Análise Comparativa de Large Language Models (LLMs) na Geração de Documentação para APIs

Gabriela Alves de Ávila

Universidade Federal de Uberlândia
Faculdade de Computação
Bacharelado em Sistemas de Informação

Monte Carmelo - MG
2025

Gabriela Alves de Ávila

Análise Comparativa de Large Language Models (LLMs) na Geração de Documentação para APIs

Trabalho de Conclusão de Curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Área de concentração: Sistemas de Informação

Orientador: Prof. Dr. Adriano Mendonça Rocha

Monte Carmelo - MG
2025

Este trabalho é dedicado à Deus, por sempre estar ao meu lado me fortalecendo não me deixando parar no meio do caminho. Também à minha mãe, que sempre deixou claro que confiava no meu potencial. E aos meus familiares e amigos que sempre estiveram ao meu lado.

Agradecimentos

Primeiramente agradeço a Deus por me permitir estar concluindo esta etapa da minha vida, me guiando e proporcionando forças a todo momento.

Agradeço a minha família, por estarem sempre junto a mim. Em especial a minha mãe, cuja dedicação, paciência, força e conselhos fundamentais para que eu continuasse em frente mesmo em meio aos desafios.

Aos amigos que fiz pelo caminho, que entregaram momentos de distração, apoio, ajuda e companhia, fazendo com que essa jornada tivesse ocasiões divertidas durante os períodos.

Aos professores que contribuíram meu ensino com conhecimento, orientação e dedicação ao longo do curso. Principalmente ao meu professor orientador Adriano Mendonça Rocha, por toda paciência, ajuda, apoio e incentivo durante o desenvolvimento do trabalho.

A todos que, de alguma forma, fizeram parte desta jornada, o meu profundo agradecimento.

"Sua vida pode ser dividida em dois períodos: antes de agora e a partir de agora."
(Prof. Obvious Stating)

Resumo

A Interface de Programação de Aplicativos (API) apresenta diferentes tipos de funcionalidades, sejam elas complexas ou não, contudo entender como utilizá-las pode acabar sendo um problema. Mesmo tendo o auxílio das documentações oficiais das APIs, ainda é possível haver limitações que podem acabar comprometendo na eficiência do desenvolvimento do sistema, como por exemplo a ausência de exemplos de uso práticos para facilitar a aplicação, a falta de informações atualizadas ou a falta de tutoriais para auxiliar no uso das funcionalidades, entre outros.

Atualmente, a utilização de Modelo de Linguagem de Grande Escala (LLM) em tarefas automatizadas tem incentivado a exploração de novas aplicações no campo da programação, principalmente na área de documentação técnica. Mostrando que as documentações das APIs são essenciais em sistemas que possuem menores recursos ou prazos de desenvolvimento.

Este trabalho tem como objetivo analisar a geração de documentação automática para API a partir do uso da Inteligência Artificial (IA), voltado para os seguintes modelos de forma gratuita: ChatGPT, Gemini e *DeepSeek*. Assim, comparando os resultados apresentados por cada uma, com os presentes na Documentação Oficial, e observando as características e limitações distintas de cada uma das LLMs com base em suas amplitudes e integridades.

Através dos resultados, observou-se que os modelos adotam estratégias distintas: Gemini prioriza a precisão mesmo que para isso precise de uma menor cobertura da API, o ChatGPT oferece dados em maior quantidade, porém com riscos de imprecisão, e o *DeepSeek* tenta garantir que haja um equilíbrio entre esses dois critérios.

Logo, a pesquisa destaca o potencial das IAs como ferramentas auxiliares na automatização das documentações de APIs, desde que sejam acompanhadas.

Palavras-chave: Large Language Model, Documentação Automática, API, Inteligencia Artificial, Documentação Oficial.

Abstract

The API offers different types of functionality, whether complex or not. However, understanding how to use them can often become a challenge. Even with the support of official API documentation, there may still be limitations that compromise the efficiency of system development, such as the absence of practical usage examples to facilitate application, the lack of updated information, or the lack of tutorials to assist in using the functionalities, among others.

Currently, the use of large-scale language model LLMs in automated tasks has encouraged the exploration of new applications in the field of programming, especially in technical documentation. This demonstrates that APIs documentation is essential in systems with limited resources or tight development deadlines.

The purpose of this work is to analyze the automatic generation of documentation for APIss through the use of IA, focusing on the following models available for free: ChatGPT, Gemini, and *DeepSeek*. Thus, comparing the results provided by each one with the content present in the Official Documentation, and observing the distinct characteristics and limitations of each LLMs based on their breadth and accuracy.

Through the results, it was observed that the models adopt different strategies: Gemini prioritizes accuracy even if it requires less API coverage, ChatGPT provides a larger amount of data but with risks of imprecision, and *DeepSeek* seeks to ensure a balance between these two criteria.

Therefore, the research highlights the potential of IAs as auxiliary tools in the automation of API documentation, as long as they are monitored.

Keywords: Large Language Models, Automatic Documentation, API, Artificial Intelligence, Official Documentation.

Lista de ilustrações

Figura 1 – Página Principal 36
Figura 2 – Página Web com as Classes da API escolhida como exemplo 37
Figura 3 – Página Web com os exemplos dos métodos da API escolhida como exemplo 37

Lista de tabelas

Tabela 1 – Análise Comparativa dos Trabalhos Correlatos 23
Tabela 2 – Comparativo das classes da API Swing entre diferentes LLMs 40
Tabela 3 – Comparativo dos métodos da classe JButton entre diferentes LLMs . . 40
Tabela 4 – Comparativo dos métodos da classe JFrame entre diferentes LLMs . . 41
Tabela 5 – Comparativo dos métodos da classe JTable entre diferentes LLMs . . . 41
Tabela 6 – Comparativo dos métodos da classe JTree entre diferentes LLMs 42
Tabela 7 – Comparativo dos métodos da classe JCheckbox entre diferentes LLMs . 42
Tabela 8 – Comparativo dos métodos da classe JList entre diferentes LLMs 42
Tabela 9 – Comparativo dos métodos da classe JSlider entre diferentes LLMs . . . 43
Tabela 10 – Comparativo dos métodos da classe JRadioButton entre diferentes LLMs 43
Tabela 11 – Comparativo dos métodos da classe JSpinner entre diferentes LLMs . . 43
Tabela 12 – Comparativo dos métodos da classe JTextField entre diferentes LLMs . 44
Tabela 13 – Média dos métodos listados da API Swing entre diferentes LLMs . . . 44
Tabela 14 – Comparativo das classes da API OpenCV python entre diferentes LLMs 45
Tabela 15 – Comparativo dos métodos da classe VideoCapture entre diferentes LLMs 46
Tabela 16 – Comparativo dos métodos da classe VideoWriter entre diferentes LLMs 46
Tabela 17 – Comparativo dos métodos da classe SIFT entre diferentes LLMs 47
Tabela 18 – Comparativo dos métodos da classe BFMatcher entre diferentes LLMs 47
Tabela 19 – Comparativo dos métodos da classe AKAZE entre diferentes LLMs . . 47
Tabela 20 – Comparativo dos métodos da classe CascadeClassifier entre diferentes LLMs 48
Tabela 21 – Comparativo dos métodos da classe ORB entre diferentes LLMs 48
Tabela 22 – Comparativo dos métodos da classe KAZE entre diferentes LLMs . . . 49
Tabela 23 – Comparativo dos métodos da classe BRISK entre diferentes LLMs . . . 49
Tabela 24 – Comparativo dos métodos da classe MSER entre diferentes LLMs . . . 49
Tabela 25 – Média dos métodos listados da API OpenCV entre diferentes LLMs . . 50
Tabela 26 – Comparativo das classes da API JavaIO java entre diferentes LLMs . . 51

Tabela 27 – Comparativo dos métodos da classe BufferedInputStream entre diferentes LLMs 51
Tabela 28 – Comparativo dos métodos da classe BufferedOutputStream entre diferentes LLMs 52
Tabela 29 – Comparativo dos métodos da classe BufferedReader entre diferentes LLMs 52
Tabela 30 – Comparativo dos métodos da classe BufferedWriter entre diferentes LLMs 52
Tabela 31 – Comparativo dos métodos da classe ByteArrayInputStream entre diferentes LLMs 53
Tabela 32 – Comparativo dos métodos da classe ByteArrayOutputStream entre diferentes LLMs 53
Tabela 33 – Comparativo dos métodos da classe CharArrayReader entre diferentes LLMs 53
Tabela 34 – Comparativo dos métodos da classe CharArrayWriter entre diferentes LLMs 54
Tabela 35 – Comparativo dos métodos da classe Console entre diferentes LLMs . . 54
Tabela 36 – Comparativo dos métodos da classe DataInputStream entre diferentes LLMs 54
Tabela 37 – Média dos métodos listados da API JavaIO entre diferentes LLMs . . . 55
Tabela 38 – Comparativo das classes da API Requests java entre diferentes LLMs . 55
Tabela 39 – Comparativo dos métodos da classe Request entre diferentes LLMs . . 56
Tabela 40 – Comparativo dos métodos da classe Response entre diferentes LLMs . 56
Tabela 41 – Comparativo dos métodos da classe PreparedRequest entre diferentes LLMs 57
Tabela 42 – Comparativo dos métodos da classe Session entre diferentes LLMs . . . 57
Tabela 43 – Comparativo dos métodos da classe HTTPAdapter entre diferentes LLMs 57
Tabela 44 – Comparativo dos métodos da classe BaseAdapter entre diferentes LLMs 58
Tabela 45 – Comparativo dos métodos da classe RequestsCookieJar entre diferentes LLMs 58
Tabela 46 – Comparativo dos métodos da classe AuthBase entre diferentes LLMs . 58
Tabela 47 – Comparativo dos métodos da classe HTTPDigestAuth entre diferentes LLMs 59
Tabela 48 – Comparativo dos métodos da classe HTTPBasicAuth entre diferentes LLMs 59
Tabela 49 – Média dos métodos listados da API Requests entre diferentes LLMs . . 59
Tabela 50 – Média dos resultados que utilizaram a linguagem Java 60
Tabela 51 – Média dos resultados que utilizaram a linguagem *Python* 60

Tabela 52 – Média das métricas do ChatGPT, considerando a quantidade de métodos nas classes das APIs avaliadas, agrupadas por intervalos de número de métodos. 61
Tabela 53 – Média das métricas do Gemini, considerando a quantidade de métodos nas classes das APIs avaliadas, agrupadas por intervalos de número de métodos. 62
Tabela 54 – Média das métricas do DeepSeek, considerando a quantidade de métodos nas classes das APIs avaliadas, agrupadas por intervalos de número de métodos. 62

Lista de siglas

GPT Transformador pré-treinado generativo - *Generative Pre-trained Transformer*

LLM Modelo de Linguagem de Grande Escala - *Large Language Model*

API Interface de Programação de Aplicativos - *Application Programming Interface*

IA Inteligência Artificial - *Artificial Intelligence*

PLN Processamento de Linguagem Natural

HTML *HyperText Markup Language*

CSS *Cascading Style Sheets*

Sumário

1 INTRODUÇÃO 14
1.1 Motivação 15
1.2 Objetivos 15
1.2.1 Objetivo Principal 15
1.2.2 Objetivos Específicos 16
1.3 Organização da Monografia 16

2 FUNDAMENTAÇÃO TEÓRICA 17
2.1 Conceitos Fundamentais 17
2.1.1 Documentação oficial de APIs 17
2.1.2 Processamento de Linguagem Natural - PLN 17
2.1.3 Large Language Models - LLMs 18
2.1.4 API Gemini 18
2.1.5 HTML 18
2.1.6 CSS 19
2.1.7 Python 19
2.1.8 JavaScript 19
2.1.9 Flask 19
2.2 Trabalhos Correlatos 20

3 ABORDAGEM PARA GERAÇÃO DE DOCUMENTAÇÃO PARA APIS 25
3.1 Estrutura Geral 25
3.2 Tecnologias Utilizadas 25
3.3 Integração do Código-Fonte 26
3.3.1 Códigos HTML 26
3.3.2 Códigos *JavaScript* 28
3.3.3 Código utilizando a API Gemini 32

3.4 Funcionamento da Página *Web* 36

4 MÉTODO E ANÁLISE DOS RESULTADOS 38
4.1 Método para a Avaliação 38
4.2 Avaliação dos Resultados 39
4.2.1 Análise dos Resultados para a API Swing linguagem Java 39
4.2.2 Análise dos Resultados para a API OpenCV linguagem Python 45
4.2.3 Análise dos Resultados para a JavaIO linguagem Java 50
4.2.4 Análise dos Resultados para a API Requests linguagem Python 55
4.2.5 Média das LLMs por Linguagem 60
4.2.6 Média das LLMs conforme a quantidade de métodos das classes 61

5 CONCLUSÃO 64
5.1 Conclusão 64
5.2 Trabalhos Futuros 65

REFERÊNCIAS 66

Capítulo 1

Introdução

APIs oferecem poderosos mecanismos de abstração, permitindo que funcionalidades complexas sejam acessadas por programas clientes. No entanto, compreender como utilizá-la nem sempre é uma tarefa simples. Embora a documentação oficial da API possa ser útil, ela muitas vezes não é suficiente por si só. Plataformas *online* como *Stack Overflow* e *Github* surgiram para tentar preencher a lacuna entre a documentação tradicional das APIs e recursos com foco em exemplos práticos (SUBRAMANIAN; INOZEMTSEVA; HOLMES, 2014).

