UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA UNIDADE ACADÊMICA DE SISTEMAS EM COMPUTAÇÃO PROJETO EM COMPUTAÇÃO I - 2025.1

DOCENTE: João Arthur Brunet Monteiro

Jamilly Venâncio da Silva - 121210097
Levi de Lima Pereira Júnior - 121210472
Matheus Hensley de Figueiredo e Silva - 120210164
Roberto Pereira dos Santos Filho - 121210587
Vitória Maria do Nascimento - 121210254

LangChain
Relatório Final

CAMPINA GRANDE
2025

SUMÁRIO

1. Introdução

A crescente demanda por sistemas inteligentes capazes de lidar com grandes volumes de informação trouxe à tona desafios relacionados à recuperação de conhecimento específico e à confiabilidade das respostas fornecidas por modelos de linguagem natural (LLMs). Embora esses modelos sejam poderosos, sua limitação está em não possuírem, por padrão, acesso a bases de dados locais ou documentos institucionais.

Nesse contexto, torna-se necessário projetar uma arquitetura de software que integre os pontos fortes dos LLMs com mecanismos de busca e recuperação de informações. O framework LangChain surge como uma solução arquitetural ao fornecer componentes que estruturam esse ecossistema, permitindo a criação de assistentes inteligentes orientados a agentes.

Este trabalho propõe e analisa a arquitetura de um assistente acadêmico voltado a estudantes da UFCG, cujo objetivo é oferecer respostas fundamentadas em documentos institucionais. A compreensão dessa arquitetura é fundamental não apenas para implementar o sistema, mas também para avaliar como diferentes camadas e componentes arquiteturais cooperam para resolver o problema central: facilitar o acesso rápido e preciso a informações oficiais.

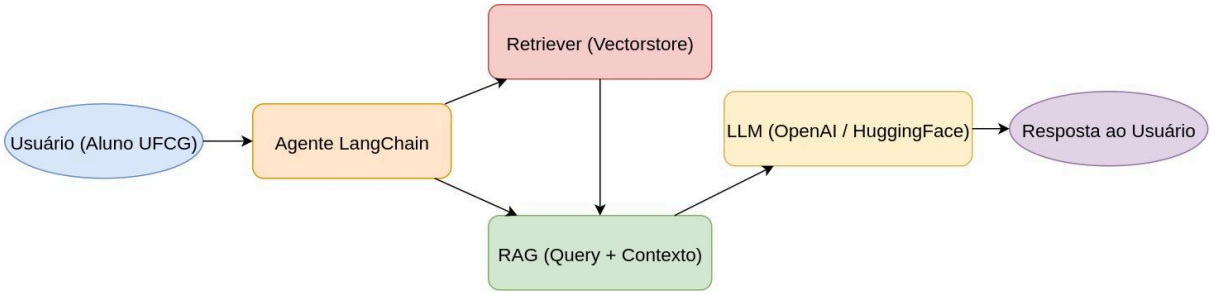

Diagrama de fluxo

1.1 Objetivos

O propósito deste relatório é analisar a arquitetura de um sistema baseado no LangChain, com foco em como sua composição e seus componentes contribuem para a resolução de problemas práticos no contexto acadêmico.

- **Objetivo geral:**
  Compreender e demonstrar como a arquitetura baseada em LangChain, RAG e agentes pode ser empregada para estruturar um sistema de recuperação e geração de respostas fundamentadas em documentos acadêmicos.

- **Objetivos específicos:**

    - Identificar os principais desafios do problema (consultar informações institucionais de forma ágil e confiável).

    - Mapear os componentes arquiteturais do sistema (agente, base vetorial, *retriever*, LLM).

    - Explicar como esses componentes se integram em um fluxo arquitetural para atender às necessidades dos usuários.

    - Avaliar de que maneira a arquitetura proposta contribui para reduzir falhas comuns dos LLMs (como respostas genéricas ou alucinações).

    - Relacionar a arquitetura estudada com o impacto prático para estudantes e instituições de ensino.

2. Fundamentação Teórica

A compreensão da arquitetura do sistema proposto exige uma análise de três eixos conceituais: framework LangChain, padrão RAG e arquitetura orientada a agentes.

2.1 LangChain como framework arquitetural

O LangChain fornece uma camada de abstração arquitetural que organiza o uso de modelos de linguagem em aplicações reais. Sua principal contribuição está em estruturar pipelines modulares, que incluem:

- Ingestão de dados → leitura e processamento de documentos.

- Indexação vetorial → transformação de trechos em embeddings.
- Recuperação de contexto → busca eficiente em bases vetoriais.
- Orquestração por agentes → decisão de rotas de execução.

