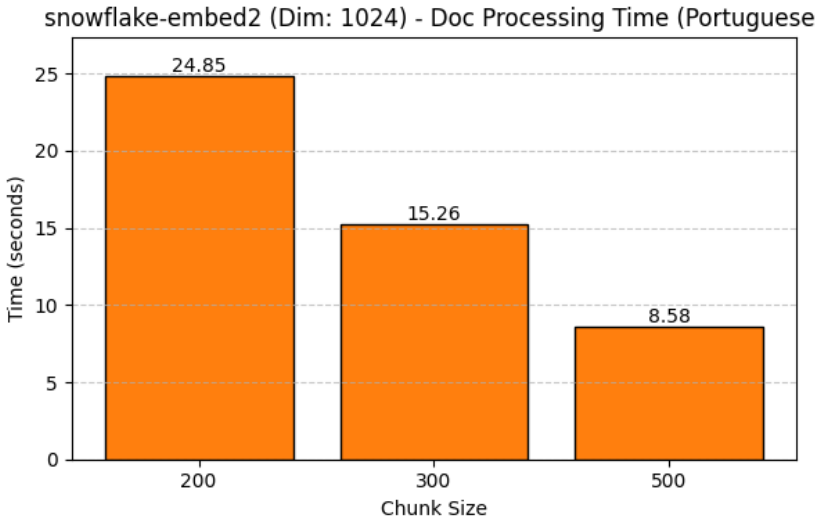

Figura 6.20: Tempo de processamento - `snowflake-embed2`

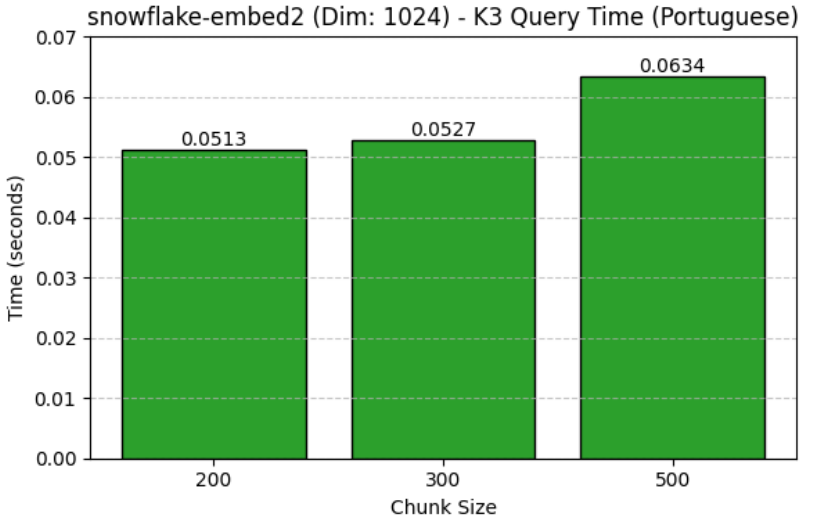

Figura 6.21: Tempo de consulta K3 - `snowflake-embed2`

6.3 Comparação de Modelos

Esta subseção apresenta uma análise comparativa dos modelos, utilizando gráficos gerados pelo script `compare_metrics.py`, destacando diferenças para

`chunk_size_300`.

A Figura 6.22 apresenta a comparação da similaridade média (K3) entre os modelos. Este indicador reflete o quão relevantes são as respostas geradas pelos modelos em relação às consultas feitas, sendo fundamental para avaliar a qualidade dos embeddings utilizados.

