Projeto de Disciplina – Projeto e Análise de Algoritmos (1/2025)

Universidade de Brasília
Departamento de Ciências da Computação

Autores:
Ryan Reis Fontenele (Matrícula: 21/1036132)
João Pedro Gomes (Matrícula: 22/1006351)
Marcus Emanuel Carvalho Tenedini de Freitas (Matrícula: 22/2025960)
Victor Henrique da Silva Costa (Matrícula: 21/2006450)
Guilherme Miranda de Matos (Matrícula: 22/1006431)
Albert Teixeira Soares (Matrícula: 21/1035108)
Arthur Guilherme Souza (Matrícula: 24/1002420)
Kaleb Henrique Silva Pereira (Matrícula: 22/1020904)
Gabriel Francisco de Oliveira Castro (Matrícula: 20/2066571)

Professor:
Jan Mendonça

Brasília, Distrito Federal
Julho de 2025

I. Introdução

Este projeto apresenta um chatbot especializado na Copa do Mundo FIFA, desenvolvido para oferecer aos usuários acesso rápido e preciso a dados históricos do torneio. Nesse sentido, foi utilizado uma arquitetura moderna que combina LlamaIndex e a potência do modelo Llama 3 via Groq,bem como o chatbot emprega a técnica de Retrieval Augmented Generation (RAG) para fornecer respostas informadas. Do ponto de vista organizacional o sistema é dividido em um backend FastAPI e um frontend React, ambos projetados para uma interação fluida. Os dados, essenciais para o funcionamento do chatbot, são extraídos de arquivos CSV que abrangem o histórico completo da Copa do Mundo (1930-2022), incluindo estatísticas de partidas, rankings e informações sobre campeões e artilheiros. Para garantir que o chatbot recupere informações relevantes de forma eficiente, enriquecendo as respostas geradas pelo Llama 3, utilizamos uma combinação de embeddings HuggingFace e FAISS. Com uma configuração simplificada para fácil implantação, o chatbot permite aos usuários explorar o vasto universo da Copa do Mundo com perguntas como "Quantas copas o Brasil tem?"ou "Quem foi o artilheiro da Copa de 2018?". Assim, a experiência de consulta de dados históricos torna-se mais interativa e acessível.

II. Análise do backend

A arquitetura do backend do projeto Chat-sport-PAA é centrada no framework FastAPI, conhecido por sua facilidade de uso para a construção de APIs assíncronas com tipagem de dados, particularmente útil para chatbots como o Chat-sport-PAA.

A. Tecnologias e dependências

As principais tecnologias empregadas no backend foram:

1) **`fastapi`**: framework web assíncrono que gerencia as rotas da API, a validação de requisições e a serialização de respostas. Sua natureza assíncrona (`async`/`await`) é útil para lidar com múltiplas requisições de chatbot concorrentemente sem bloquear o processo principal [5].
2) **`uvicorn[standard]`**: um servidor ASGI (Asynchronous Server Gateway Interface) de alta performance, utilizado para servir a aplicação FastAPI. O Uvicorn é otimizado para aplicações assíncronas e é frequentemente a escolha padrão para deploy de APIs FastAPI em produção [6].
3) **`pandas`**: uma biblioteca para manipulação e análise de dados. Neste projeto, o Pandas é utilizado para carregar e processar os datasets CSV da Copa do Mundo [7].
4) **`llama-index`**: uma estrutura de dados e ferramentas para construir aplicações com LLMs. O LlamaIndex é utilizado para a implementação do RAG (Retrieval Augmented Generation), facilitando a ingestão de dados, a criação de índices e a consulta de informações para enriquecer as respostas do LLM. Embora o código `api.py` não utilize explicitamente todas as funcionalidades avançadas do LlamaIndex (como índices vetoriais complexos), ele aproveita a integração com LLMs e a gestão de prompts [8].
5) **`llama-index-llms-groq`**: um pacote específico do LlamaIndex que fornece a integração com os modelos de linguagem da Groq. Isso permite que o backend utilize os LLMs da Groq de forma simplificada, aproveitando a infraestrutura de inferência de alta velocidade da Groq [9].
6) **`pydantic`**: uma biblioteca para validação de dados e configurações usando anotações de tipo Python. O FastAPI utiliza o Pydantic internamente para validar os corpos das requisições e respostas [10].
7) **`python-dotenv`**: uma biblioteca para carregar variáveis de ambiente de um arquivo `.env`. Essencial para gerenciar credenciais sensíveis, como a `GROQ_API_KEY`, de forma segura e desacoplada do código-fonte, facilitando o desenvolvimento e a implantação em diferentes ambientes [11].