Dessa forma, o LangChain não é apenas uma biblioteca, mas uma arquitetura de referência que sistematiza como conectar dados, agentes e modelos de linguagem.

2.2 Retrieval-Augmented Generation (RAG)

O RAG é um padrão arquitetural que resolve um problema típico dos LLMs: a dependência de conhecimento prévio estático.

- Sem RAG → o modelo responde apenas com base no que já "aprendeu", correndo risco de passar informações desatualizadas ou imprecisas.
- Com RAG → o modelo é enriquecido com informações recuperadas dinamicamente de uma fonte confiável (neste caso, documentos institucionais da UFCG).

Assim, o RAG atua como um conector arquitetural entre a camada de persistência (base vetorial) e a camada de raciocínio (LLM), garantindo que as respostas sejam contextuais e verificáveis.

3. Metodologia

A metodologia adotada neste trabalho busca demonstrar, de forma estruturada, como a arquitetura baseada em LangChain é organizada para permitir que um assistente acadêmico responda a perguntas fundamentadas em documentos institucionais da UFCG. Mais do que descrever etapas de implementação, esta seção enfatiza como cada componente arquitetural contribui para a resolução do problema de acesso rápido e confiável à informação.

3.1 Coleta e Preparação dos Documentos

Um dos maiores desafios na construção de sistemas de Pergunta e Resposta, baseados em documentos, é lidar com a natureza não estruturada e volumosa das

informações. Modelos de Linguagem (LLMs) não conseguem processar documentos extensos de uma só vez, devido às limitações de sua janela de contexto, por exemplo, alimentar diretamente um PDF de 200 páginas é inviável.

Para viabilizar a recuperação de informações, é necessário segmentar os documentos em partes semanticamente coerentes e convertê-las em representações vetoriais. Assim, cria-se uma base de conhecimento indexada e otimizada para buscas rápidas e contextualizadas. O problema central, portanto, é transformar um grande volume de texto cru em uma estrutura pesquisável e eficiente.

3.1.1 Solução Abordada

A solução implementada consistiu em um pipeline de processamento de documentos, apoiado no ecossistema LangChain, que pode ser dividido nas seguintes etapas:

- **Carregamento (Loading):** Utilizou-se a abstração Document Loaders do LangChain para carregar os arquivos PDF. Especificamente, o **PyPDFLoader** foi empregado para extrair o texto bruto de cada página dos documentos fonte.
- **Divisão (Splitting):** O texto completo foi segmentado em blocos menores e coesos (chunks) através de estratégias como a do **RecursiveCharacterTextSplitter**, que preserva a continuidade semântica entre os trechos. Os chunks são divididos em tamanhos definidos e dentro de cada chunk possui informações do texto em si e os metadados deste texto que também podem ser definidos, como o número da página onde a chunk se encontra no documento original, índice, entre outros.

A implementação desta arquitetura RAG resulta em benefícios diretos para o sistema:

- **Eficiência:** Permite respostas rápidas a perguntas pontuais sobre prazos, requisitos e procedimentos, sem que o usuário precise ler o edital inteiro.
- **Escalabilidade:** A solução é capaz de indexar e consultar diversos editais de diferentes processos seletivos (vestibulares, concursos, bolsas), criando uma base de conhecimento centralizada.
- **Relevância Semântica:** O sistema consegue encontrar a informação correta mesmo que a pergunta não use os termos exatos do documento.

3.1.2 Como o LangChain Resolve

Em vez de construir cada etapa do pipeline do zero, o LangChain atua como um orquestrador que resolve problemas-chave de engenharia de software:

- **Abstração e Modularidade:** Oferece interfaces padronizadas que permitem trocar componentes com facilidade. Essa modularidade aumenta a adaptabilidade do sistema a diferentes fontes e formatos de documentos normativos. Por exemplo, poderíamos facilmente substituir o **PyPDFLoader** por um **WebBaseLoader** para processar uma resolução publicada diretamente no site da universidade, sem alterar a lógica principal do pipeline.
- **Orquestração e Redução de Código:** Gerencia o fluxo de dados desde o carregamento do edital até a indexação dos vetores, reduzindo a quantidade de código repetitivo.

O LangChain não apenas viabiliza a construção do pipeline, mas o torna mais robusto, flexível e fácil de manter.

3.2 Geração de Embeddings

Embeddings são representações numéricas (vetores) que posicionam textos em um espaço latente, permitindo medir similaridade semântica entre frases, palavras ou documentos. No caso de linguagem natural, é necessário realizar um pré-processamento do texto para gerar os *embeddings* adequados.