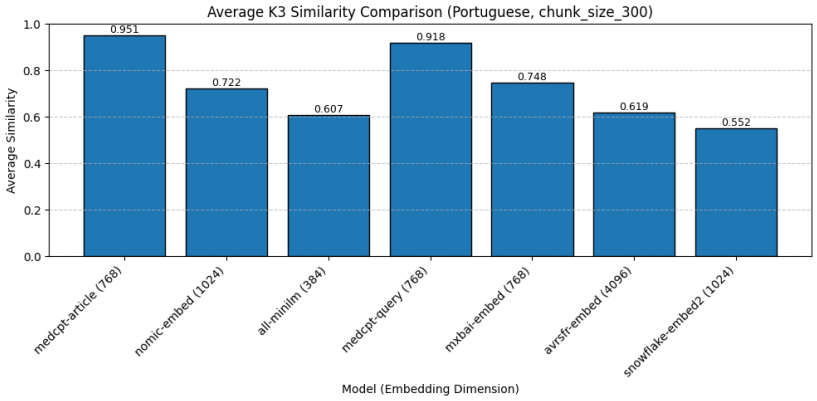

Figura 6.22: Comparação da similaridade média (K3) entre os modelos para *chunk size* 300.

A Figura 6.23 mostra o tempo médio de consulta K3 entre os modelos. Este tempo representa a eficiência dos modelos ao recuperar as três passagens mais semelhantes a uma consulta, sendo um fator importante em aplicações que exijam resposta rápida.

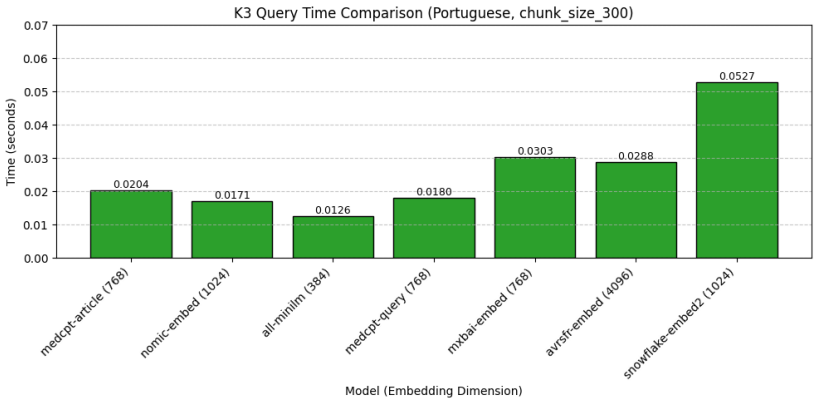

Figura 6.23: Comparação do tempo de consulta K3 entre os modelos para *chunk size* 300.

A Figura 6.24 exibe o tempo de processamento necessário para indexar os documentos. Este tempo afeta diretamente a escalabilidade e o custo computacional das soluções, sendo um critério relevante para aplicações em larga escala.

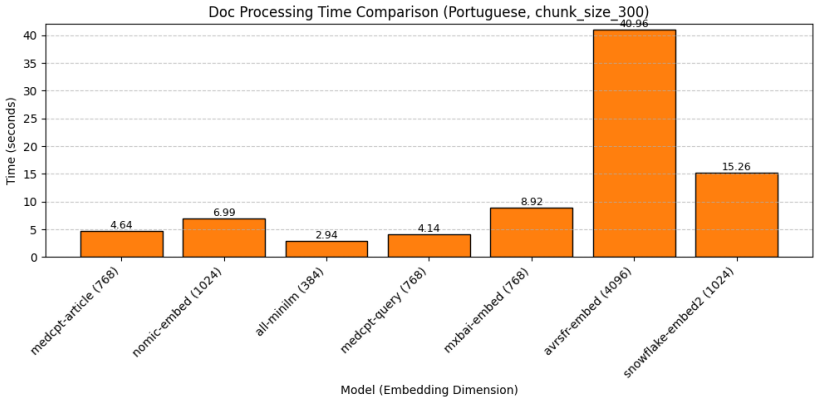

Figura 6.24: Comparação do tempo de processamento de documentos entre os modelos para *chunk size* 300.