Embora existam iniciativas que utilizam dados para melhorar a documentação de APIs, como a abordagem proposta por Souza, Campos e Maia (2014), que identifica temas relacionados a API no *Stack Overflow* e cria um "livro de receitas" com base nessas informações, algumas limitações permanecem. A metodologia proposta não leva em consideração a complexidade dos exemplos apresentados, o que pode resultar em conteúdos difíceis de seguir para usuários menos experientes.

O trabalho apresentado por Rocha e Maia (2016) teve como critério resolver essas limitações, desenvolvendo e analisando metodologias diferentes para filtrar *posts* do *Stack Overflow* e organizá-los conforme o nível de complexidade de compreensão. Assim, um desenvolvedor com pouca experiência em uma determinada API poderia iniciar com exemplos mais simples e, após entendê-los, avançar para conteúdos mais difíceis. Porém o estudo não abrange uma avaliação em larga escala, levando em consideração uma diversidade maior de APIs e de avaliadores.

Com o avanço das LLMs houve uma transformação significativa na programação. Pré-treinados em grandes volumes de dados de código, conseguem desenvolver uma compreensão aprofundada do tema, o que os torna altamente eficazes em várias atividades relacionadas ao desenvolvimento (LIN et al., 2024). As LLMs despertaram o interesse da comunidade de pesquisa em IA, surpreendendo com suas capacidades linguísticas além das expectativas para sistemas de conclusão de texto (KAMBHAMPATI et al., 2024).

Sendo assim, com os modelos de LLMs, como por exemplo o Transformador pré-treinado generativo (GPT) e o Gemini, surgiram novas possibilidades para criar docu-

mentações mais claras e detalhadas, capazes de tornar o conteúdo mais acessível e fácil de entender. Esses modelos exibem capacidades notáveis, como a geração de textos coerentes, tradução automática e resposta a perguntas complexas. Dessa forma, explorar o uso de LLMs para a geração de documentações adaptadas e didáticas pode representar um crescimento significativo, aprimorando a qualidade das documentações disponíveis.

1.1 Motivação

Utilizar API de maneira correta pode ser um desafio sem um entendimento claro de sua finalidade. Para facilitar seu uso, a maioria das APIs disponibiliza documentações que explicam seus parâmetros de entrada, valores de saída e objetivos. No entanto, algumas dessas documentações são ambíguas, o que pode levar a erros de uso, pois geralmente estão escritas em linguagem natural, voltada para leitura humana. Por isso, os desenvolvedores frequentemente procuram informações adicionais mais formais, como exemplos de código, (KIM et al., 2009).

Uma documentação bem elaborada facilita a inovação tecnológica, amplia a acessibilidade e melhora a eficiência do desenvolvimento de *software*. Ela fornece orientações claras, evita mal-entendidos e reduz o tempo necessário para integrar novas funcionalidades, minimizando a ocorrência de erros. Além disso, a inclusão de exemplos de uso práticos auxilia na compreensão das funcionalidades das APIs, permitindo que os desenvolvedores se concentrem em entregar produtos de maior qualidade.

Diante disso, esta pesquisa se justifica pela necessidade de melhorar a qualidade da documentação de APIs, com o objetivo de proporcionar materiais mais didáticos, atualizados e com exemplos práticos que favoreçam o aprendizado e a aplicação correta. Ao explorar a geração de documentação por LLMs e comparar com abordagens tradicionais, o estudo visa contribuir para o desenvolvimento de soluções que aumentem a produtividade dos desenvolvedores e promovam um ambiente de desenvolvimento de *software* mais eficiente e acessível.

1.2 Objetivos

Nesta sessão será apresentado o objetivo principal do estudo, seguido de seus objetivos específicos.

1.2.1 Objetivo Principal

O objetivo deste trabalho é realizar uma avaliação quantitativa de documentações para APIs geradas automaticamente por modelos de LLMs, comparando-as com suas respectivas documentações oficiais e analisar a viabilidade dessa geração automática.

1.2.2 Objetivos Específicos

- ❏ Comparar a documentação gerada por LLMs com as documentações oficiais de APIs.
- ❏ Construir um código *Python* no qual vai utilizar LLM para gerar a documentação automaticamente.
- ❏ Construir uma página *web* permitindo o usuário entrar com a linguagem de programação e API desejada.
- ❏ Analisar o percentual de cobertura de métodos das classes presentes nas documentações das APIs, geradas pelas LLMs, em comparação com as documentações oficiais.

1.3 Organização da Monografia

Este trabalho está organizado com os seguintes capítulos: o Capítulo 2, explora os conceitos necessários para a compreensão do estudo, sendo eles a documentação oficial de APIs, LLMs, API Gemini e Processamento de Linguagem Natural (PLN), acompanhado de estudos acadêmicos que abordam questões similares. O Capítulo 3 detalha a abordagem utilizada para a geração da documentação para APIs, incluindo o código desenvolvido e as telas de funcionamento da página *web*. No Capítulo 4 é apresentado o método utilizado para a avaliação, e a análise dos resultados, explicando cada etapa do processo e destacando os dados obtidos. E por fim, o Capítulo 5 traz a conclusão do estudo, baseada nas análises do capítulo anterior, e em seguida aponta os aspectos que podem ser explorados em trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo, serão apresentados os conceitos teóricos que embasam o desenvolvimento deste trabalho. Abordaremos as tecnologias utilizadas e os trabalhos relacionados à reutilização de *software* e documentação de API, fundamentais para entender os desafios e soluções propostas.

2.1 Conceitos Fundamentais

Nesta seção, serão abordados os conceitos fundamentais para proporcionar uma compreensão mais aprofundada do trabalho, incluindo descrições detalhadas e explicações que contextualizam cada um deles dentro do tema estudado.

2.1.1 Documentação oficial de APIs

A documentação de API, conforme citada por (FAN et al., 2021), exerce um papel essencial no desenvolvimento e na reutilização de *software*, sendo importante tanto para os desenvolvedores quanto para estudantes ou usuários que procuram esse tipo de conteúdo. Ela tem a finalidade de auxiliar os desenvolvedores a compreender os recursos disponíveis, e reutilizar os códigos com eficiência, permitindo que eles consigam realizar sua aplicação de maneira simples. Contudo, a documentação nem sempre atende a todas as necessidades dos programadores. Há casos onde a ausência de informações relevantes leva os desenvolvedores a recorrer a outras fontes, como fóruns de dúvidas, ou comunidades técnicas, para buscar soluções.

2.1.2 Processamento de Linguagem Natural - PLN

O PLN, de acordo com Caseli e Nunes (2024), é um campo de pesquisa dedicado a investigar e desenvolver métodos e sistemas para o processamento computacional da linguagem humana. O termo “Natural” se refere às línguas usadas pelos seres humanos,

diferenciando-as de outras linguagens, como as matemáticas, visuais, gestuais e de programação. Na Ciência da Computação, o PLN está associado à IA e intimamente relacionado à Linguística Computacional.

Segundo Maurer e Zamberlan (2024), o PLN tem desempenhado um grande papel na evolução dos sistemas automatizados, tornando-os cada vez mais inteligentes e capazes de lidar com a complexidade e a ambiguidade da linguagem humana. Assim, essa área combina técnicas, algoritmos, e ferramentas capazes de compreender, interpretar e gerar linguagem de forma semelhante ao modo como os seres humanos se comunicam.

2.1.3 Large Language Models - LLMs

A *Large Language Model* (LLM), de acordo com Rachmat e Kesuma (2024), atualmente vem sendo um dos tipos de IA que mais vem se destacando, sendo a evolução de pesquisas anteriores envolvendo PLN.

Conforme o estudo citado, as LLMs são sistemas capazes de compreender e gerar textos em linguagem natural, por serem treinados com grandes volumes de dados e utilizarem técnicas para identificar padrões e estruturas linguísticas. Assim, permitindo que os modelos executem diferentes tarefas relacionadas ao uso da linguagem, como geração de texto, tradução automática, resumo de documento, respostas automáticas, entre outras.

2.1.4 API Gemini

De acordo com (APRILIA et al., 2024), a API Gemini foi desenvolvida com o objetivo de facilitar na aplicação de recursos de IA em aplicações *web* e móvel. A plataforma disponibiliza diferentes funcionalidades, como análise de dados, sistemas de recomendação e processamento de linguagem natural (PLN). Dessa forma, ela permite que os desenvolvedores consigam utilizá-la em seus projetos de maneira prática, sem a necessidade de construir e treinar modelos do zero.

2.1.5 HTML

Segundo Krause (2016), o *HyperText Markup Language* (HTML) é uma linguagem de marcação responsável por estruturar o conteúdo de uma página *web*. A versão atual, HTML 5, ampliou as capacidades da linguagem, oferecendo suporte a diferentes funcionalidades além de uma simples marcação. Mesmo diante dessas melhorias, o HTML por si só não atende às exigências atuais do desenvolvimento *web*, por isso surgiram *frameworks* e sistemas de *templates* que buscam completar suas funcionalidades e superar suas limitações.

2.1.6 CSS

De acordo com (KRAUSE, 2016), o *Cascading Style Sheets* (CSS) é uma linguagem de formatação e *layout* que permite aplicar estilos a linguagens de marcação, como o HTML. Enquanto o documento HTML contém apenas as informações semânticas, o CSS é responsável por definir a apresentação visual e tipográfica desses elementos. Ele também possibilita a definição de estilos específicos para diferentes modos de exibição, como tela de monitores, projeções e impressões, oferecendo maior flexibilidade e controle sobre a apresentação do conteúdo.

2.1.7 Python

De acordo com Srinath (2017), *Python* é uma linguagem de programação de alto nível, dinâmica e orientada a objetos, amplamente utilizada em diversas áreas da computação. Criada com foco na simplicidade e facilidade de aprendizagem, se tornou umas das linguagens mais acessíveis para iniciantes, além de ser bastante utilizada por desenvolvedores mais experientes. Nos últimos anos a linguagem ganhou popularidade, chegando a substituir a linguagem Java como linguagem introdutória mais utilizada em ambientes acadêmicos e cursos de programação. Por ser dinâmica, ela proporciona mais flexibilidade no desenvolvimento, e permite maior tolerância a erros, onde os programas podem ser executados mesmo na presença de inconsistências em partes específicas do código.

2.1.8 JavaScript

Como afirmado no estudo Mikkonen e Taivalsaari (2007), o *JavaScript* é uma linguagem de programação orientada a objetos baseada em protótipos, conhecida por seu papel como linguagem de *script* na *web*. Por ser dinâmica e interativa, não precisa que haja declaração prévia dos tipos das variáveis, e possibilita alterações de código em tempo de execução, o que garante uma maior flexibilidade no desenvolvimento. Durante muito tempo, a linguagem ficou conhecida por ser restrita a *scripts* simples e animações de páginas *web*, porém atualmente com o crescimento na complexidade dos aplicativos desenvolvidos e na criação de bibliotecas mais avançadas, a percepção da linguagem vem ganhando mais valor.

2.1.9 Flask

De acordo com o estudo Alemu (2014), o *Flask* é um *microframework* para desenvolvimento *web* em *Python*, que permite a criação de aplicações de forma simples e eficiente. É uma ferramenta versátil, utilizada em projetos de diferentes escalas, desde pequenas aplicações à sistemas mais complexos, como as redes sociais. Ele possui três características principais: simplicidade, permitindo o desenvolvedor aprender durante o processo

de implementação, com o auxílio de uma documentação de início rápido para facilitar a configuração e as funcionalidades básicas; possui código aberto, o que o torna mais acessível; e possui uma documentação abrangente que é constantemente atualizada, oferecendo tutoriais detalhados e suporte nas versões mais recentes.

2.2 Trabalhos Correlatos

Nesta seção, serão abordados os conceitos fundamentais para proporcionar uma compreensão mais aprofundada do trabalho, incluindo descrições detalhadas e explicações que contextualizam cada um deles dentro do tema estudado.

No artigo *Improving API Documentation Using API Usage Information*, (STYLOS et al., 2009), é proposto o uso do sistema Jadeite para aprimorar a documentação de APIs. Ele integra dados de uso efetivo, exibindo as classes comuns de forma mais proeminente na documentação para facilitar a navegação e a descoberta das funcionalidades, e permitindo a adição de "placeholders" para métodos inesperados que não estão presentes na documentação oficial. No entanto, esse sistema apresenta limitações, como a dependência de dados de uso para eficácia, onde APIs menos populares, com poucos exemplos disponíveis, podem não se beneficiar totalmente do sistema, o que levanta questões sobre sua aplicabilidade em diferentes contextos. Sendo assim, uma das propostas deste trabalho de conclusão de curso é realizar uma avaliação da geração da documentação automática através de três tipos de LLMs - ChatGPT, Gemini e DeepSeek - além de mostrar a aplicação *web*, que não irá possuir limitação com relação a API e linguagem de programação escolhidas.