O texto é inicialmente convertido em *tokens* através do algoritmo *Byte-Pair Encoding (BPE)*. Esses *tokens* representam pedaços das palavras presentes no texto, permitindo que o modelo gere *embeddings* para palavras que nunca viu durante o treinamento. Além disso, essa abordagem ajuda a reduzir o impacto de palavras incorretas no resultado final, já que o modelo foi treinado para prever *tokens* seguintes com base em grandes volumes de texto corretamente escrito.

Cada *token* é transformado em um vetor denso pela camada de *embeddings* do modelo. Esse vetor incorpora informações sintáticas e semânticas dos *tokens*, assim como informações sobre a posição das palavras no texto, por meio da codificação posicional, que pode ser fixa ou aprendida durante o treinamento.

Os vetores são então processados pelas camadas do encoder de um modelo Transformer, até chegar a saída do encoder, gerando ricas representações contextuais, que capturam o significado das palavras levando em conta o contexto em que aparecem.

3.2.1 Solução Abordada

A solução implementada para a geração de embeddings utilizou o LangChain como camada de orquestração, em conjunto com um modelo Transformer pré-treinado disponível em bibliotecas como Hugging Face ou OpenAI. O pipeline foi estruturado para:

- Realizar o pré-processamento do texto (tokenização, normalização e chunking).
- Gerar embeddings contextuais por meio do encoder do modelo Transformer.
- Armazenar os embeddings em um banco vetorial junto com seus respectivos metadados.

Essa abordagem permitiu criar uma base vetorial de documentos eficiente, escalável e pronta para ser consultada a partir de novas entradas textuais.

3.2.2 Como o LangChain Resolve

O LangChain orquestra todo esse processo, chamando as bibliotecas necessárias para a geração dos embeddings do modelo. Quando um novo texto é recebido, ele é processado pelo encoder do Transformer para gerar embeddings contextuais. Em seguida, os embeddings e metadados armazenados no banco vetorial são recuperados, e apenas os embeddings recuperados são utilizados para calcular a similaridade entre o novo embedding e todos os embeddings existentes no banco a partir da equação de similaridade cosseno (distância euclidiana entre dois vetores, e isso é feito para cada par de vetor (V, Vi), onde V é o embedding do novo texto recebido, e Vi é o embedding de cada chunk i armazenada), permitindo recuperar os metadados dos documentos que mais se assemelham semanticamente ao contexto da nova entrada.

3.3 Indexação em Base Vetorial

A indexação em base vetorial surge como resposta ao problema de recuperar informações relevantes em grandes volumes de dados não estruturados, como documentos em PDF, textos longos ou conjuntos de registros. O desafio central é que, diferentemente de buscas tradicionais baseadas em palavras-chave, o objetivo é encontrar trechos semanticamente semelhantes ao que o usuário consulta. Para isso, os textos precisam ser transformados em vetores de alta dimensão (embeddings) que representam o significado semântico das palavras e frases. Na prática, sem o uso de bibliotecas como o LangChain, o processo envolveria três etapas principais: primeiro, a conversão de documentos em embeddings por meio de um modelo pré-treinado, em seguida, o armazenamento desses vetores em uma estrutura especializada de busca, como **Faiss**, **Chroma** ou até mesmo **HNSWLib**, que possibilita cálculos rápidos de similaridade. Essa abordagem demanda implementar manualmente a lógica de ingestão dos documentos, a fragmentação em chunks adequados, a persistência dos vetores e a integração com os algoritmos de busca, o que pode se tornar complexo em sistemas mais amplos.

3.3.1 Solução Abordada

Para a etapa de vetorização, como mencionado, foram empregados os módulos OpenAIEmbeddings e HuggingFaceEmbeddings, que geram representações numéricas dos trechos de texto. Esses vetores, uma vez gerados, foram armazenados em duas opções de bases vetoriais: **Faiss**, que oferece busca em memória com alta performance, e **Chroma**, que garante persistência local dos dados. Assim, o LangChain foi utilizado como camada de integração, abstraindo a complexidade de conectar cada uma dessas etapas em um fluxo contínuo e funcional.

3.3.2 Como o LangChain Resolve

O LangChain resolve o problema original ao fornecer uma abstração de alto nível para todo o processo de indexação vetorial. Em vez de implementar manualmente a sequência de geração de embeddings, configuração de índices e persistência, o framework oferece classes já integradas, como VectorStore e seus adaptadores (FAISS e Chroma). Com isso, o desenvolvedor precisa apenas carregar os documentos, dividi-los em chunks e escolher o modelo de embeddings, enquanto o LangChain cuida da transformação para vetores, do armazenamento no backend escolhido e da interface unificada para consultas posteriores. Essa abordagem reduz drasticamente a complexidade de desenvolvimento e

permite alternar facilmente entre diferentes motores de indexação, sem que seja necessário reescrever a lógica da aplicação. Na prática, o LangChain atua como uma ponte entre modelos de embeddings e bases vetoriais, simplificando a implementação de pipelines de recuperação de informação baseados em similaridade semântica.