6.4 Descrição dos Resultados

O gráfico de dispersão compara sete modelos de embeddings - *MedCPT Article* [9], *Nomic Embed* [12], *All MiniLM* [7], *MedCPT Query* [10], *MxBAI Embed* [11], *AVRSFR Embed* [8] e *Snowflake Embed* [14] - aplicados a tarefas de pesquisa semântica em português com *chunk size* de 300. O eixo x representa o tempo médio de consulta K3 (em segundos), refletindo a velocidade de recuperação das três passagens mais semelhantes. O eixo y apresenta a similaridade média K3, indicando a relevância das respostas retornadas. O gráfico está dividido em quatro quadrantes com base nas medianas: tempo de consulta ($\sim$ 0,0287 segundos) e similaridade ($\sim$ 0,7219), possibilitando uma análise clara entre velocidade e qualidade.

6.4.1 Análise por Quadrantes

Quadrante 1 - Similaridade Alta e Tempo de Consulta Baixo (Superior Esquerdo):
Modelos: *MedCPT Article* [9] (0,9508 / 0,0204s), *MedCPT Query* [10] (0,9181

/ 0,0180s) e *Nomic Embed* [12] (0,7219 / 0,0171s).
Esses modelos oferecem excelente precisão e tempos de resposta rápidos. O *MedCPT Article* é o mais preciso; o *MedCPT Query* é ainda mais rápido; o *Nomic Embed*, embora menos preciso, é o mais ágil dos três.

Quadrante 2 - Similaridade Alta e Tempo de Consulta Alto (Superior Direito):
Modelo: *MxBAI Embed* [11] (0,7476 / 0,0303s).
Apresenta boa similaridade, mas com tempo de consulta superior à mediana. Útil quando a precisão é mais importante que a velocidade.

Quadrante 3 - Similaridade Baixa e Tempo de Consulta Baixo (Inferior Esquerdo):
Modelo: *All MiniLM* [7] (0,6067 / 0,0126s).
Embora muito rápido, apresenta a menor qualidade entre os modelos do quadrante inferior. Adequado para cenários em que rapidez é mais importante que precisão.

Quadrante 4 - Similaridade Baixa e Tempo de Consulta Alto (Inferior Direito):
Modelos: *AVRSFR Embed* [8] (0,6186 / 0,0288s) e *Snowflake Embed* [14] (0,5516 / 0,0527s).
Modelos com o pior desempenho geral: lentos e imprecisos. Não recomendados sem melhorias.

6.4.2 Conclusões Gerais

- **Melhores modelos:** *MedCPT Article* [9] e *MedCPT Query* [10] combinam Similaridade Alta com tempos de consulta baixos.

- **Modelos equilibrados:** *Nomic Embed* [12] e *MxBAI Embed* [11] entregam desempenho razoável em ambas as métricas.

- **Extremos:** *All MiniLM* [7] prioriza velocidade em detrimento da qualidade. *Snowflake Embed* [14] apresenta baixa qualidade e lentidão.

- **Aplicações práticas:** Modelos *MedCPT* são ideais para tarefas de alta precisão. *Nomic Embed* é recomendado para sistemas responsivos. *All MiniLM* serve para contextos com recursos limitados.

6.4.3 Limitações e Trabalho Futuro

- A análise foi realizada com *chunk size* 300 e textos em português. Resultados podem variar em outros idiomas ou tamanhos.

- Tempos de consulta variam conforme o hardware. Processamentos em lote ou com uso de GPU devem ser investigados.

- A divisão em quadrantes baseada na mediana pode ser substituída por critérios adaptados ao domínio.

- Métricas adicionais como $K3_{\text{precision}}$ ou robustez a ruído devem ser exploradas em trabalhos futuros.