B. Gerenciamento de credenciais e da LLM

Realizamos o gerenciamento da `GROQ_API_KEY` carregando num arquivo `.env` através da função `load_dotenv()`. Isso evita que a chave de API seja hardcoded no código-fonte. A inicialização do LLM é encapsulada na função `initialize_llm()`, que verifica a presença da chave e configura o modelo `llama3-70b-8192` da Groq.

C. Carregamento e otimização de dados

A função `load_csv_datasets()` é responsável por:

1) **Carregamento de CSVs**: Lê os arquivos `world_cup.csv` e `matches_1930_2022.csv` localizados no diretório `wcdataset/`.
2) **Criação de Template**: A criação de um `template` de texto a partir dos dados CSV é uma técnica de RAG simplificada, mas eficaz. Em vez de usar um banco de dados vetorial complexo, o projeto injeta diretamente os dados relevantes no prompt do LLM. O formato `Ano|Sede|Campeão|Vice|Artilheiro` para os dados da Copa do Mundo e um formato similar para os dados das partidas é uma forma de serialização de dados compreendida pelo LLM, embora menos flexível que uma busca vetorial, é extremamente rápida e eficiente para datasets pequenos e bem estruturados.

D. FastAPI

A API FastAPI é bem estruturada, com endpoints claros e bem definidos:

- `@app.on_event(\startup\")`: este marcador do FastAPI é usado para executar a função `initialize_system()` na inicialização da aplicação. Isso garante que o LLM e os datasets sejam carregados antes que a API comece a aceitar requisições, evitando erros em tempo de execução.
- **`/health`**: um endpoint de verificação de saúde é uma prática recomendada para monitorar a disponibilidade e o

estado da aplicação. Ele retorna um status `healthy` ou `unhealthy` dependendo se o LLM e os datasets foram carregados com sucesso.

- **`/chat`**: o endpoint principal da aplicação. Ele utiliza a validação de dados do Pydantic com os modelos `QuestionRequest` e `QuestionResponse`. A lógica de negócio é clara: primeiro, verifica se a pergunta pode ser respondida por uma das "respostas rápidas" pré-definidas (uma forma de cache ou otimização para perguntas frequentes). Se não, a pergunta é passada para a função `process_question_with_llm()`, que a combina com o template de dados e a envia para o LLM da Groq.
- **`/status`**: este endpoint fornece um status mais detalhado do sistema, incluindo o estado do LLM, dos datasets, o tamanho do template e a configuração do modelo. É uma ferramenta útil para depuração e monitoramento.
- **`/` (Raiz)**: O endpoint raiz fornece informações básicas sobre a API, como a versão, descrição e endpoints disponíveis.

III. Análise do frontend

O frontend do projeto Chat-sport-PAA é uma aplicação web interativa, focado na experiência do usuário e na manutenibilidade do código.

A. Tecnologias e dependências

1) **`react` e `react-dom`**: bibliotecas para a construção de interfaces de usuário baseadas em componentes. O React permite a criação de UIs declarativas e eficientes, enquanto o `react-dom` é responsável pela renderização desses componentes no DOM do navegador [12].
2) **`vite`**: uma ferramenta de build de próxima geração que oferece um ambiente de desenvolvimento extremamente rápido [13].
3) **`@vitejs/plugin-react`**: um plugin específico para o Vite que habilita o suporte ao React, permitindo que os desenvolvedores escrevam código React com JSX e outras funcionalidades modernas [13].
4) **`tailwindcss`**: um framework CSS utilitário que permite a construção de designs personalizados diretamente no markup HTML, sem a necessidade de escrever CSS tradicional [14].
5) **`autoprefixer` e `postcss`**: ferramentas que trabalham em conjunto com o Tailwind CSS. O PostCSS é um transformador de CSS que permite o uso de plugins para automatizar tarefas CSS, e o Autoprefixer adiciona automaticamente prefixos de fornecedor aos estilos CSS, garantindo compatibilidade entre navegadores [15].
6) **`lucide-react`**: uma biblioteca de ícones que oferece uma vasta coleção de ícones vetoriais, facilmente integráveis em componentes React. A utilização de ícones é melhora a usabilidade e estética de uma interface de chatbot, fornecendo pistas visuais e melhorando a navegação [16].
7) **`@tiptap/react`, `@tiptap/starter-kit`, `@tiptap/extension-placeholder`, `@tiptap/pm`**: um conjunto de bibliotecas que compõem o Tiptap, um editor de texto rico baseado em ProseMirror [17].
8) **`eslint` e `@typescript-eslint/*`**: ferramentas de linting para JavaScript e TypeScript. O ESLint ajuda a identificar e corrigir problemas de código, garantindo a qualidade, consistência e aderência a padrões de codificação [18].