No artigo *Adding Examples into Java Documents*, (KIM et al., 2009), é proposto uma abordagem inovadora para aumentar a quantidade de exemplos de código em documentos de APIs, fazendo sua extração automática a partir de repositórios públicos, resultando na criação dos chamados *eXoaDocs*. Essa estratégia visa enriquecer a documentação, tornando-a mais útil e acessível para os desenvolvedores que utilizam API. Porém, há desafios significativos na garantia da relevância e qualidade dos exemplos gerados, por não haver a comparação com a documentação oficia, além de possuir limitação da API e da linguagem de programação na documentação gerada. Em vista disso, este TCC tem como uma de suas finalidades avaliar a geração da documentação automática através de três LLMs, comparando estas com suas documentações oficiais, e também apresentar uma *interface web* que permite a escolha da API e linguagem de programação para a geração da documentação automática.

No artigo *On the Extraction of Cookbooks for APIs from the Crowd Knowledge*, Souza, Campos e Maia (2014), é proposto uma abordagem para organizar o conhecimento coletivo de desenvolvedores disponível no *Stack Overflow*, visando criar "cookbooks" (livros de receitas) para APIs. A proposta desse artigo tinha como foco reunir e estruturar infor-

mações valiosas em um formato acessível e prático, facilitando a utilização de APIs por meio de exemplos e orientações claras. Contudo, essa abordagem apresenta algumas limitações, como garantir a coerência entre os capítulos, já que as informações são extraídas de diversas fontes o que pode variar em estilo e profundidade. Além disso, a qualidade dos exemplos pode ser inconsistente, refletindo diferentes níveis de experiência dos desenvolvedores, e há o risco de incluir conteúdo desatualizado ou incorreto. Então, uma das propostas do presente trabalho é utilizar três tipos de LLMs para analisar a geração de documentação automática, comparando-as com suas respectivas docuemnetações oficiais, e garantir também uma *interface web* que onde não há limitações na escolha da API e da linguagem de programação desejada.

No artigo *Automated API Documentation with Tutorials Generated From Stack Overflow*, Rocha e Maia (2016), é apontado uma abordagem para gerar tutoriais automáticos de APIs utilizando informações extraídas do *Stack Overflow*. O objetivo é melhorar a documentação de APIs, tornando-a mais compreensível e acessível para desenvolvedores de diferentes níveis de experiência, através da integração de exemplos práticos e explicações detalhadas. Porém, a abordagem apresenta algumas limitações, com foco apenas na API *Swing*, sem o auxilio das LLMs para a geração da documentação automática. Logo, este TCC busca analisar a geração da documentação automática feita através de três LLMs, além de permitir que através da *interface web* seja possível a seleção de diferentes APIs e linguagens de programação.

No artigo *Enriching API Documentation by Relevant API Methods Recommendation based on Version History*, (ARIMATSU et al., 2018), é proposto uma abordagem para enriquecer a documentação de APIs ao recomendar métodos relevantes com base no histórico de versões do *software*. O método desenvolvido analisa *commits* de adição e modificação para identificar padrões de co-mudança entre métodos, permitindo gerar automaticamente grupo de métodos relacionados e inseridos na documentação *Javadoc*. Contudo essa técnica apresenta algumas limitações, como a dependência da qualidade do histórico de versões e a ausência da análise prevista sobre os métodos recomendados. O presente estudo busca analisar a geração de documentação automática através de três LLMs, e compará-las com suas documentações oficiais. Além de permitir que haja a geração da documentação automática através de uma *interface web* sem limite na escolha da API e linguagem de programação desejada.

O artigo *Automated API-Usage Update for Android Apps*, Fazzini, Xin e Orso (2019), propõe o *AppEvolve* para que os desenvolvedores possam adaptar seus aplicativos, uma técnica automatizada para atualizações de código-fonte em resposta a mudanças da API. Este método identifica usos obsoletos de API em um código de destino e busca exemplos de atualização em outras bases de código, analisando padrões de modificação e gerando *patches* genéricos para aplicar as mudanças permitidas. No entanto essa técnica possui limitações, como a dependência de um conjunto de exemplos representativos e a necessi-

dade de ajustes manuais em casos mais complexos. O presente trabalho busca garantir que não haja limitação na escolha da API e linguagem de programação da documentação automática a ser gerada. Além de realizar uma análise que compara a geração automática de três LLMs com a documentação oficial da API.

No artigo *gDoc: Automatic Generation of Structured API Documentation*, Wang, Tian e He (2023), é proposto o "gDoc" para manter a documentação atualizada com exemplos de uso para APIs em evolução, utilizando um modelo "Seq2Seq", baseado em aprendizado profundo. Seu objetivo é reduzir o tempo e o esforço necessários para a criação de documentação de APIs, garantindo consistência e qualidade na geração automática de descrições de parâmetros e exemplos de uso. Porém apresenta algumas limitações em relação a APIs menos populares, e na precisão da documentação gerada. Este trabalho de conclusão de curso busca explorar o uso das LLMs para gerar as documentações e compará-las com suas respectivas documentações oficiais.

No artigo *Automatic Code Documentation with Syntax Trees and GPT*, Procko e Collins (2023), é proposto um sistema *Automatic Code Documentation with Compilers* (ACDC), que utiliza árvores de sintaxe abstratas (ASTs) e GPT-3.5 para gerar documentação automática de código na linguagem C#/NET. Seu objetivo é integrar a documentação de fluxo CI/CD, reduzindo a necessidade de intervenção manual por parte dos desenvolvedores e garantindo que a documentação permaneça atualizada. Contudo este sistema apresenta limitações na adaptação a outras linguagens de programação e na documentação extensa. Este TCC tem como uma de suas finalidades explorar a geraração das documentações automáticas através de três modelos de LLMs comparando com a docuemntação oficial das APIs, além de permitir que, por meio de uma *interface web*, possa ser solicitado qualquer tipo de API e linguagem de programação para a geração da documentação.

No artigo *Enriching Application Programming Interface (API) Documentation Using a Large Language Model (LLM) Architecture to Improve LLM API Interpretability*, (BASHAN et al., 2024), é proposto um sistema automatizado para melhorar a documentação das APIs. Seu objetivo é melhorar a qualidade e a atualização das documentações utilizando LLMs para que as alterações sejam analisadas automaticamente. Porém este estudo apresenta limitações com relação a interpretação de APIs complexas, e com a adaptação de diferentes padrões de documentação. Então este trabalho de conclusão de curso busca explorar e analisar o uso de LLMs para a geração da documentação automática, comparando-a com sua docuemntação oficial. Além de permitir a escolha da API e linguagem de programação através de uma *interface web*.

O artigo *Generation of API Documentation using Large Language Models Towards Self-explaining APIs*, (JORELLE, 2024), estuda as LLMs para gerar documentação de APIs e auxiliar os desenvolvedores iniciantes em sua aprendizagem. Neste estudo um aplicativo *Web* foi desenvolvido para ser avaliado por estudantes. Como resultado, foi

observado que as explicações geradas pelas LLMs são úteis para iniciantes, mas sua utilidade diminui conforme aumenta o nível de experiência do programador. No entanto, o estudo apresenta algumas limitações, como o impacto das novidades das LLMs e o foco em uma única API. Logo, este trabalho de conclusão de curso busca ampliar a variedade de APIs e de LLMs para análise.

O artigo *Automated API Docs Generator using Generative AI*, (DHYANI et al., 2024), propõe um gerador automatizado de documentação de API utilizando IA generativa para melhorar os resultados, a velocidade e a escalabilidade da documentação. O modelo adotado foi treinado com dados extraídos de *web scraping* de documentações de grandes empresas de tecnologia, aprimorado posteriormente com ajustes baseados em modelos GPT. Seu objetivo é melhorar a manutenção da documentação de APIs, tornando-as mais acessíveis. Porém essa abordagem apresenta algumas limitações, sendo uma delas o desafio da adaptação a APIs menos documentadas. Este trabalho de conclusão de curso tem como um de seus objetivos explorar três modelos de LLMs para a geração da documentação automática, comparando-as com suas documentações oficiais.

O artigo *Leveraging Large Language Models for the Automated Documentation of Hardware Designs*, (FERNANDO et al., 2024), propõe um sistema que gera documentação legível através de modelos formais de hardware, onde usam LLMs para a documentação automatizada. Essa abordagem utiliza uma estrutura baseada em *Model Driven Architecture* (MDA), onde as especificações são transformadas em segurança textuais por meio da engenharia de *prompts* aplicada as LLMs. Contudo, esse método possui suas limitações, onde futuramente pretendem melhorar a qualidade e a consistência da documentação gerada. Logo, a presente pesquisa tem como um de seu objetivos explorar o uso das LLMs para gerar documentações automáticas e compará-las com suas documentações oficiais.

Tabela 1 – Análise Comparativa dos Trabalhos Correlatos

Trabalhos Relacionados	**Documentação automática**	**LLMs**	**Avaliação Qualitativa**	**Avaliação Quantitativa**	**Comparação com a Documentação Oficial**	**Sem Limitação de APIs**	**Sem Limitação de Linguagens**	**APIs Avaliadas**
(STYLOS et al., 2009)	✖	✖	✓	✖	✓	✖	✖	Javadoc
(KIM et al., 2009)	✓	✖	✓	✓	✖	✖	✖	JDK5
Souza, Campos e Maia (2014)	✓	✖	✓	✓	✖	✖	✖	SWT, STL e LINQ
Rocha e Maia (2016)	✓	✖	✓	✓	✓	✖	✖	Swing
(ARIMATSU et al., 2018)	✓	✖	✓	✖	✖	✖	✖	Javadoc
Fazzini, Xin e Orso (2019)	✓	✖	✓	✓	✖	✖	✖	Android
Wang, Tian e He (2023)	✓	ChatGPT	✓	✓	✖	✓	✓	OpenAPI
Procko e Collins (2023)	✓	ChatGPT	✓	✖	✖	✖	✖	OpenAPI
(BASHAN et al., 2024)	✓	✓	✓	✖	✖	✖	✖	OpenAPI
(JORELLE, 2024)	✓	ChatGPT	✖	✓	✓	✖	✓	OpenAPI
(DHYANI et al., 2024)	✓	ChatGPT	✓	✖	✓	✖	✓	ChatGPT
(FERNANDO et al., 2024)	✓	✓	✓	✓	✖	✖	✖	OpenAI
Presente Trabalho de Conclusão de Curso	✓	ChatGPT, Gemini e DeepSeek	✖	✓	✓	✓	✓	Swing, OpenCV, JavaIO, Requests

Na Tabela 1 é feita uma análise comparativa entre os trabalhos correlatos e o presente

estudo. Nota-se que grande parte das pesquisas anteriores abordaram a documentação automática, onde sua maior parte fica restrita a APIs ou linguagens de programação específicas. Alguns trabalhos mais recentes exploraram o uso de LLMs, aqueles que especificaram o modelo escolhido destacaram a abordagem através do *ChatGPT*. E observa-se que nem todos optaram por comparar os resultados conseguidos com a documentação oficial da API para verificar a integridade.

O presente trabalho busca a geração da documentação automática comparando os resultados obtidos através de diferentes tipos de LLMs (*ChatGPT, Gemini e DeepSeek*) e APIs (*Swing, JavaIO, OpenCV, Requests*), comparando-os com a documentação oficial da API, além de permitir que não haja limitações nas escolhas das APIs e das linguagens de programação, durante o uso da abordagem proposta.

Capítulo 3

Abordagem para Geração de Documentação para APIs

Neste capítulo, será apresentado o passo a passo da implementação da página *web* desenvolvida, que utiliza uma LLM para gerar e retornar a documentação da API de acordo com os parâmetros informados pelo usuário.

3.1 Estrutura Geral

A solução proposta foi desenvolvida com o objetivo de facilitar a geração de documentação para APIs por meio da utilização de LLMs. A arquitetura do sistema é composta por dois principais componentes: a *interface web* (*frontend*) e o servidor de aplicação (*backend*).

A *interface web* foi desenvolvida em HTML, CSS e *JavaScript*, permitindo que o usuário insira informações relacionadas à API, como os parâmetros ou linguagem desejada para a documentação. Essas informações são enviadas para o *backend*, que foi implementado utilizando o *framework Flask*, em *Python*.

O *backend* tem a função de receber os dados do usuário, estruturar o *prompt* de entrada para a LLM e realizar a requisição para a API da mesma (neste caso, o Google Gemini, por ser gratuito, permite a geração de uma chave de acesso à API, possibilitando a utilização de suas funcionalidades). Após obter a resposta com a documentação gerada, o *backend* trata o conteúdo retornado e o repassa para o *frontend*, onde é exibido ao usuário de forma organizada.