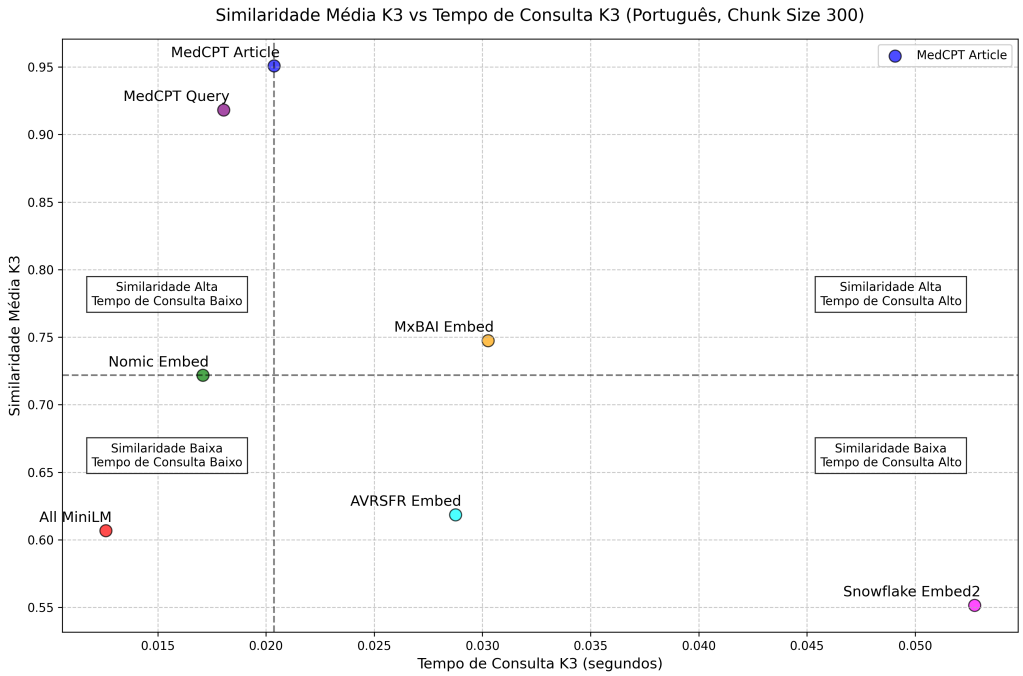

Figura 6.25: Gráfico de Similaridade vs Tempo comparando a similaridade média K3 e o tempo de consulta K3 para modelos de embeddings com *chunk size* 300.

6.5 Script de Teste

6.6 Script `test_model.py`

O script automatiza a avaliação dos modelos:

1. **Processamento de Documentos**: Lê documentos, divide em *chunks* (200, 300, 500 tokens), gera *embeddings* via Ollama.
2. **Indexação com FAISS**: Armazena *embeddings* para busca eficiente.
3. **Execução de Consultas**: Mede similaridade (K3), tempos de consulta e processamento.
4. **Avaliação de Desempenho**: Calcula médias das métricas.
5. **Geração de Resultados**: Exporta para JSON, permitindo análises e gráficos.

6.7 Exemplos de Uso da Interface

A interface em React permite selecionar *buckets*, realizar consultas e visualizar resultados.

Figura 6.26: Tela da interface.