B. Estrutura de componentes e fluxo de interação

A estrutura do frontend é organizada em componentes React, seguindo o padrão de desenvolvimento modular. O arquivo `App.jsx` atua como o componente raiz, orquestrando a renderização dos principais elementos da interface do chatbot:

- **`App.jsx`**: este componente importa e renderiza `Header`, `MessageList` e `ChatInput`, que são os blocos de construção da interface do chatbot. Ele também gerencia o estado global da aplicação relacionado às mensagens do chat e ao status da API, utilizando hooks personalizados (`useChat` e `useChatEditor`).
- **`Header.jsx`**: exibe o título do chatbot e informações sobre o status da conexão.
- **`MessageList.jsx`**: responsável por renderizar a lista de mensagens trocadas entre o usuário e o chatbot. Ele recebe as mensagens como `props` e gerencia a rolagem automática para a mensagem mais recente (`messagesEndRef`).
- **`ChatInput.jsx`**: o componente de entrada de texto onde o usuário digita suas perguntas. A integração com o Tiptap (`editor` prop) permite uma experiência de digitação mais rica. Ele também lida com o envio de mensagens (`onSendMessage`) e eventos de teclado (`onKeyDown`).
- **`APIModeIndicator.jsx`**: um componente que exibe o status da conexão com a API do backend informando ao usuário se o chatbot está online e funcional.
- **`TypingIndicator.jsx`**: um componente visual que indica quando o chatbot está processando uma resposta, melhorando a experiência do usuário ao fornecer feedback sobre o estado da aplicação.
- **`WelcomeMessage.jsx`**: exibe uma mensagem inicial ao usuário, orientando sobre o uso do chatbot.

O fluxo de interação do usuário é intuitivo:

1. O usuário digita uma pergunta no `ChatInput`.
2. A pergunta é enviada para o backend através da função `sendMessage` (provavelmente definida no hook `useChat`).
3. Enquanto o backend processa, o `TypingIndicator` é exibido.
4. A resposta do backend é recebida e adicionada à lista de mensagens no `MessageList`.
5. A lista de mensagens rola automaticamente para exibir a nova resposta.

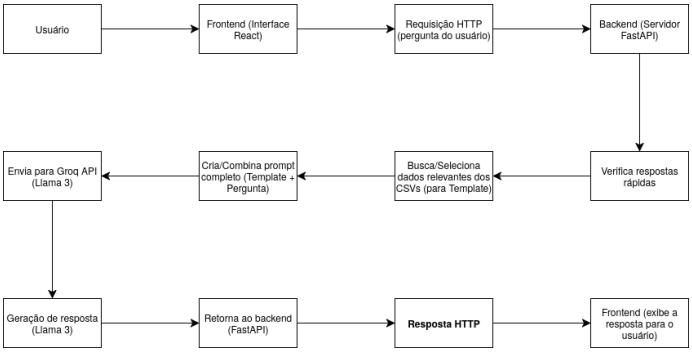

Figura 1. Diagrama do funcionamento do sistema

IV. Análise de Complexidade

Esta seção detalha a análise de complexidade temporal e espacial do projeto Chat-sport-PAA, um chatbot especializado em Copa do Mundo FIFA. A análise abrange tanto o backend, quanto o frontend. Serão examinadas as operações críticas em cada componente para determinar como o desempenho do sistema escala com o aumento do volume de dados e de requisições.

A. Análise de Complexidade do Llama 3

O Llama 3 utiliza a arquitetura Transformer, um modelo de linguagem autorregressivo que utiliza uma arquitetura de transformador otimizada [19], a complexidade dos LLMs que utilizam essa arquitetura é dominada por duas operações principais: o mecanismo de atenção e as camadas totalmente conectadas (Feed-Forward Networks - FFNs).

A complexidade computacional do mecanismo de autoatenção é detalhadamente analisada e mostrada como sendo $O(L^2 \cdot d)$, onde L é o comprimento da sequência e d é a dimensão do modelo (ou da chave/valor). Como d é uma constante para um modelo específico, a complexidade dominante é $O(L^2)$ [1].