3.2 Tecnologias Utilizadas

A implementação contou com diversas tecnologias que atuam de forma integrada para permitir a coleta de dados, a comunicação com a LLM e a apresentação dos resultados ao usuário. A seguir serão descritas as principais ferramentas e linguagens utilizadas:

- HTML e CSS: utilizados para estruturar e estilizar a página *web*, permitindo a criação de um ambiente funcional para o usuário;
- *JavaScript*: responsável em lidar com os eventos da *interface*, como o envio dos dados do usuário para o *backend*;
- *Python*: linguagem de programação escolhida pela simplicidade e ampla extenção da biblioteca para integração da API;
- *Flask*: *framework web* que facilita a criação de rotas HTTP e o tratamento das requisições da *interface*;
- API Gemini: utilizada para a geração da documentação automática;
- *Visual Studio Code*: editor de código-fonte utilizado no desenvolvimento do projeto.

3.3 Integração do Código-Fonte

Nesta sessão serão apresentados os principais trechos do código que compõem a página *web* desenvolvida. Mostrando como ocorre a interação entre o usuário, *backend* e a LLM escolhida para gerar a documentação automatizada.

Serão apresentados o código HTML responsável pela interface que irá se apresentada ao usuário, o código de *javascript* que irá lidar com a interação e o envio de dados, e o *backend* em *python* utilizando a biblioteca *Flask* onde serão processados os dados recebidos e requisitados para a API do modelo Gemini.

3.3.1 Códigos HTML

```
1     <header>
2     <h1>API Gemini</h1>
3     </header>
4     <main>
5     <div class="container">
6     <label>Informe a API desejada:</label>
7     <input id = "api" type="text">
8     <label>Informe a Linguagem desejada:</label>
9     <input id = "linguagem" type="text">
10
11    <button onclick="enviardados()">Enviar</button>
12    <div id="resultados" style="margin-top: 20px;"></div>
13    </div>
14    </main>
```

Exemplo 3.1 – Código HTML da Página Inicial.

No código do Exemplo 3.1 é apresentado o código-fonte da página inicial, onde dentro de uma única *div* são criados os dois *inputs* necessários para que o usuário preencha com os dados de sua escolha (API e Linguagem de Programação), e um *button*, que após receber o *click*, realizará a ação de enviar os dados para o *script* apresentado no Exemplo 3.4.

```
1     <header>
2     <h1>API Gemini</h1>
3     <nav>
4         <a href="{{ url_for('voltar') }}">Voltar a Pagina Principal</a>
5     </nav>
6     </header>
7
8     <main>
9     <div id="menu">
10        <h3>Classes</h3>
11        <ul id="menu-classes"></ul>
12    </div>
13    <div class="container" id="conteudo">
14        <h2>Selecione uma Classe</h2>
15    </div>
16    </main>
```

Exemplo 3.2 – Código HTML da Página para Escolha da Classe.

O código no Exemplo 3.2 apresenta uma estrutura básica de aplicação. No elemento '*<header>*' contém o cabeçalho, onde são incluídos o título da página e em seguida, na linha 4, um link com a função "url_for('voltar')" própria do *Flask* para redirecionar caso desejado para a rota definida como 'voltar' no *backend*. Enquanto no elemento '*<main>*' são integrados um menu lateral de navegação que listará as classes da API, e ao lado uma '* *' que é uma área que será utilizada com um propósito no Exemplo 3.3, enquanto isso será exibida apenas a mensagem "Selecione uma Classe".

```
1     <header>
2     <h1>API Gemini</h1>
3     <nav>
4         <a href="{{ url_for('voltar') }}">Voltar a Pagina Principal</a>
5     </nav>
6     </header>
7
8     <main>
9     <div class="container" id="menu">
10        <h3>Classes</h3>
11        <ul id="menu-classes"></ul>
12    </div>
13    <div class="container" id="conteudo">
14        <h2>Metodos da Classe: <span id="nome-classe"></span></h2>
15        <div id="metodos-lista"></div>
16    </div>
```

```
17      </main>
```

Exemplo 3.3 – Código HTML da Página para Apresentar os Métodos da Classe.

O código no Exemplo 3.3 é uma *interface* composta por um cabeçalho fixo e uma área principal dividida em duas partes: um menu lateral e uma área central. No elemento '*<header>*' apresenta o título da página e a função "url_for('voltar')" com o mesmo propósito mostrado no Exemplo 3.2. No elemento '*<main>*' possui duas '*divs*', uma com o propósito de exibir uma lista com as classes da API, e a outra com o intuito de mostrar os métodos da classe selecionada no menu lateral.

3.3.2 Códigos *JavaScript*

```
1   async function enviardados() {
2       const api = document.getElementById('api').value;
3       const linguagem = document.getElementById('linguagem').value;
4       try {
5           const response = await fetch('/enviardados', {
6               method: 'POST',
7               headers: {
8                   'Content-Type': 'application/json',
9               },
10              body: JSON.stringify({ api: api, linguagem: linguagem })
11          });
12
13          if (response.ok) {
14              window.location.href = `/resultados?api=${
    encodeURIComponent(api)}`;
15          } else {
16              const errorText = await response.text();
17              console.error('Erro ao enviar dados:', errorText);
18              alert('Erro ao carregar os dados.');
19          }
20      } catch (error) {
21          console.error('Erro na requisicao:', error);
22          alert('Erro ao realizar a requisicao.');
23      }
24  }
```

Exemplo 3.4 – Código JavaScript da Página Inicial.

O Exemplo 3.4 apresenta o código-fonte do *script* da página inicial mostrada na Figura 1. No código é criado uma função chamada "enviar dados", que será responsável por enviar os dados para o processamento no *backend*. Nela são armazenados em variáveis os dados digitados pelo o usuário, neste caso a "API" e a linguagem de programação escolhida.

```
1   document.addEventListener('DOMContentLoaded', async function() {
```

```
2     const urlParams = new URLSearchParams(window.location.search);
3     const apiName = urlParams.get('api');
4
5     if (apiName) {
6         document.querySelector('header h1').textContent = `API ${apiName
    }`;
7     }
8
9     const response = await fetch('/obter_resultados');
10    if (response.ok) {
11        const data = await response.json();
12        const classes = data.resultados;
13
14        const menuClasses = document.getElementById('menu-classes');
15
16        classes.forEach(classe => {
17            const li = document.createElement('li');
18            const link = document.createElement('a');
19            link.textContent = classe;
20            link.href = `/metodos?classe=${encodeURIComponent(classe)}&
    api=${encodeURIComponent(apiName)}`;
21            li.appendChild(link);
22            menuClasses.appendChild(li);
23        });
24    } else {
25        console.error('Erro ao carregar resultados.');
26    }
27    });
```

Exemplo 3.5 – Código JavaScript da Página para Escolha de Classe.

O Exemplo 3.5 é responsável por carregar as classes de uma determinada API e preencher o menu lateral com os *links* correspondentes. Ele é executado assim que a página for carregada utilizando "*DOMContentLoaded*". Na linha 4 é extraído o nome da API e armazenado na variável "apiName", isso será utilizado para atualizar o título da página com o nome da API escolhida no '*if*' da linha 5. Na linha 9 é feita uma requisição ao *endpoint* '/obter_resultados'. Em seguida, na linha 10, se a requisição tiver sido bem sucedida será retornado um JSON com a lista das classes extraídas da API e armazenada na variável 'classes'. Na linha 16, para cada classe recebida, o *script* cria um item de lista(<li>) e um *link*(<a>), definindo o texto do *link* com o nome da classe e redirecionando para a página '/metodos', onde são passados juntos o nome da classe e da API. Caso haja falha na requisição a mensagem "Erro ao carregar resultados" é registrada no *console*.

```
1
2     document.addEventListener('DOMContentLoaded', async function() {
3         const urlParams = new URLSearchParams(window.location.search);
4         const apiName = urlParams.get('api');
```

```
5         const classe = urlParams.get('classe');
6
7         if (apiName) {
8             document.querySelector('header h1').textContent = `API ${
    apiName}`;
9         }
10        if (classe) {
11            document.getElementById('nome-classe').textContent = classe;
12        }
13
14        const response = await fetch('/obter_resultados');
15        if (response.ok) {
16            const data = await response.json();
17            const classes = data.resultados;
18
19            const menuClasses = document.getElementById('menu-classes');
20
21            classes.forEach(classeItem => {
22                const li = document.createElement('li');
23                const link = document.createElement('a');
24                link.textContent = classeItem;
25                link.href = `/metodos?classe=${encodeURIComponent(
    classeItem)}&api=${encodeURIComponent(apiName)}`;
26                li.appendChild(link);
27                menuClasses.appendChild(li);
28            });
29        } else {
30            console.error('Erro ao carregar resultados.');
31        }
32    });
33
34    function obterParametroClasse() {
35        const urlParams = new URLSearchParams(window.location.search);
36        return urlParams.get('classe');
37    }
38
39    async function carregarMetodos() {
40        const classe = obterParametroClasse();
41        document.getElementById('nome-classe').textContent = classe;
42
43        try {
44            const response = await fetch(`/obter_metodos/${
    encodeURIComponent(classe)}`);
45            if (response.ok) {
46                const data = await response.json();
47                const metodos = data.metodos;
48
```

```
49                const metodosDiv = document.getElementById('metodos-
    lista');
50                metodosDiv.innerHTML = '';
51
52                metodos.forEach((metodo, index) => {
53                    const divMetodo = document.createElement('div');
54                    divMetodo.className = 'metodo-item';
55
56                    divMetodo.style.backgroundColor = index % 2 === 0 ?
    '#f0f0f0' : '#d9d9d9';
57
58                    divMetodo.innerHTML = `<pre><code class="codigo">${
    metodo}</code></pre>`;
59
60                    metodosDiv.appendChild(divMetodo);
61                });
62            }
63            else {
64                console.error(`Erro ao carregar metodos para a classe ${
    classe}`);
65            }
66        } catch (error) {
67            console.error('Erro na requisicao:', error);
68        }
69    }
70    document.addEventListener('DOMContentLoaded', carregarMetodos);
```

Exemplo 3.6 – Código JavaScript da Página para Apresentar os Métodos da Classe.

O trecho de código no Exemplo 3.6 é responsável por montar a *interface* da página de métodos. Ele utiliza os parâmetros da URL para identificar a API e a classe selecionadas, atualiza o título da página e preenche o menu lateral com as classes, e a área central com a lista de métodos.

Assim como o Exemplo 3.5, é obtido o valor da API e feita a atualização no título da página, porém tem como adicional a classe selecionada para atualizar o título de carregamento dos métodos. Nele é feito novamente a requisição ao *backend* preencher o menu lateral.

A linha 34 possui a função "obterParametroClasse" para auxiliar no retorno do nome da classe baseado na URL da página atual. Na linha 39 a função "carregarMetodos" realiza a requisição para obter os métodos da classe. O conteúdo anterior é apagado com "*innerHTML*=' ' " para evitar duplicação ao recarregar. O *forEach* na linha 52 tem como finalidade exibir os métodos com cores alternadas, dentro de um elemento '*<pre><code>*' para que o conteúdo seja mais apresentável ao usuário. A linha 70 a função "carregarMetodos" é chamada automaticamente ao final do carregamento da página, garantindo que os métodos da classe estejam disponíveis para visualização imediatamente.

3.3.3 Código utilizando a API Gemini

```
1   genai.configure(api_key='API_KEY')
2
3   generation_config = {
4     "temperature": 1,
5     "top_p": 0.95,
6     "top_k": 64,
7     "max_output_tokens": 8192,
8     "response_mime_type": "text/plain",
9   }
10
11  model = genai.GenerativeModel(
12    model_name="gemini-1.5-flash",
13    generation_config=generation_config,
14  )
15
16  chat_session = model.start_chat(
17    history=[
18    ]
19  )
20
21  def save_to_file(text, filename="resultados.txt"):
22      with open(filename, "a", encoding="utf-8") as file:
23          file.write(text + "\n")
```

Exemplo 3.7 – Código Python utilizando Flask para criar a rota inicial.

O trecho do código mostrado no Exemplo 3.7 realiza a configuração inicial da API Gemini. Na linha 1 é configurado a chave de autenticação da API, necessária para autorizar as requisições ao serviço da Google. Na linha 3 define os parâmetros da geração de conteúdo do modelo. A linha 11 cria uma instância do modelo "gemini-1.5-flash" com as configurações definidas. A linha 18 inicia uma nova sessão de *chat* com o modelo, o histórico começa vazio, mas pode ser atualizado posteriormente para manter o conteúdo de conversas anteriores com o modelo. Essas configurações apresentadas foram retiradas diretamente da documentação da API Gemini. Já a linha 23 implementa uma função para salvar os resultados em um arquivo local "resultados.txt".

```
1   app = Flask(__name__)
2   @app.route('/')
3   def home():
4   return render_template('principal.html')
```

Exemplo 3.8 – Código Python utilizando Flask para criar a rota inicial.