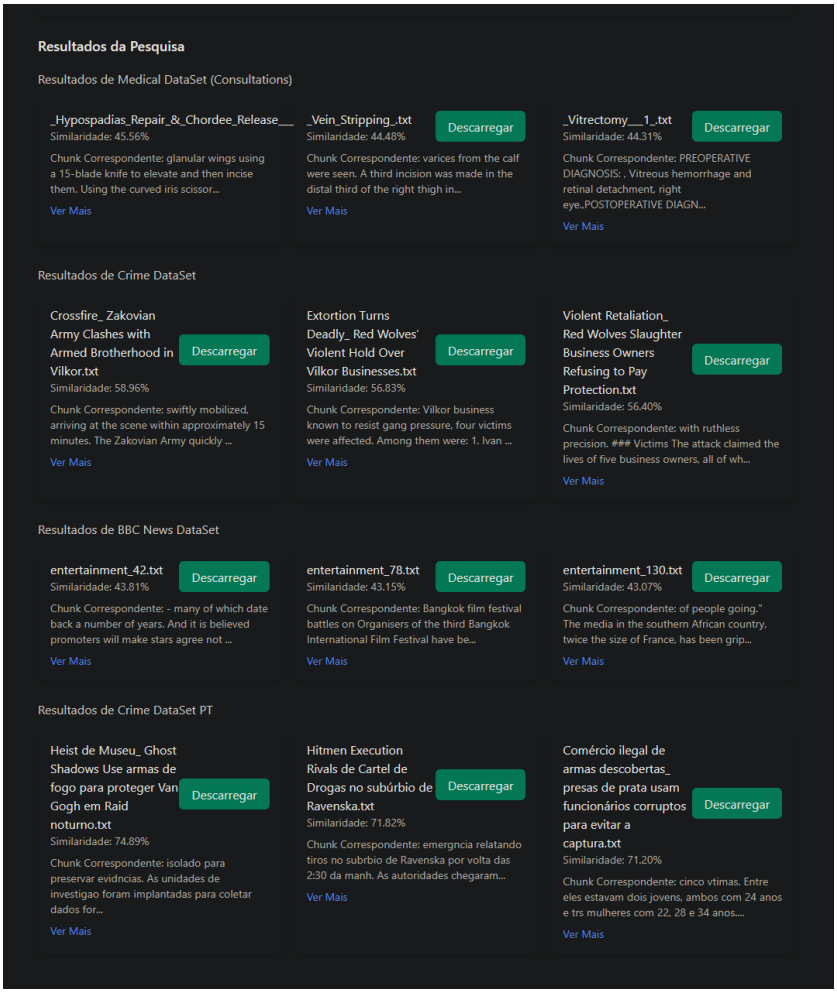

Figura 6.27: Resultado de uma consulta.

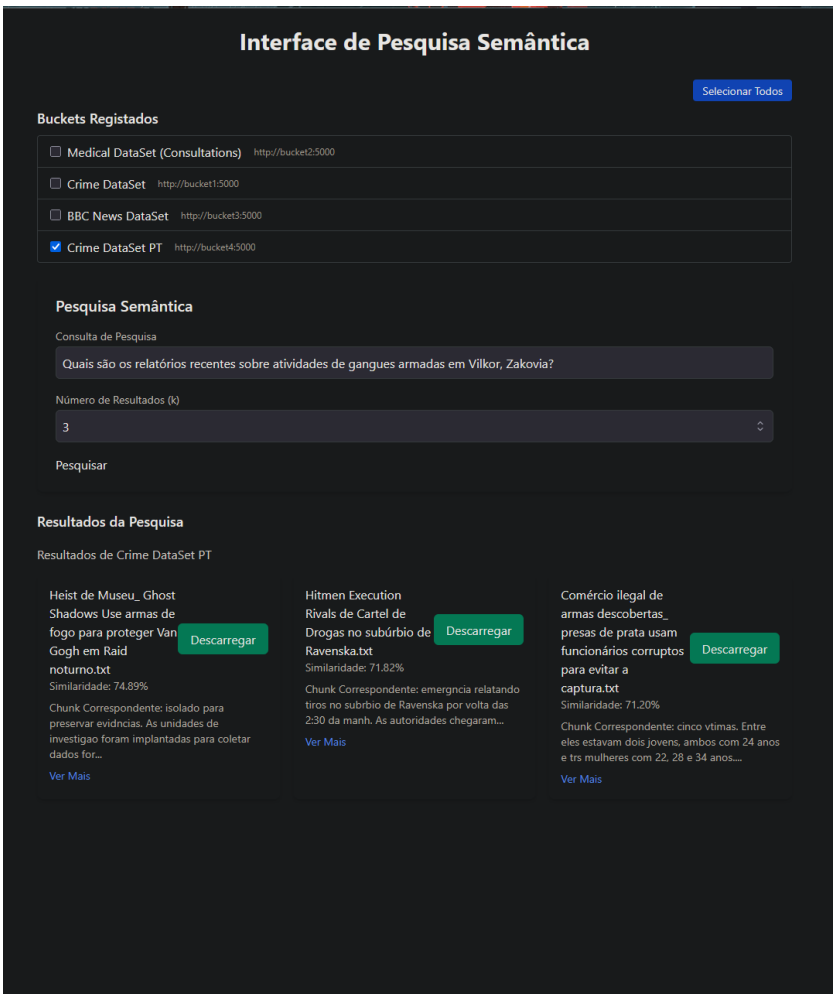

Figura 6.28: Consulta num só *bucket* com $k = 3$.