Embora a complexidade algorítmica interna do modelo (para treinamento ou processamento batch em cenários específicos) seja $O(L^2)$, para uma inferência típica de geração, o tempo é linear em relação ao comprimento total da sequência gerada. Isso acontece porque o modelo gera um token de cada vez, e para gerar cada novo token, ele precisa "olhar"para todos os tokens anteriores (entrada + os tokens já gerados).

Assim, na prática, para uma única chamada de inferência (quando você envia um prompt e recebe uma resposta), o custo de tempo e computação é diretamente proporcional à quantidade total de tokens que o modelo precisa processar: os tokens de entrada (prompt) e os de saída (resposta gerada).

Mais especificamente, para o modelo `llama3-70b-8192` que utilizamos, o tempo de inferência é aproximadamente linear com o número total de tokens processados. Se `T_in` for o número de tokens de entrada e `T_out` for o número de tokens de saída, a complexidade é **O(T_in + T_out)**.

B. Análise de complexidade do backend

O backend do Chat-sport-PAA é construído com FastAPI e Python, e suas operações principais envolvem o carregamento de dados, a inicialização do modelo de linguagem e o processamento de requisições de chat. A complexidade será analisada para as fases de inicialização e para as operações por requisição.

1) Fase de Inicialização

A fase de inicialização ocorre uma única vez quando a aplicação FastAPI é iniciada. Ela envolve a inicialização do LLM e o carregamento dos datasets.

a) Inicialização do LLM

Esta função configura o cliente Groq LLM. A complexidade aqui é dominada pela alocação de recursos e configuração interna da biblioteca `llama_index.llms.groq`. Embora possa haver operações internas complexas, do ponto de vista da aplicação, é uma operação que ocorre uma vez e não depende do tamanho da entrada do usuário. Portanto, sua complexidade pode ser considerada **O(1)**.

b) Carregamento de Datasets

Esta é a parte mais intensiva da inicialização em termos de dados. A função lê dois arquivos CSV (`world_cup.csv` e `matches_1930_2022.csv`) e processa seus conteúdos para criar um `template` de texto.

- **Leitura de CSVs**: A leitura de arquivos CSV usando `pandas.read_csv` tem uma complexidade que depende do número de linhas (N) e colunas (M) do arquivo. Para um arquivo CSV, a complexidade é aproximadamente **O(N * M)**, pois cada célula precisa ser lida e processada. No caso do `world_cup.csv`, N é o número de edições da Copa do Mundo (22) e M é o número de colunas (9). Para `matches_1930_2022.csv`, N é o número de partidas (centenas) e M é o número de colunas (dezenas). Embora o `matches_1930_2022.csv` seja posteriormente truncado, a leitura inicial ainda processa o arquivo completo.
- **Otimização e Criação do Template**: Após a leitura, os dataframes são iterados para construir o `template` de texto. O `world_cup.csv` é processado completamente (O(N_wc * M_wc)), e o `matches_1930_2022.csv` é limitado a 5 linhas (`df.head(5)`), o que torna essa parte da iteração **O(1)** em relação ao tamanho total do arquivo de partidas. A concatenação de strings para formar o `template` tem uma complexidade que depende do comprimento final do template (L). Em Python, a concatenação de strings pode ser ineficiente se feita repetidamente em um loop, mas para a construção de um único template, é dominada pelo tamanho final L, sendo **O(L)**.

Complexidade Total da Inicialização: A complexidade dominante é a leitura dos arquivos CSV. Se considerarmos N_total como o número total de linhas em todos os CSVs e M_max como o número máximo de colunas em qualquer CSV, a complexidade da fase de inicialização é **O(N_total * M_max)**. Como esta é uma operação de inicialização única, ela não afeta a complexidade por requisição.

2) Operações por Requisição

O endpoint `/chat` é o principal ponto de interação do usuário com o backend. Sua complexidade é crucial para a responsividade do chatbot.

a) Verificação de Respostas Rápidas

Esta função realiza uma busca em um dicionário (`quick_answers`) usando a mensagem do usuário como chave. A busca em um dicionário (hashmap) tem uma complexidade média de **O(1)**. No pior caso (colisões de hash), pode degradar para O(N) onde N é o número de itens no dicionário, mas para um número pequeno e fixo de respostas rápidas, é efetivamente constante.

b) Processamento com LLM

Esta é a operação mais complexa e dominante por requisição.