O trecho no Exemplo 3.8 é responsável por iniciar a aplicação *Flask*, definindo a rota principal responsável por carregar a interface inicial da página *web*.

```
1   @app.route('/enviardados', methods=['POST'])
2   def enviardados():
3   global resultados
4   data = request.get_json()
5   if not data or 'api' not in data or 'linguagem' not in data:
6       return jsonify({"error": "Dados faltando na requisicao."}), 400
7
8   api = data.get('api')
9   linguagem = data.get('linguagem')
10
11  prompt = f"Liste apenas os nomes de todas as classes da API {api}
    com a linguagem {linguagem}."
12
13  try:
14      response = chat_session.send_message(prompt)
15      nova_resposta = response.text.replace("*", "")
16      resposta_texto = nova_resposta.strip()
17
18      if ":" in resposta_texto:
19          pos = resposta_texto.find(":") + 1
20          texto_apos_colon = resposta_texto[pos:].strip()
21          resposta_lista = texto_apos_colon.split('\n')
22      else:
23          resposta_lista = [resposta_texto]
24
25      resultados = resposta_lista
26
27      return '', 204
28
29  except Exception as e:
30      print(f"Erro ao gerar resposta: {str(e)}")
31      return jsonify({"error": f"Erro ao gerar resposta: {str(e)}"}),
    500
```

Exemplo 3.9 – Código Python utilizando Flask para criar a rota enviar dados.

O trecho no Exemplo 3.9 cria a rota "/enviardados", responsável por processar os dados recebidos da *interface web*, gerar a resposta utilizando o modelo da API Gemini, extrair as classes retornadas e armazená-las em memória para serem usadas posteriormente.

Primeiramente é definido uma rota "*POST*" para receber os dados enviados pela interface HTML. Na linha 5 verifica se os dados foram enviados corretamente, caso não tenha sido é retornado um erro.

Na linha 11 é montado o *prompt* com o seguinte comando: "Liste apenas os nomes de todas as classes da API { api} com a linguagem { linguagem}." onde 'api' e 'linguagem' fazem parte dos dados informados pelo usuário que foram recebidos da interface através

do uso de '*GET*'.

Em seguida é feito o tratamento de excessão, na linha 14 o *prompt* é enviado para a sessão da API Gemini para o processamento da resposta. As linhas 15 e 16 são responsáveis por remover símbolos, como o asterisco, e espaços em branco. Na linha 18 caso a resposta tenha o símbolo ":", é considerado que os dados úteis estão após esse caracter. Após isso a resposta é dividida em uma lista de *strings*, cada uma representando uma classe. Caso não haja o símbolo ":" considera a resposta como uma única *string*. Na linha 25 a lista contendo as classes é armazenada na variável global "resultados", e retorna um *status* 204 para indicar que a operação foi bem sucedida. E na linha 29, em caso de erro, é retornado uma mensagem de erro detalhada.

```
1     @app.route('/resultados')
2     def resultados_page():
3         return render_template('index.html')
```

Exemplo 3.10 – Código Python utilizando Flask para criar a rota resultados.

O código do Exemplo 3.10 é responsável por carregar a página onde as classes retornadas pela API serão exibidas ao usuário. Nele a rota "/resultados" possui uma função chamada "resultados_page" que renderiza o arquivo "index.html", que é responsável por exibir o menu lateral e permitir que o usuário selecione uma delas para visualizar seus métodos.

```
1     @app.route('/obter_resultados', methods=['GET'])
2     def obter_resultados():
3         global resultados
4         return jsonify(resultados=resultados)
```

Exemplo 3.11 – Código Python utilizando Flask para criar a rota obter resultados.

O código do Exemplo 3.11 é responsável por fornecer as classes obtidas para a *interface web* de forma dinâmica utilizando o formato JSON. Quando a rota "/obter_resultados" for acessada o função "obter_resultados" será executada, com o intuito de utilizar a variável global "resultados", onde as classes foram armazenadas na execução da rota "/enviardados", no Exemplo 3.9, converter a lista de classes armazenadas em um objeto JSON e retornar para o *frontend*.

```
1     @app.route('/obter_metodos/<classe>')
2     def obter_metodos(classe):
3         try:
4         prompt = f"Liste todos os metodos pertencentes a classe {classe}
      explicando suas funcionalidades e forneca um exemplo pratico para
     cada metodo listado. Ao final tambem forneca um exemplo completo
     utilizando os metodos listados anteriormente."
5         texto = chat_session.send_message(prompt)
6         texto_str = texto.text
7         response = re.sub(r"['']|(?<!\d)\*(?!\d)", "", texto_str)
```

```
8         resposta_texto = response.strip()
9
10        if ":" in resposta_texto:
11            pos = resposta_texto.find(":") + 1
12            texto_apos_colon = resposta_texto[pos:].strip()
13
14            blocos_metodos = re.split(r'\n(?=\w+\()', texto_apos_colon)
15
16            resposta_lista = [bloco.strip() for bloco in blocos_metodos
    if bloco.strip()]
17        else:
18            resposta_lista = [resposta_texto]
19
20        return jsonify({"metodos": resposta_lista})
21
22    except Exception as e:
23        print(f"Erro ao carregar metodos: {str(e)}")
24        return jsonify({"erro": f"Erro ao carregar metodos: {str(e)}"}),
    500
```

Exemplo 3.12 – Código Python utilizando Flask para criar a rota obter metodos.

O código do Exemplo 3.12 é responsável por buscar e retornar, de maneira detalhada, os métodos e exemplos práticos pertencentes a uma classe específica.

Na primeira linha é criado a rota "obter_metodos" que recebe o nome de uma classe como parâmetro, e que após ser acessada realiza a chamada da função "obter_metodos(classe)".

Na linha 3 é iniciado o tratamento de excessão, onde um segundo *prompt* é criado, sendo ele "Liste todos os metodos pertencentes a classe classe explicando suas funcionalidades e forneca um exemplo pratico para cada metodo listado. Ao final tambem forneca um exemplo completo utilizando os metodos listados anteriormente .", e em seguida enviado a API Gemini para obter a resposta. Após receber o resultado é feito a remoção de possíveis caracteres indesejados. Na linha 10 se o texto possuir ":" será selecionado apenas a parte do texto após esse sinal, dividi-lo em blocos e o armazenar em uma lista chamada "resposta_lista".

```
1     @app.route('/metodos')
2     def metodos_page():
3         return render_template('metodos.html')
```

Exemplo 3.13 – Código Python utilizando Flask para criar a rota metodos.

O código do Exemplo 3.13 é responsável em carregar a página que exibirá os métodos de uma classe de API. Na primeira linha é definida a rota "/metodos" que, ao ser acessada, chamará a função "metodos_page" que carrega e exibe o arquivo "metodos.html".

```
1     @app.route('/voltar')
2     def voltar():
```

```
3         return render_template('principal.html')
4
5     if __name__ == '__main__':
6         app.run(debug=True)
```

Exemplo 3.14 – Código Python utilizando Flask para criar a rota voltar.

O código no Exemplo 3.14 permite que o usuário retorne à página principal do sistema. Onde uma rota chamada "/voltar" foi criada para quando o usuário clicar no *link* o *Flask* chama a função "voltar():" que vai renderizar a página "principal.html".

```
1     if __name__ == '__main__':
2         app.run(debug=True)
```

Exemplo 3.15 – Código Python utilizando Flask para criar a rota voltar.

O trecho de código do Exemplo 3.15 é responsável por iniciar o servidor *Flask*, permitindo o acesso às páginas e funcionalidades presentes no projeto.

3.4 Funcionamento da Página *Web*

A *interface web* desenvolvida tem como objetivo oferecer uma forma simples e intuitiva de interação entre o usuário e o sistema de geração de documentação. Por meio dessa página, o usuário informa os parâmetros da API desejada e visualiza a documentação gerada pela LLM.

A página contém campos de entrada para o nome da API e a Linguagem de programação desejada (ex: java, *python*, entre outras). Como apresentado na Figura 1.

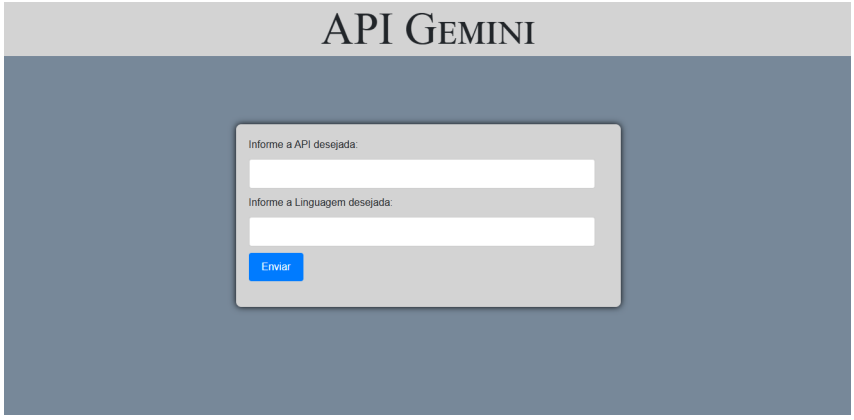

Figura 1 – Página Principal

Após o preenchimento dos dados, o usuário aciona a funcionalidade por meio de um botão, o qual envia as informações ao *backend* via método POST. O *backend* processa esses dados e envia o *prompt*.

O conteúdo gerado pela LLM é então retornado e exibido, dentro de uma área dedicada à visualização da documentação. Conforme apresentado na Figura 2.

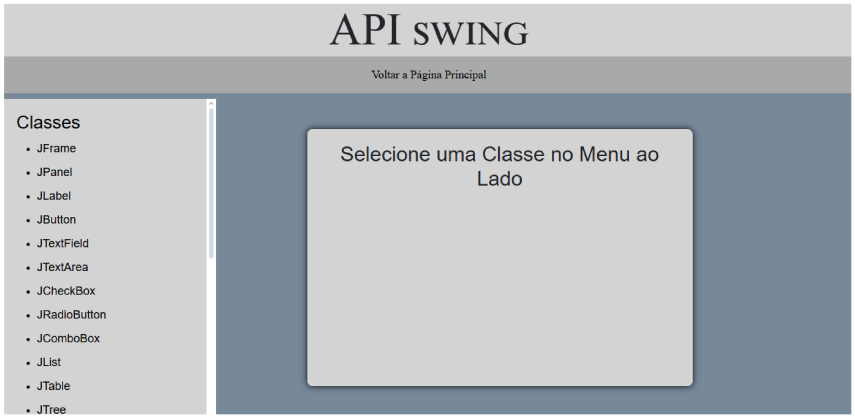

Figura 2 – Página Web com as Classes da API escolhida como exemplo

Em seguida o usuário escolherá uma classe para que sejam gerados os exemplos dos métodos que ela possui. A página *web* retornará os resultados conforme mostrado na Figura 3.

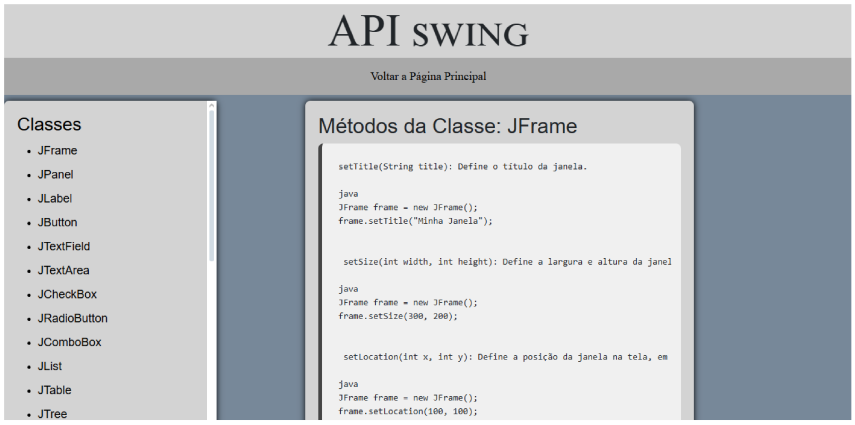

Figura 3 – Página Web com os exemplos dos métodos da API escolhida como exemplo

Capítulo 4

Método e Análise dos Resultados

Neste capítulo serão apresentados os procedimentos metodológicos adotados para a realização deste estudo, e também a análise dos resultados obtidos. Serão detalhados os métodos de avaliação e comparação utilizados, permitindo interpretar os resultados de forma consistente. Em seguida, busca-se evidenciar como os resultados se relacionam com os objetivos propostos, destacando padrões, tendências, e *insights* relevantes obtidos a partir da aplicação da abordagem proposta.

4.1 Método para a Avaliação

Este estudo utiliza metodologia quantitativa para avaliar o percentual de cobertura de métodos e classes presentes nas documentações de APIs, geradas pelas LLMs avaliados.