Figura 6.29: Consulta num só *bucket* com $k = 12$.

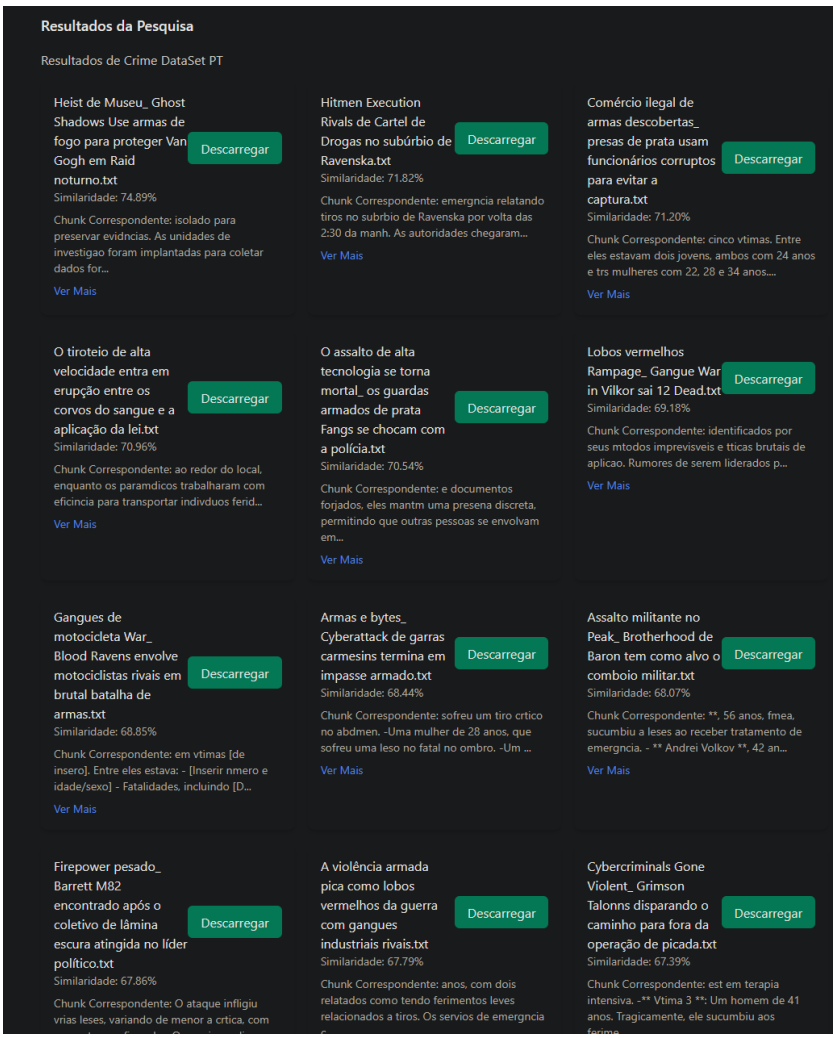

Figura 6.30: Resultados da consulta num só *bucket* com $k = 12$.

Figura 6.31: Demonstração de um *bucket* que falhou.

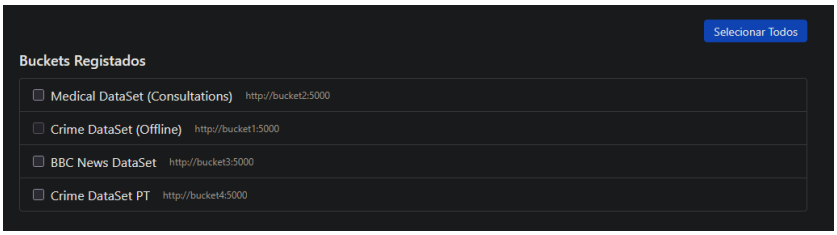

Figura 6.32: Seleção do *bucket* que falhou agora indisponível.