- **Construção do Prompt**: A construção do `full_prompt` envolve a concatenação do `template` pré-carregado com a pergunta do usuário. A complexidade é **O(L_template + L_question)**, onde L_template é o comprimento do template e L_question é o comprimento da pergunta do usuário. Como o `template` é fixo e a `L_question` é tipicamente pequena, esta parte é relativamente rápida.
- **Chamada ao LLM (`llm.complete`)**: conforme visto na seção anterior, a complexidade da chamada a um Large Language Model (LLM) é dada por **O(T_in + T_out)**. Como `T_in` inclui o textttttemplate completo (que pode ser grande, embora otimizado) e a pergunta, e `T_out` depende da extensão da resposta gerada pelo LLM, esta é a operação mais custosa em termos de tempo de execução por requisição.

Complexidade Total por Requisição: A complexidade dominante por requisição é a chamada ao LLM, que é **O(T_in + T_out)**. Todas as outras operações (verificação de respostas rápidas, construção do prompt) são significativamente mais rápidas.

3) Complexidade espacial do backend

- **Armazenamento de Datasets**: Os dataframes do Pandas e o `template` de texto são carregados na memória. A complexidade espacial é **O(N_total * M_max + L_template)**, onde N_total é o número total de linhas nos CSVs lidos, M_max é o número máximo de colunas, e L_template é o comprimento do template. Como esses dados são mantidos em memória durante toda a vida útil da aplicação, eles contribuem para o consumo de RAM.
- **LLM em Memória**: O modelo LLM em si (ou pelo menos seus componentes de inferência) consome uma quantidade significativa de memória. A complexidade espacial para o LLM é **O(S_model)**, onde S_model é o tamanho do modelo (parâmetros). Modelos como `llama3-70b-8192` são grandes e exigem considerável memória RAM ou VRAM.

Complexidade Espacial Total: A complexidade espacial é dominada pelo tamanho do LLM e pelos dados carregados, sendo **O(N_total * M_max + L_template + S_model)**.

C. Análise de Complexidade do Frontend

O frontend do Chat-sport-PAA é uma aplicação React. A complexidade aqui é avaliada em termos de renderização de componentes e manipulação de estado, que geralmente dependem do número de mensagens na conversa.

1) Renderização de Componentes

O componente `MessageList` é responsável por exibir todas as mensagens da conversa. Cada mensagem é um componente `Message` individual. Quando uma nova mensagem é adicionada, o React re-renderiza a lista.

- **`MessageList`**: Se `K` for o número de mensagens na conversa, a renderização da `MessageList` envolve a iteração sobre `K` mensagens. Cada `Message` componente tem uma complexidade de renderização constante **O(1)** (assumindo que o conteúdo da mensagem não é excessivamente complexo). Portanto, a renderização da lista de mensagens é **O(K)**.
- **`ChatInput` e `Header`**: Estes componentes têm complexidade de renderização **O(1)**, pois seu conteúdo e estrutura não mudam significativamente com o número de mensagens.

2) Manipulação de estado

O estado das mensagens é gerenciado no hook `useChat`. Quando uma nova mensagem é recebida, o array de mensagens é atualizado.

- **Adição de Mensagem**: Adicionar uma nova mensagem a um array (ou lista) em JavaScript é uma operação **O(1)**, assumindo que a operação de `push` ou `concat` é eficiente.
- **Rolagem Automática**: A rolagem para a última mensagem (`messagesEndRef.current?.scrollIntoView()`) é uma operação **O(1)**, pois apenas move a visualização para um elemento específico.

3) Complexidade espacial do frontend

- **Armazenamento de Mensagens**: O frontend armazena todas as mensagens da conversa no estado. Se `K` for o número de mensagens e `L_msg` for o comprimento médio de uma mensagem, a complexidade espacial para armazenar as mensagens é **O(K * L_msg)**. Para conversas muito longas, isso pode levar a um aumento no consumo de memória do navegador.
- **Recursos da Aplicação**: O código JavaScript, CSS, imagens e outros ativos da aplicação contribuem para o consumo de memória, mas são geralmente constantes após o carregamento inicial.

Complexidade Total do Frontend: A complexidade temporal dominante por interação é a renderização da lista de mensagens, **O(K)**. A complexidade espacial é **O(K * L_msg)**.