A primeira etapa do processo consiste no desenvolvimento de um código em *Python* que irá utilizar o modelo LLM para gerar a documentação automaticamente. Esse código fornecerá descrições e exemplos contextualizados com base nas especificações da API. Para facilitar a interação do usuário será criada uma interface *web*, onde serão coletadas o nome da API e o nome da linguagem de programação desejada. Para a realização desta etapa será utilizada a ferramenta *VS Code* para implementação. Também terá uma análise feita com cada *prompt*, separadamente, para verificar de forma clara os resultados retornados pelas LLMs.

Em seguida, será realizada uma comparação entre os diferentes tipos de documentação. Para isso, serão selecionadas duas abordagens distintas: a documentação gerada automaticamente pelo modelo LLM e a documentação oficial da API. Serão estabelecidas métricas qualitativas e quantitativas para avaliar a qualidade das documentações, considerando alguns aspectos, como por exemplo a clareza, a aplicabilidade, entre outros.

Por fim, os dados coletados serão analisados. Com base nos resultados obtidos, serão elaboradas conclusões sobre a eficácia da documentação gerada por LLM, além de recomendações para melhorar futuras gerações de documentação.

4.2 Avaliação dos Resultados

Esta seção apresenta os testes conduzidos com o intuito de avaliar a eficácia e aplicabilidade da documentação gerada automaticamente por LLMs. Os testes foram divididos em duas etapas: a análise comparativa entre as documentações oficiais e as geradas automaticamente, e a avaliação prática realizada com usuários desenvolvedores.

Os cálculos são feitos conforme a quantidade de classes e/ou métodos retornados, e a quantidade presente na documentação oficial das API. Onde para obter as classes de uma determinada API será utilizado o seguinte *prompt*:

Prompt 1: "Liste apenas os nomes de todas as classes da API {X} com a linguagem {Y}."

Onde as variáveis 'X' e 'Y' serão substituídas pela API e pela linguagem de programação escolhidas respectivamente. Logo após essa análise será utilizado um segundo *prompt*:

Prompt 2: "Liste todos os métodos pertencentes à classe {X} da api {Y} {Z} explicando suas funcionalidades e forneça um exemplo prático para cada método listado. Ao final também forneça um exemplo completo utilizando os métodos listados anteriormente."

As variáveis 'X', 'Y' e 'Z' são a classe, a API e a linguagem escolhida respectivamente. E então para análise, serão consideradas quatro métricas:

- Quantidade, que será o total de resultados retornados por cada uma das LLMs;
- Precisão, que será o valor dos verdadeiros positivos, dividido pelo total retornado;
- *Recall*, que será o valor de verdadeiros positivos dividido pelo total presente na documentação oficial;
- *F1-Score*, que combina a avaliação da precisão com o *recall* em uma única medida.

$$\text{F1-Score} = 2 \cdot \frac{\text{Precisão} \cdot \text{Recall}}{\text{Precisão} + \text{Recall}} \tag{1}$$

4.2.1 Análise dos Resultados para a API Swing linguagem Java

Para primeira comparação, foi escolhido a linguagem de programação Java e API Swing, utilizada no trabalho de pesquisa realizado por (ROCHA; MAIA, 2016).

Os resultados retornados conforme os *prompts* foram:

Classes da API Swing Java				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	115	60,90%	49,29%	54,50%
Gemini	20	100%	14,08%	24,68%
DeepSeek	50	100%	35,20%	52,07%
Total na documentação oficial: 142				

Tabela 2 – Comparativo das classes da API Swing entre diferentes LLMs

A Tabela 2 apresenta a avaliação das respostas geradas por três modelos de linguagem (LLMs) — ChatGPT, Gemini e DeepSeek — ao listar as classes da API *Swing* em Java. Os resultados são comparados com a documentação oficial, que lista um total de 142 classes.

O modelo ChatGPT identificou 115 classes, apresentando uma precisão de 60,90%, *recall* de 49,29% e *F1-Score* de 54,50%. Isso indica que pouco mais metade delas correspondem às existentes na documentação oficial.

O modelo Gemini, por sua vez, retornou 20 classes, todas presentes na documentação oficial, o que resultou em uma precisão de 100%, mas um *recall* baixo de 14,08%. Isso mostrou a preferência do modelo em retornar apenas resultados com alta confiança.

Já o modelo DeepSeek retornou 50 classes com 100% de precisão e com um *recall* de 35,20%, resultando em um *F1-Score* de 52,07%. Esse resultado mostra um equilíbrio entre confiabilidade e abrangência em comparação ao Gemini.

Após isso, foram selecionadas as 10 primeiras classes listadas pelas LLMs para verificar se os métodos retornados estavam presentes na documentação oficial da API. Sendo elas as listadas nas tabelas a seguir:

Métodos JButton				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	15	100%	3,85%	7,41%
Gemini	10	100%	2,57%	5,01%
DeepSeek	50	96%	12,34%	21,87%
Total na documentação oficial: 389				

Tabela 3 – Comparativo dos métodos da classe JButton entre diferentes LLMs

Na Tabela 3, apenas o modelo DeepSeek não apresentou precisão de 100%, os demais retornaram métodos totalmente presentes na documentação oficial. Contudo, os valores de *recall* foram baixos para todos os modelos, especialmente para o ChatGPT (3,85%) e o Gemini (2,57%), indicando que apenas uma pequena parcela dos métodos documentados foi recuperada. O modelo DeepSeek se destacou por apresentar uma quantidade maior

de saídas (50), mantendo uma alta precisão (96%) e alcançando os melhores valores de *recall* (12,34%) e *F1-score* (21,87%).

Métodos JFrame				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	15	100%	4,08%	7,84%
Gemini	9	100%	2,45%	4,78%
DeepSeek	42	100%	11,44%	20,53%
Total na documentação oficial: 367				

Tabela 4 – Comparativo dos métodos da classe JFrame entre diferentes LLMs

Na Tabela 4, todos os modelos avaliados apresentaram precisão de 100%, ou seja, todos os métodos retornados por eles estão presentes na documentação oficial da classe JFrame. No entanto, os valores de *recall* foram baixos, evidenciando que os modelos identificaram apenas uma pequena fração do total de métodos disponíveis (367). O ChatGPT obteve recall de 4,08%, enquanto o Gemini atingiu apenas 2,45%. O modelo DeepSeek, mais uma vez, se destacou por gerar mais saídas (42) e alcançar o melhor desempenho em *recall* (11,44%) e *F1-score* (20,53%).

Métodos JTable				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	20	100%	4,11%	7,90%
Gemini	12	100%	2,47%	4,82%
DeepSeek	52	100%	10,70%	19,33%
Total na documentação oficial: 486				

Tabela 5 – Comparativo dos métodos da classe JTable entre diferentes LLMs

Na Tabela 5, observa-se que todos os modelos mantiveram uma precisão de 100%, indicando que os métodos gerados pelas LLMs estão corretamente presentes na documentação oficial da classe. Contudo, o *recall* segue baixo para todos os modelos. O ChatGPT apresentou um *recall* de 4,11% com 20 métodos retornados, enquanto o Gemini obteve um valor ainda menor (2,47%) com apenas 12 saídas. O DeepSeek novamente teve o melhor desempenho, com maior número de métodos gerados (52) e os melhores valores de *recall* (10,70%) e *F1-score* (19,33%).

Métodos JTree				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	8	100%	1,64%	3,23%
Gemini	17	100%	3,50%	6,76%
DeepSeek	43	93,02%	8,43%	15,46%
Total na documentação oficial: 486				

Tabela 6 – Comparativo dos métodos da classe JTree entre diferentes LLMs

Na Tabela 6, percebe-se novamente uma alta precisão por parte das LLMs, com ChatGPT e Gemini alcançando 100%, enquanto o DeepSeek teve uma leve queda para 93,02%, indicando a presença de alguns métodos não oficiais entre os gerados. Apesar disso, o DeepSeek se destacou mais uma vez por retornar a maior quantidade de métodos (43), o que resultou nos melhores valores de *recall* (8,43%) e *F1-score* (15,46%).

Métodos JCheckbox				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	15	100%	3,37%	6,52%
Gemini	5	100%	1,12%	2,22%
DeepSeek	15	100%	3,37%	6,52%
Total na documentação oficial: 445				

Tabela 7 – Comparativo dos métodos da classe JCheckbox entre diferentes LLMs

Na Tabela 7, todos os modelos alcançaram 100% de precisão, indicando que os métodos gerados estão de fato presentes na documentação oficial da classe. No entanto, o número de métodos identificados foi bastante baixo em relação ao total oficial (445). ChatGPT e DeepSeek retornaram 15 métodos, com *recall* de apenas 3,37% e *F1-score* de 6,52%. O Gemini teve desempenho menor ainda, com apenas 5 métodos identificados, *recall* de 1,12% e *F1-score* de 2,22%.

Métodos JList				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	8	100%	1,88%	3,69%
Gemini	10	100%	2,35%	4,59%
DeepSeek	52	100%	12,23%	21,79%
Total na documentação oficial: 425				

Tabela 8 – Comparativo dos métodos da classe JList entre diferentes LLMs

Na Tabela 8, observa-se novamente que todos os modelos alcançaram 100% de precisão, garantindo que os métodos listados são corretos conforme a documentação oficial

da classe. O DeepSeek se destacou por gerar uma quantidade muito superior de métodos (52), atingindo o melhor *recall* (12,23%) entre os modelos, embora seu *F1-score* (9,07%) tenha sido ligeiramente inferior ao de outras tabelas anteriores. ChatGPT e Gemini apresentaram um menor desempenho, retornando apenas 8 e 10 métodos, respectivamente, com *recalls* baixos.

Métodos JSlider				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	19	100%	4,75%	9,07%
Gemini	21	100%	5,25%	9,98%
DeepSeek	61	100%	15,25%	26,46%
Total na documentação oficial: 400				

Tabela 9 – Comparativo dos métodos da classe JSlider entre diferentes LLMs

Na Tabela 9, o modelo DeepSeek novamente se destaca, com uma quantidade significativa de métodos (61), o que resulta em um melhor *recall* (15,25%) e em um maior *F1-score* (26,46%) entre os modelos comparados. Isso mostra que, apesar das demais LLMs apresentarem precisão de 100%, DeepSeek conseguiu cobrir uma parte maior da documentação oficial.

Métodos JRadioButton				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	19	100%	4,30%	8,25%
Gemini	9	100%	2,04%	4,00%
DeepSeek	43	100%	9,75%	17,77%
Total na documentação oficial: 441				

Tabela 10 – Comparativo dos métodos da classe JRadioButton entre diferentes LLMs

Na Tabela 10, os resultados revelam um padrão semelhante ao das outras classes analisadas. O modelo DeepSeek se destaca com uma quantidade considerável de métodos (43), apresentando o melhor *recall* (9,75%) e *F1-score* (17,77%) entre os três modelos.

Métodos JSpinner				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	10	100%	2,69%	5,24%
Gemini	10	100%	2,69%	5,24%
DeepSeek	15	100%	4,04%	7,77%
Total na documentação oficial: 371				

Tabela 11 – Comparativo dos métodos da classe JSpinner entre diferentes LLMs

Na Tabela 11, observa-se que tanto o ChatGPT quanto o Gemini apresentaram o mesmo desempenho em termos de quantidade de métodos (10), precisão (100%), *recall* (2,69%) e *F1-score* (5,24%). Enquanto o modelo DeepSeek, com 15 métodos listados, alcançou um *recall* de 4,04% e um *F1-score* de 7,77%, superando os outros dois modelos.

Métodos JTextField				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	18	100%	4,01%	7,71%
Gemini	10	100%	2,23%	4,36%
DeepSeek	57	100%	12,72%	22,57%
Total na documentação oficial: 448				

Tabela 12 – Comparativo dos métodos da classe JTextField entre diferentes LLMs

Na Tabela 12, o modelo DeepSeek se destaca novamente, com 57 métodos identificados e uma precisão de 100%. O modelo obteve um *recall* de 12,72% e um *F1-score* de 22,57%, que são os melhores resultados entre os três modelos analisados.

Média dos Métodos Swing				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	14,70	100%	3,47%	6,69%
Gemini	11,30	100%	2,67%	5,18%
DeepSeek	43	98,90%	10,03%	18,01%
Média da quantidade total na documentação oficial: 425,8				

Tabela 13 – Média dos métodos listados da API Swing entre diferentes LLMs

Na Tabela 13 apresenta a média dos resultados obtidos na extração dos métodos das classes da API *Swing*, observa-se que o modelo *DeepSeek* se manteve superior em relação aos demais. Mesmo com uma leve redução na precisão média (98,90%), ele manteve uma quantidade média de métodos listados significativamente maior (43) e obteve os melhores índices de *recall* (10,03%) e *F1-score* (18,01%).