Capítulo 7

Conclusões e Trabalho Futuro

7.1 Conclusão

Este projeto demonstrou, de forma prática e eficaz, como a aplicação de técnicas modernas de processamento de linguagem natural pode transformar a recuperação de informação em domínios sensíveis e complexos, como os registos criminais da Polícia Judiciária. Ao desenvolver uma API de pesquisa semântica robusta, escalável e tolerante a erros ortográficos, foi possível superar limitações das abordagens tradicionais baseadas em palavras-chave, oferecendo assim uma solução mais alinhada com a complexidade dos dados tratados pela PJ.

A escolha do modelo medcpt-article, após uma avaliação de desempenho entre sete alternativas, revelou-se determinante para atingir uma precisão de 95,1%, mesmo que tenha sido originalmente treinado em domínios distintos. Essa capacidade de generalização, aliada a uma arquitetura baseada em microserviços com Docker, React, Flask e FAISS, permitiu construir um sistema modular e eficiente, preparado para evolução contínua.

O projeto seguiu uma abordagem iterativa e prática, enfentrou desafios técnicos como a transição para uma arquitetura containerizada, a definição ideal do tamanho de chunk, e a mitigação de resultados duplicados. Todos esses obstáculos foram ultrapassados com sucesso, consolidando um sistema pronto para possível integração real com sistemas institucionais, como os da Polícia Judiciária, e com potencial de adaptação para outras áreas como saúde ou jornalismo investigativo.

7.2 Principais Conclusões

- A pesquisa semântica demonstrou ser substancialmente mais eficaz do que abordagens tradicionais baseadas em palavras-chave, especialmente em documentos longos e com vocabulário técnico, como os registos criminais.
- A arquitetura modular com buckets independentes garantiu escalabilidade e resiliência, que permitiu a adição dinâmica de novos conjuntos de dados sem comprometer o desempenho do sistema e sem precisar desligar o sistema.
- A utilização de ferramentas como *FAISS*, *Docker*, *Flask*, *React* e *Ollama* revelou-se acertada porque propocionou uma base sólida para o desenvolvimento, integração e expansão da solução.
- A metodologia iterativa adotada foi essencial para a evolução do projeto, permitindo a rápida identificação e resolução de problemas, como o ajuste do tamanho de *chunk*, a filtragem de duplicados e a estabilidade dos serviços em *Docker*.
- A solução desenvolvida provou estar apta não só para a modernização de sistemas judiciais, mas também para ser adaptada a outros contextos onde a recuperação inteligente de informação é crítica.

7.3 Trabalho Futuro

O trabalho futuro prevê o *fine-tuning* do modelo `medcpt_article` em colaboração com a Polícia Judiciária, otimizando-o para dados específicos da PJ. Além disso, planeia-se expandir a API para suportar múltiplos idiomas, melhorar a interface do utilizador com funcionalidades como filtros avançados, otimizar a escalabilidade com técnicas de indexação avançada e explorar funcionalidades como resumo automático de documentos e análise preditiva de padrões criminais, com o objetivo de uma integração eficaz com os sistemas da PJ.

Bibliografia

[1] Decreto-Lei n.º 35.042, de 20 de Outubro de 1945. Criação da Polícia Judiciária.

Webgrafia

- Polícia Judiciária: `https://www.policiajudiciaria.pt/`
- Direção-Geral da Administração da Justiça: `https://dgaj.justica.gov.pt/`
- Docker: `https://www.docker.com/`
- Python: `https://www.python.org/`
- Flask: `https://flask.palletsprojects.com/`
- React: `https://reactjs.org/`
- Ollama: `https://ollama.com/`
- FAISS: `https://faiss.ai/`
- Git: `https://git-scm.com/`
- GitHub: `https://github.com/`
- Elasticsearch: `https://www.elastic.co/elasticsearch/`
- PyPDF2: `https://pypdf2.readthedocs.io/en/latest/`

Apêndice A

Gestão de Código e Controlo de Versões

O projeto foi gerido com recurso ao sistema de controlo de versões *Git*, com o repositório hospedado na plataforma *GitHub*. Foi utilizado um único ramo principal, denominado `master`, concentrando todo o desenvolvimento da aplicação.