D. Complexidade dominante do sistema

A operação mais custosa em termos de tempo de execução por interação do usuário é a chamada ao LLM no backend, com complexidade **O(T_in + T_out)**. Esta é a principal determinante da latência percebida pelo usuário. A complexidade

do frontend (O(K)) é geralmente menor, a menos que as conversas se tornem extremamente longas, com milhares de mensagens.

V. Conclusão

O projeto Chat-sport-PAA representa uma iniciativa na aplicação de inteligência artificial conversacional para um domínio de conhecimento específico: a Copa do Mundo FIFA. Através da combinação de tecnologias como FastAPI para o backend e React para o frontend, e a integração com o modelo de linguagem de alta performance da Groq, o projeto demonstra uma arquitetura funcional e responsiva para um chatbot. A estratégia de Geração Aumentada por Recuperação (RAG), embora simplificada pela injeção direta de um template de dados no prompt do LLM, é eficaz para o escopo atual do projeto e para a natureza dos dados históricos utilizados.

No entanto, para que o Chat-sport-PAA evolua para uma aplicação de nível de produção e demonstre um conhecimento mais abrangente em engenharia de software, diversas melhorias e extensões foram propostas. A utilização de mais datasets para expandir a possibilidade de respostas. A implementação de um pipeline RAG completo, com a utilização de embeddings e um banco de dados vetorial, o que permitiria a indexação e recuperação dinâmica de um volume muito maior de dados, superando as limitações atuais do template estático. Além disso, aprimoramentos na experiência do usuário, como a persistência do histórico de conversas e a integração de funcionalidades de voz, bem como a adoção de práticas de DevOps como containerização e CI/CD, são cruciais para a robustez e escalabilidade do sistema.

Referências

[1] ASHISH, Vaswani. Attention is all you need. Advances in neural information processing systems, v. 30, p. I, 2017.

[2] Python Software Foundation. *Abstract Base Classes (abc)*. Disponível em: https://docs-python-org.translate.goog/3/library/abc.html?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc. Acesso em: 05/07/2025.

[3] The GTK+ Project. *Python GTK+ 3 Tutorial*. Disponível em: https://python-gtk-3-tutorial.readthedocs.io/en/latest/. Acesso em: 05/07/2025.

[4] PEREIRA, Geraldo. *Geraldo Pereira* (Canal no YouTube). Disponível em: https://www.youtube.com/watch?v=TjSayD845KI. Acesso em: 05/07/2025.

[5] FastAPI Documentation. (n.d.). *Concurrency and async /await*. Disponível em: https://fastapi.tiangolo.com/async/, Acesso em: 05/07/2025.

[6] Uvicorn Documentation. (n.d.). *Welcome to Uvicorn*. Disponível em: https://www.uvicorn.org/, Acesso em: 05/07/2025.

[7] Pandas Documentation. (n.d.). *pandas documentation*. Disponível em: https://pandas.pydata.org/docs/, Acesso em: 05/07/2025.

[8] LlamaIndex Documentation. (n.d.). *What is lamaIndex?*. Disponível em: https://docs.llamaindex.ai/en/stable/, Acesso em: 05/07/2025.

[9] LlamaIndex Documentation. (n.d.). *Groq*. Disponível em: https://docs.llamaindex.ai/en/stable/api_reference/llms/groq.html

[10] Pydantic Documentation. (n.d.). *What is Pydantic*. Disponível em: https://docs.pydantic.dev/latest/

[11] python-dotenv on PyPI. (n.d.). *python-dotenv*. Disponível em: https://pypi.org/project/python-dotenv/

[12] React Documentation. (n.d.). *React – A JavaScript library for building user interfaces*. Disponível em: https://react.dev/

[13] Vite Documentation. (n.d.). *Vite | Next Generation Frontend Tooling*. Disponível em: https://vitejs.dev/

[14] Tailwind CSS Documentation. (n.d.). *Tailwind CSS - A utility-first CSS framework*. Disponível em: https://tailwindcss.com/

[15] PostCSS Documentation. (n.d.). *PostCSS*. Disponível em: https://postcss.org/

[16] Lucide Icons. (n.d.). *Lucide*. Disponível em: https://lucide.dev/

[17] Tiptap Documentation. (n.d.). *Tiptap - A renderless editor for Vue.js and React*. Disponível em: https://tiptap.dev/

[18] ESLint Documentation. (n.d.). *ESLint - Pluggable JavaScript linter*. Disponível em: https://eslint.org/

[19] LLAMA 3 Disponível em: https://huggingface.co/meta-llama/Llama-3.1-8B Acesso em: 05/07/2025.