Com relação aos exemplos pedidos no *prompt2*, em sua grande maioria foram retornados amostras de aplicação de seus métodos, mas nem todas as LLMs retornavam um exemplo completo ao final como solicitado. O modelo Gemini foi a IA que em sua maioria retornou os exemplos aplicáveis de cada método e um exemplo completo.

Portanto a análise dos resultados apresentados nas tabelas revelou que, entre os três modelos LLMs avaliados (ChatGPT, Gemini e DeepSeek), o modelo DeepSeek se destaca em termos de *recall* e *F1-score*. Embora todos os modelos apresentem uma alta precisão, o DeepSeek foi o único que conseguiu identificar uma quantidade significativamente maior

de métodos, com resultados superiores em termos de cobertura, embora ainda abaixo do total presente nas documentações oficiais.

4.2.2 Análise dos Resultados para a API OpenCV linguagem Python

Agora, os mesmos procedimentos serão realizados utilizando a API OpenCV com a linguagem *Python*, que tem se destacado em diversos projetos devido a uma de suas principais funcionalidades: a leitura e exibição de imagens, em diferentes formatos, e de vídeos. Essa capacidade permite a captura e manipulação de mídia tanto em tempo real quanto a partir de arquivos previamente gravados.

OpenCV Python				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	58	91,38%	2,25%	4,39%
Gemini	22	100%	0,93%	1,84%
DeepSeek	100	87%	3,70%	7,10%
Total na documentação oficial: 2351				

Tabela 14 – Comparativo das classes da API OpenCV python entre diferentes LLMs

A Tabela 14 apresenta uma comparação entre três LLMs — ChatGPT, Gemini e DeepSeek — quanto à sua capacidade de identificar classes presentes na API OpenCV para Python. Observa-se que, embora a quantidade absoluta de classes reconhecidas pelas LLMs varie consideravelmente (com DeepSeek listando 100 classes e Gemini apenas 22), a precisão dos modelos é alta, ultrapassando 87% em todos os casos e chegando a 100% no Gemini. No entanto, métricas como *recall* e *F1-Score* revelam uma limitação importante: todas as LLMs obtiveram baixos valores de *recall* (menores que 4%), indicando que elas reconhecem apenas uma fração pequena do total real de classes da documentação oficial (2.351 classes). Isso sugere que, apesar de serem assertivas nas classes que identificam (alta precisão), os modelos deixam de citar uma grande quantidade de elementos existentes na API, o que compromete sua completude. A análise do *F1-Score*, que equilibra precisão e *recall*, reforça essa limitação, com valores baixos em todas as LLMs testadas.

Assim como a API da subseção anterior, serão selecionadas as 10 primeiras classes listadas pelas LLMs para análise. As tabelas a seguir apresentam os resultados com os métodos das classes pertencentes a API OpenCV.

Métodos VideoCapture				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	8	75%	21,43%	33,34%
Gemini	6	100%	21,43%	35,30%
DeepSeek	9	100%	32,14%	48,65%
Total na documentação oficial: 28				

Tabela 15 – Comparativo dos métodos da classe VideoCapture entre diferentes LLMs

A Tabela 15 apresenta os resultados da avaliação de diferentes LLMs na identificação dos métodos pertencentes à classe VideoCapture da API OpenCV. Os dados indicam que todos os modelos obtiveram alta precisão, com Gemini e DeepSeek atingindo 100%, o que demonstra que os métodos mencionados por essas LLMs são consistentes com a documentação oficial. No entanto, os valores de *recall* revelam que apenas uma parte dos métodos totais foi identificada, com o DeepSeek alcançando o melhor resultado (32,14%). O *F1-Score* também destaca o desempenho superior do DeepSeek (48,65%), seguido por Gemini (35,30%) e ChatGPT (33,34%).

Métodos VideoWriter				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	6	100%	27,27%	42,85%
Gemini	5	100%	22,72%	37,03%
DeepSeek	4	100%	18,18 %	30,77%
Total na documentação oficial: 22				

Tabela 16 – Comparativo dos métodos da classe VideoWriter entre diferentes LLMs

Os resultados da Tabela 16 mostram que todos os modelos atingiram 100% de precisão, evidenciando que os métodos listados por cada LLM correspondem exatamente aos métodos oficiais. Contudo, a métrica de *recall* demonstra que a cobertura ainda é limitada: o ChatGPT obteve o melhor resultado com 27,27%, seguido por Gemini com 22,72% e DeepSeek com 18,18%. Esses valores indicam que, embora os métodos fornecidos sejam corretos, uma quantidade significativa dos métodos disponíveis na documentação oficial não foi mencionada pelos modelos. O *F1-Score* reflete essa situação, variando de 30,77% a 42,85%, e reforça que as LLMs analisadas, apesar de assertivas, ainda não garantem uma cobertura completa no contexto específico da classe VideoWriter.

Métodos SIFT				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	5	100%	16,66%	28,56%
Gemini	3	100%	10,10%	18,18%
DeepSeek	4	100%	13,13%	23,52%
Total na documentação oficial: 30				

Tabela 17 – Comparativo dos métodos da classe SIFT entre diferentes LLMs

Na Tabela 17 os resultados mostram que todos os modelos alcançaram 100% de precisão. No entanto, o desempenho em *recall* permanece reduzido, evidenciando que apenas uma parte dos métodos totais foi recuperada: o ChatGPT apresentou o maior *recall*, com 16,66%, seguido pelo DeepSeek (13,13%) e Gemini (10,10%). O *F1-Score* variou de 18,18% a 28,56%, reforçando a limitação dos modelos em termos de cobertura completa dos métodos disponíveis (total de 30 na documentação oficial).

Métodos BFMatcher				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	5	100%	20,83%	34,48%
Gemini	3	100%	12,50%	22,22%
DeepSeek	4	100%	16,66%	28,56%
Total na documentação oficial: 24				

Tabela 18 – Comparativo dos métodos da classe BFMatcher entre diferentes LLMs

Na Tabela 18, os três modelos de LLMs alcançaram 100% de precisão, o que confirma que as respostas fornecidas são consistentes com a documentação oficial. Como esta classe herda métodos de uma outra, chamada *DescriptorMatcher*, houve um aumento no número de métodos o que fez os resultados terem uma queda. O ChatGPT obteve o melhor resultado identificando 20,83% dos métodos disponíveis, seguido por DeepSeek com 16,66% e Gemini com 12,50%. No *F1-Score* o ChatGPT também se destacou, alcançando 34,48%.

Métodos AKAZE				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	4	100%	11,76%	21,05%
Gemini	4	100%	11,76%	21,05%
DeepSeek	3	100%	8,82%	16,21%
Total na documentação oficial: 34				

Tabela 19 – Comparativo dos métodos da classe AKAZE entre diferentes LLMs

A Tabela 19 mostra que os três modelos — ChatGPT, Gemini e DeepSeek — apresentaram uma precisão máxima de 100%, indicando que todas as respostas fornecidas por eles correspondem corretamente à documentação oficial. Apesar desse índice elevado de acerto, os valores de *recall* revelam uma cobertura bastante limitada: tanto ChatGPT quanto Gemini identificaram apenas 11,76% dos métodos, enquanto DeepSeek apresentou um desempenho ligeiramente inferior, com 8,82%. Os resultados de *F1-Score*, que variaram entre 16,21% e 21,05%, reforçam essa limitação.

Métodos CascadeClassifier				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	7	71,43%	31,25%	43,48%
Gemini	4	100%	25%	40%
DeepSeek	4	100%	25%	40%
Total na documentação oficial: 16				

Tabela 20 – Comparativo dos métodos da classe CascadeClassifier entre diferentes LLMs

A Tabela 20, no que se refere à precisão, observou-se uma pequena diferença entre os modelos: enquanto Gemini e DeepSeek mantiveram uma taxa perfeita de 100%, o ChatGPT obteve uma precisão de 71,43%, indicando a presença de alguns métodos incorretos ou não oficiais em suas respostas. Já os índices de *recall* evidenciam uma cobertura moderada, com o ChatGPT atingindo 31,25%, seguido por Gemini e DeepSeek, ambos com 25%. Os valores de *F1-Score* variaram de 40% a 43,48%, mostrando um desempenho relativamente próximo entre os modelos.

Métodos ORB				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	5	100%	13,51%	23,80%
Gemini	3	100%	8,11%	15%
DeepSeek	9	100%	24,32%	39,12%
Total na documentação oficial: 37				

Tabela 21 – Comparativo dos métodos da classe ORB entre diferentes LLMs

A Tabela 21 mostra que todos os modelos atingiram 100% de precisão. No entanto, a cobertura dos métodos disponíveis variou de forma significativa entre os modelos. O DeepSeek foi o que mais se aproximou da totalidade, com um *recall* de 24,32%, enquanto o ChatGPT alcançou 13,51% e o Gemini, 8,11%. Consequentemente, os valores de *F1-Score* seguiram essa mesma tendência, com DeepSeek apresentando o melhor equilíbrio entre precisão e *recall* (39,12%).

Métodos KAZE				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	16	100%	32,26%	48,78%
Gemini	4	100%	12,90%	22,85%
DeepSeek	4	100%	12,90%	22,85%
Total na documentação oficial: 31				

Tabela 22 – Comparativo dos métodos da classe KAZE entre diferentes LLMs

Na Tabela 22, todos os modelos mantiveram uma precisão de 100%. No entanto, a abrangência dos métodos identificados variou consideravelmente. O ChatGPT destacou-se ao recuperar 32,26% do total de métodos, apresentando também o maior *F1-Score* de 48,78%.

Métodos BRISK				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	4	100%	14,81%	25,80%
Gemini	5	100%	18,52%	31,25%
DeepSeek	11	100%	40,74%	57,89%
Total na documentação oficial: 27				

Tabela 23 – Comparativo dos métodos da classe BRISK entre diferentes LLMs

Na Tabela 23, como observado nas demais classes, todos os modelos atingiram 100% de precisão. No entanto, com relação ao *recall*, houve diferenças: o DeepSeek apresentou o melhor desempenho, recuperando 40,74% dos métodos oficiais e alcançando o maior *F1-Score* (57,89%). Já o Gemini e o ChatGPT obtiveram *recalls* mais modestos, de 18,52% e 14,81%, respectivamente.

Métodos MSER				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	3	100%	7,69%	14,28%
Gemini	1	100%	2,56%	4,99%
DeepSeek	12	100%	30,77%	47,06%
Total na documentação oficial: 39				

Tabela 24 – Comparativo dos métodos da classe MSER entre diferentes LLMs

Na Tabela 24 todos os modelos mantiveram precisão total (100%), evidenciando que os métodos retornados são corretos conforme a documentação oficial. No entanto, o desempenho em termos de cobertura variou de forma significativa entre eles. O DeepSeek

foi o que mais se destacou, conseguindo listar 30,77% dos métodos documentados, com um *F1-Score* de 47,06%. Em contraste, o ChatGPT recuperou apenas 7,69% e o Gemini, 2,56%, com *F1-Scores* proporcionalmente mais baixos.

Média dos Métodos OpenCV				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	6,30	94,64%	19,75%	31,64%
Gemini	3,8	100%	14,55%	24,79%
DeepSeek	6,4	100%	22,29%	35,46%
Média da quantidade total na documentação oficial: 28,8				

Tabela 25 – Média dos métodos listados da API OpenCV entre diferentes LLMs

A Tabela 25 apresenta a média geral dos resultados obtidos pelas LLMs na listagem de métodos da API OpenCV. Observa-se que todas as LLMs mantêm uma precisão elevada, com destaque para Gemini e DeepSeek, que atingiram 100%. No entanto, o *recall* está abaixo dos 25%, indicando que apenas uma parte dos métodos oficialmente documentados foram recuperados por essas LLMs. O modelo DeepSeek obteve o melhor desempenho médio, tanto em *recall* (22,29%) quanto no *F1-Score* (35,46%), demonstrando maior capacidade de listar métodos corretamente e com maior abrangência. O ChatGPT ficou em segundo lugar, enquanto o Gemini, apesar da precisão perfeita, apresentou os menores valores de *recall* e *F1-Score*, mostrando ter uma listagem mais restrita.

Esses resultados reforçam que, embora as LLMs sejam ferramentas eficazes para indicar métodos válidos da API, elas ainda não garantem uma cobertura completa da documentação oficial. Portanto, seu uso é recomendado como apoio inicial na exploração da API, mas não como fonte única para desenvolvedores que buscam uma visão mais ampla dos recursos disponíveis no *OpenCV*.