O repositório encontra-se organizado em diretórios específicos, separando os scripts em Python, configurações de *Docker*, ficheiros de composição `docker-compose.yml` e os testes automatizados, promovendo uma estrutura modular e facilmente extensível.

Apêndice B

Exemplo de Documento de Teste

Título: *A extorsão se torna mortal: violenta retenção dos lobos vermelhos sobre as empresas de Vilkor*

Resumo: Na noite de 10 de outubro de 2023, ocorreu um violento incidente relacionado a armas na cidade industrial de Vilkor, Zakovia, envolvendo a gangue dos Red Wolves. Essa organização criminosa tem ameaçado empresas locais com extorsão e esquemas de proteção. Durante o ataque a um negócio que resistia à pressão, quatro vítimas foram feridas, uma fatalmente. A polícia respondeu rapidamente, evitando mais vítimas e iniciando uma investigação com foco na gangue, suspeita de ligação com terrorismo internacional.

Trecho do relatório:

> [...] Esse ataque é atribuído à notória gangue de lobos vermelhos, cuja influência criminal atormentou cada vez mais as empresas locais através de suas atividades de extorsão e raquete de proteção. [...] Os Red Wolves se envolvem em uma variedade de atividades criminosas, incluindo assalto à mão armada, sequestro e até tráfico de órgãos. [...] A aplicação da lei também iniciou um bloqueio de bairro enquanto a investigação começou, coletando evidências e perseguindo os autores envolvidos.

Apêndice C

Documentação

C.1 Guia de Utilização

Este apêndice fornece instruções detalhadas para usar a aplicação, baseadas no README, e uma explicação da interface com imagens ilustrativas.

C.1.1 Instruções do README

Para utilizar a aplicação, siga os passos abaixo:

1. Certifique-se de ter o Docker instalado no seu sistema.
2. Clone o repositório do projeto a partir do GitHub `https://github.com/Rexi10/PRJ_55_49734`.
3. Na pasta raiz do projeto, execute o comando `docker-compose up -d` para iniciar os serviços em *background*.
4. Aceda à interface web no endereço `http://localhost:5000/`.
5. Na interface, selecione os *buckets* que deseja consultar, clicando nas caixas de seleção correspondentes.
6. Insira a sua consulta no campo de pesquisa e defina o número de resultados desejados (k).
7. Clique no botão "Pesquisar" para submeter a consulta. Os resultados serão exibidos com o nome do documento, a similaridade, o *bucket* de origem e o *chunk* relevante.

8. Para descarregar um documento, clique no botão “Descarregar” ao lado do resultado correspondente.

Estas instruções permitem configurar e usar a aplicação de forma eficaz.

Bibliografia

[1] Docker. https://www.docker.com/.

[2] Flask. https://flask.palletsprojects.com/.

[3] Python. https://www.python.org/.

[4] React. https://react.dev/.

[5] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. BioBERT: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*, 36(4):1234–1240, 2020. URL https://academic.oup.com/bioinformatics/article/36/4/1234/5566506.

[6] Microsoft. Viva topics: Discover knowledge across your organization. https://www.microsoft.com/en-us/microsoft-viva/topics, 2021. Acesso em 2025.

[7] Ollama. all-minilm embedding model. https://ollama.com/library/all-minilm, 2024.

[8] Ollama. avrsfr-embed model. https://ollama.com/library/avrsfr-embed, 2024.

[9] Ollama. medcpt-article embedding model. https://ollama.com/oscardp96/medcpt-article, 2024.

[10] Ollama. medcpt-query embedding model. https://ollama.com/oscardp96/medcpt-query, 2024.

[11] Ollama. mxbai-embed model. https://ollama.com/library/mxbai-embed, 2024.

[12] Ollama. nomic-embed-text model. https://ollama.com/library/nomic-embed-text, 2024.

[13] Ollama. Plataforma de modelos de embeddings. https://ollama.com, 2024.

[14] Ollama. snowflake-embed2 model. https://ollama.com/library/snowflake-embed2, 2024.

[15] ROSS Intelligence. Artificial intelligence for legal research. https://rossintelligence.com, 2021. Acesso em 2025.