3.4 Recuperação de Contexto (Retriever)

A qualidade da resposta de um sistema RAG é diretamente dependente da qualidade dos documentos recuperados. Métodos de busca puramente semânticos, como os baseados em vetores (FAISS, ChromaDB), são excelentes para entender a intenção e o significado por trás de uma pergunta. No entanto, podem apresentar limitações em cenários onde a consulta depende de palavras-chave específicas, como códigos ou acrônimos (e.g. “O que fala o Art. 7º?”). Por outro lado, as abordagens baseadas em busca léxica, como o algoritmo TF-IDF e sua extensão BM25, são eficientes na identificação desses termos exatos, mas carecem de uma compreensão contextual mais profunda. Uma alternativa é a combinação dessas duas abordagens em uma estratégia híbrida, também conhecida como "ensemble de retrievers", que busca integrar os pontos fortes de ambas. Contudo, a implementação dessas diferentes técnicas adiciona uma camada de complexidade ao processo.

3.4.1 Solução Abordada

A solução foi construída progressivamente, explorando as capacidades de recuperação do LangChain:

- **Retriever Vetorial Padrão:** Inicialmente, utilizou-se o método *as_retriever()* das bases FAISS e ChromaDB, que realiza uma busca por similaridade de cosseno no espaço vetorial.
- **Retriever Lexical:** Em seguida, foi implementado o BM25Retriever, que opera sobre o corpus de texto bruto, recuperando documentos com base na frequência e relevância de palavras-chave (TF-IDF).
- **Retriever Híbrido:** Por fim, para unir as duas abordagens, foi utilizado o EnsembleRetriever. Este componente avançado foi configurado para receber uma lista de outros retrievers (neste caso, o retriever do FAISS e o BM25Retriever) e pesos distintos para cada um, permitindo um balanceamento entre a busca semântica e a lexical.

3.4.2 Como o LangChain Resolve

O LangChain não apenas oferece os blocos de construção para implementar essas técnicas, mas também proporciona as ferramentas necessárias para lidar com a complexidade de maneira simplificada:

- **Implementação Simplificada:** Adicionar um retriever robusto como o BM25 se resume a instanciar uma classe. O framework cuida de todo o pré-processamento e da lógica de busca do algoritmo..
- **Interface Padronizada:** Todos os retrievers no LangChain, independentemente de sua lógica interna (vetorial, lexical, etc.), seguem a mesma interface. Isso permite que sejam trocados ou combinados com o mínimo de alteração no código, facilitando a experimentação.
- **Abstração da Complexidade Híbrida:** A maior contribuição está no EnsembleRetriever. Implementar uma busca híbrida manualmente é uma tarefa complexa que envolve: executar buscas em paralelo, normalizar os scores de algoritmos fundamentalmente diferentes (similaridade de vetores vs. scores TF-IDF) e fundir os resultados usando um algoritmo de ranqueamento, como o Reciprocal Rank Fusion (RRF). O EnsembleRetriever abstrai toda essa complexidade. O desenvolvedor apenas fornece os retrievers e os pesos, e o LangChain gerencia o ranqueamento e a fusão dos resultados, entregando uma lista de documentos unificada.

3.5 Retrieval-Augmented Generation (RAG)

Embora a escolha de bons métodos de recuperação seja essencial, ter apenas um retriever e um modelo de linguagem (LLM) não é suficiente. O verdadeiro desafio arquitetural está em conectar esses dois componentes de forma eficiente. É necessário transformar a pergunta do usuário em uma busca, utilizar o retriever para localizar os documentos mais relevantes, incorporar esses documentos ao prompt do LLM de maneira estruturada e, por fim, gerar uma resposta coerente e útil com base nesse contexto. Gerenciar manualmente esse fluxo, desde a recuperação até a geração da resposta, pode ser trabalhoso, sujeito a erros e difícil de manter.

3.5.1 Solução Abordada

A solução foi desenvolvida utilizando as Chains do LangChain, especificamente a RetrievalQA. Esta chain encapsula todo o fluxo do padrão RAG. Ela foi configurada para receber o LLM e o retriever (seja ele o FAISS, Chroma, BM25 ou o Ensemble) e orquestrar a interação entre eles. Arquiteturalmente, essa chain foi encapsulada como uma ferramenta (Tool), um componente que o agente pode escolher executar.

3.5.2 Como o LangChain Resolve

O LangChain resolve este problema de integração de três maneiras fundamentais:

- **Abstração do Fluxo:** A RetrievalQA abstrai toda a lógica de "buscar-e-depois-responder". O desenvolvedor não precisa se preocupar em formatar o prompt com os documentos recuperados ou em gerenciar as chamadas sequenciais. A chain cuida disso, tornando o código mais limpo e declarativo.
- **Modularidade e Reusabilidade:** Ao encapsular a lógica RAG em uma chain, ela se torna um bloco de construção modular. Pode-se facilmente trocar o LLM ou o retriever sem alterar o resto da aplicação. Mais importante, essa chain pode ser transformada em uma Tool, tornando o sistema de perguntas e respostas sobre documentos uma capacidade específica que um agente mais geral pode utilizar.
- **Padronização:** A chain garante que a interação entre o retriever e o LLM siga um padrão testado e otimizado, incluindo funcionalidades como o controle do tipo de formatação de prompt (e.g. *stuff*, *map_reduce*), o que impacta diretamente na qualidade e no custo da resposta.

3.6 Orquestração por Agente

Mesmo com um pipeline bem estruturado entre retriever e LLM, um sistema acadêmico moderno não deve se limitar a responder perguntas a partir apenas de uma base de documentos estática. Usuários reais podem apresentar necessidades mais complexas, como: “Quais são as últimas notícias do site da UFCG?” ou “Compare o que o edital de matrícula diz sobre prazos com as notícias mais recentes”. Nesses casos, um sistema baseado apenas em RAG falharia, pois ele não é capaz de interpretar a intenção do usuário, nem de buscar informações dinâmicas ou externas em tempo real.

Além disso, em sistemas reais, diferentes tipos de consultas exigem rotas distintas de execução. Por exemplo:

- Perguntas sobre documentos institucionais → devem acionar o pipeline de RAG.
- Perguntas sobre notícias atuais da UFCG → exigem uma integração externa para buscar dados recentes.

Sem um agente, seria necessário codificar manualmente regras de decisão, analisando a entrada do usuário, roteando a chamada para a ferramenta adequada e tratando cada caso de forma individual. Esse tipo de lógica condicional rapidamente se torna difícil de manter e escalar conforme o número de funcionalidades cresce.

Dessa forma, um cenário como esse exige um componente adicional: a capacidade de raciocinar sobre a intenção da pergunta e escolher, de forma autônoma, a fonte de informação mais adequada para respondê-la. Esse é um salto de complexidade importante, e representa uma mudança da simples recuperação para um modelo mais flexível de interação.

3.6.1 Solução Abordada

A solução utiliza um componente de Agente do LangChain, especificamente o *initialize_agent* com o tipo *zero shot*. Este tipo de agente utiliza um LLM não para responder diretamente, mas para raciocinar sobre qual ferramenta (Tool) deve ser usada a cada passo. Foram definidas duas ferramentas principais:

- **Ferramenta de Documentos (RAG):** A chain RetrievalQA (usando o EnsembleRetriever), responsável por responder perguntas com base nos PDFs institucionais. Sua descrição informa ao agente que essa ferramenta deve ser usada para consultas sobre regras, editais e documentos oficiais.
- **Ferramenta de Notícias:** Uma função customizada que realiza scraping no portal da UFCG, empacotada como uma Tool. Sua descrição informa ao agente que essa ferramenta é apropriada para obter informações recentes e atualizações da universidade.

3.6.2 Como o LangChain Resolve

O framework de agentes do LangChain é a peça central que confere inteligência e flexibilidade à arquitetura.

- **Motor de Raciocínio (ReAct):** A sigla ReAct significa *Reason* (Raciocinar) e *Act* (Agir). Em vez de apenas gerar uma resposta, o agente opera em um ciclo: ele recebe a pergunta, pensa qual ferramenta usar com base em suas descrições, age executando a ferramenta escolhida, observa o resultado e repete o ciclo até chegar a uma resposta final. O LangChain gerencia todo esse ciclo de raciocínio, fornecendo também a opção de deixar esse “pensamento” transparente.
- **Interface Unificada de Ferramentas:** O LangChain define uma interface Tool simples e poderosa. Qualquer função, chain ou outro componente pode ser convertido em uma ferramenta, bastando fornecer um nome e uma descrição. Isso torna a arquitetura extremamente extensível. Adicionar uma nova capacidade ao assistente (como consultar o calendário acadêmico via API) se resume a criar uma nova função e adicioná-la como uma Tool.