4.2.3 Análise dos Resultados para a JavaIO linguagem Java

Realizando os mesmos procedimentos para a API JavaIO da linguagem *Java*, fundamental para que o desenvolvedor possa lidar com a entrada e saída de dados, seja para criar, excluir ou quaisquer outros tipos de manipulação de arquivos, Alura (2024). Os resultados retornados pelas IAs com base nos *prompts* destacados na Sessão 4.2 foram os seguintes:

Classes JavaIO Java				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	46	100%	90,20%	94,85%
Gemini	50	100%	98,04%	99,01%
DeepSeek	51	100%	100%	100%
Total na documentação oficial: 51				

Tabela 26 – Comparativo das classes da API JavaIO java entre diferentes LLMs

A Tabela 26 com o comparativos dos dados retornados das três LLMs (ChatGPT, Gemini e DeepSeek) com foco nas classes da API *JavaIO*. Observa-se que o DeepSeek conseguiu identificar todas as classes e atingiu 100% em todos os parâmetros (Precisão, *Recall* e *F1-Score*), enquanto o Gemini e o ChatGPT também apresentaram bons resultados atingindo o total de precisão e quase alcançando esse mesmo valor de *Recall* e *F1-Score*. Sendo assim, esses resultados mesmo evidenciando que, apesar de todos os modelos alcançarem uma alta precisão, existe uma variação na capacidade de cobertura dos dados, onde destaca o DeepSeek como o mais capacitado para a tarefa.

Nesta API, assim como as anteriores, foram selecionadas as 10 primeiras classes listadas pelas LLMs para análise. A seguir serão mostrados as tabelas com as classes da API *JavaIO*:

Métodos BufferedInputStream				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	8	100%	100%	100%
Gemini	8	100%	100%	100%
DeepSeek	8	100%	100%	100%
Total na documentação oficial: 8				

Tabela 27 – Comparativo dos métodos da classe BufferedInputStream entre diferentes LLMs

Na Tabela 27 todos os três modelos de LLMs foram capazes de identificar todos os métodos pertencentes a classe, alcançando 100% em todas as métricas avaliadas (precisão, *recall* e *F1-Score*).

Métodos BufferedOutputStream				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	3	100%	100%	100%
Gemini	3	100%	100%	100%
DeepSeek	3	100%	100%	100%
Total na documentação oficial: 3				

Tabela 28 – Comparativo dos métodos da classe BufferedOutputStream entre diferentes LLMs

Na Tabela 28 todas as IAs conseguiram identificar todos os métodos da classe, o que permitiu fazer com que obtessem um aproveitamento total de suas métricas.

Métodos BufferedReader				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	9	100%	90%	94,74%
Gemini	9	100%	90%	94,74%
DeepSeek	8	100%	80%	88,89%
Total na documentação oficial: 10				

Tabela 29 – Comparativo dos métodos da classe BufferedReader entre diferentes LLMs

Na Tabela 29, o ChatGPT e o Gemini identificaram 9 dos 10 métodos existentes na classe, atingindo 100% de precisão, 90% de *recall* e 94,74% de *F1-Score*, enquanto o DeepSeek, mesmo reconhecendo apenas 8 dos métodos, manteve 100% de precisão, mas com *recall* de 80% e 88,89% de *F1-Score*. Isso mostra que mesmo listando os métodos corretamente, ainda houve uma limitação na cobertura dos métodos documentados.

Métodos BufferedWriter				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	6	100%	100%	100%
Gemini	6	100%	100%	100%
DeepSeek	6	100%	100%	100%
Total na documentação oficial: 6				

Tabela 30 – Comparativo dos métodos da classe BufferedWriter entre diferentes LLMs

Na Tabela 30 todos modelos detectaram corretamente os métodos listados na documentação oficial, atingindo um aproveitamento total em todas as métricas avaliadas.

Métodos ByteArrayInputStream				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	9	100%	45%	62,07%
Gemini	8	100%	40%	57,14%
DeepSeek	8	100%	40%	57,14%
Total na documentação oficial: 20				

Tabela 31 – Comparativo dos métodos da classe ByteArrayInputStream entre diferentes LLMs

Na Tabela 31, o ChatGPT foi o que mais conseguiu reconhecer métodos da classe presentes na documentação oficial, mantendo uma precisão de 100% mas com *recall* de 45% e *F1-Score* 62,07%. Equanto o Gemini e o DeepSeek identificaram apenas 8 dos 20 métodos, mesmo com 100% de precisão, o *recall* foi de 40% e o *F1-Score* de 57,14%.

Métodos ByteArrayOutputStream				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	10	100%	45,45%	62,50%
Gemini	10	100%	45,45%	62,50%
DeepSeek	9	100%	40,91%	58,06%
Total na documentação oficial: 22				

Tabela 32 – Comparativo dos métodos da classe ByteArrayOutputStream entre diferentes LLMs

Na Tabela 32 todos os modelos conseguiram apresentar 100% de precisão, porém não tiveram uma alta cobertura dos métodos pertencentes à classe. O ChatGPT e o Gemini identificaram 10 dos 22 métodos constatados na documentação oficial, o que resultou em um *recall* de 45,45% e *F1-Score* de 62,50%, enquanto o DeepSeek reconheceu apenas 9 dos métodos, obtendo um *recall* de 40,91% *F1-Score* de 58,06%.

Métodos CharArrayReader				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	7	100%	33,33%	50%
Gemini	7	100%	33,33%	50%
DeepSeek	8	100%	38,10%	55,17%
Total na documentação oficial: 21				

Tabela 33 – Comparativo dos métodos da classe CharArrayReader entre diferentes LLMs

Na Tabela 33, todas as três LLMs conseguiram atingir 100% de precisão, mas o DeepSeek foi a que conseguiu ter um maior alcance e encontrar 8 dos 21 métodos da classe,

com *recall* de 38,10% e *F1-Score* de 55,17%, enquanto as outas IAs encontraram apenas 7 métodos, alcançando um *recall* de 33,33% e *F1-Score* de 50%.

Métodos CharArrayWriter				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	12	100%	48%	64,86%
Gemini	13	100%	52%	68,42%
DeepSeek	12	100%	48%	64,86%
Total na documentação oficial: 25				

Tabela 34 – Comparativo dos métodos da classe CharArrayWriter entre diferentes LLMs

Na Tabela 34 todas as IAs conseguiram atingir 100% de precisão, e dentre elas a Gemini foi a que conseguiu obter uma maior cobertura dos métodos, encontrando 13 dos 25 presentes na documentação oficial, com um *recall* de 52% e *F1-Score* 68,42%, enquanto as demais LLMs conseguiram um *recall* de 48% e *F1-Score* de 64,86%.

Métodos Console				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	8	100%	40%	57,14%
Gemini	6	100%	30%	46,15%
DeepSeek	9	100%	45%	62,07%
Total na documentação oficial: 20				

Tabela 35 – Comparativo dos métodos da classe Console entre diferentes LLMs

Na Tabela 35 todas as LLMs conseguiram ter 100% de precisão, porém nenhuma conseguiu ter um alto nível de cobertura dos métodos existentes na classe, onde o que demonstrou um melhor desempenho foi o DeepSeek cobrindo 9 dos 20 métodos presentes, permitindo um *recall* de 45% e *F1-Score* de 62,07%, seguido pelo ChatGPT e Gemini, respectivamente.

Métodos DataInputStream				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	15	100%	41,67%	58,82%
Gemini	14	100%	38,89%	56%
DeepSeek	14	100%	38,89%	56%
Total na documentação oficial: 36				

Tabela 36 – Comparativo dos métodos da classe DataInputStream entre diferentes LLMs

Na Tabela 36 todos os modelos obteram 100% de precisão, porém apresentaram uma cobertura limitada onde o que melhor contribuiu foi o ChatGPT, com 41,67% de *recall* e 58,82% de *F1-Score*. Enquanto o Gemini e o DeepSeek mostraram um *recall* de 38,89% e um *F1-Score* de 56%.

A seguir foi feita uma analise envolvendo todas as classes destacadas para realizar uma média com relação as LLMs e as métricas utilizadas.

Média dos Métodos JavaIO				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	8,7	100%	64,35%	75,01%
Gemini	8,4	100%	62,67%	73,50%
DeepSeek	8,5	100%	63,09%	74,22%
Média da quantidade total na documentação oficial: 17,1				

Tabela 37 – Média dos métodos listados da API JavaIO entre diferentes LLMs

Na Tabela 37, todas as LLMs obtiveram 100% de precisão, permitindo confiança nos dados retornados. Com relação ao *Recall* e ao *F1-Score* o ChatGPT se destacou superior com 64,35% e 75,01% respectivamente, seguido pelo DeepSeek e Gemini. Esses resultados mostram que, embora com a precisão perfeita, ainda há uma limitação entre os modelos, o que pode impactar a integridade da documentação gerada automaticamente.

4.2.4 Análise dos Resultados para a API Requests linguagem Python

Com os mesmos procedimentos para a API Requests da linguagem *Python*, escolhida por ser bastante utilizada para desenvolvimento *web*, facilitando o envio e recebimento de dados pela *internet*, e com uma documentação com menos componentes em comparação a API OpenCV. Os resultados retornados pelas IAs com base nos *prompts* destacados na Sessão 4.2 foram:

Classes da API Requests				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	24	45,83%	55%	50%
Gemini	5	100%	25%	40%
DeepSeek	7	100%	35%	51,85%
Total na documentação oficial: 20				

Tabela 38 – Comparativo das classes da API Requests java entre diferentes LLMs

Na Tabela 38, o ChatGPT identificou 24 classes, ultrapassando a quantidade total presente na documentação, que é 20, o que afetou seu resultado nas métricas, com precisão

de 45,83%, *recall* de 55% e *F1-Score* de 50%. Enquanto isso as outras LLMs listaram uma quantidade menor de classes, ambas com 100% de precisão, mas com *recalls* inferior ao do ChatGPT, contudo no *F1-Score* o DeepSeek apresentou um melhor número em comparação com os demais. Esses resultados mostraram que o ChatGPT, mesmo com riscos, priorizou atender uma maior cobertura, e enquanto o Gemini e o DeepSeek optaram por respostas certeiras com uma menor cobertura.

Assim como as demais APIs, foram selecionadas as 10 primeiras classes listadas pelas LLMs para avaliação. A seguir as tabelas com as análises das classes da API Requests:

Métodos Request				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	2	50%	33,33%	40%
Gemini	2	50%	33,33%	40%
DeepSeek	3	66,67%	66,67%	66,67%
Total na documentação oficial: 3				

Tabela 39 – Comparativo dos métodos da classe Request entre diferentes LLMs

Na Tabela 39, o DeepSeek foi o que apresentou melhor desempenho dentre as LLMs, encontrando mais métodos da classe, com precisão, *recall* e *F1-Score* de 66,67%.

Métodos Response				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	6	83,33%	20,83%	33,33%
Gemini	10	100%	41,67%	58,82%
DeepSeek	16	100%	66,67%	80%
Total na documentação oficial: 24				

Tabela 40 – Comparativo dos métodos da classe Response entre diferentes LLMs

Na Tabela 40, o DeepSeek também se destacou, identificando 16 dos 24 métodos oficiais, com 100% de precisão, 66,67% de *recall* e 80% de *F1-Score*. O Gemini listou 10 métodos com 100% de precisão, 41,67% de *recall* e 58,82% de *F1-Score*, e o ChatGPT teve um desempenho inferior, listando 6 métodos, com 83,33% de precisão, 20,83% de *recall* e 33,33% de *F1-Score*.

Métodos PreparedRequest				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	15	73,33%	64,71%	68,75%
Gemini	5	100%	29,41%	45,45%
DeepSeek	9	100%	52,94%	69,23%
Total na documentação oficial: 17				

Tabela 41 – Comparativo dos métodos da classe PreparedRequest entre diferentes LLMs

Na Tabela 41, o ChatGPT mostrou melhor desempenho listando 15 dos 17 métodos presentes na documentação oficial, com precisão inferior de 73,33%, mas com maior *recall*. Enquanto o DeepSeek, mesmo listando apenas 9 dos métodos, garantiu 100% de precisão e o maior *F1-Score* (69,23%). Já o Gemini não conseguiu garantir uma grande cobertura mas conseguiu garantir 100% de precisão, tendo resultados inferiores de *recall* e *F1-Score*.

Métodos Session				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	16	100%	51,61%	68,09%
Gemini	9	100%	29,03%	45%
DeepSeek	13	100%	41,94%	59,09%
Total na documentação oficial: 31				

Tabela 42 – Comparativo dos métodos da classe Session entre diferentes LLMs

Na Tabela 42, o ChatGPT se destacou dentre as LLMs, cobrindo 16 dos 31 métodos oficiais, com 100% de precisão, 51,61% de *recall* e 68,09% de *F1-Score*, enquanto o Gemini foi o que apresentou resultados inferiores, listando 9 métodos, com 100% de precisão, 29,03% de *recall* e 45% de *F1-Score*.

Métodos HTTPAdapter				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	5	80%	41,67%	54,79%
Gemini	8	87,5%	66,67%	75,68%
DeepSeek	8	87,5%	66,67%	75,68%
Total na documentação oficial: 12				

Tabela 43 – Comparativo dos métodos da classe HTTPAdapter entre diferentes LLMs

Na Tabela 43, os modelos Gemini e DeepSeek apresentaram o melhor desempenho e a mesma quantidade, listando 8 dos 12 métodos oficiais, 66,67% de *recall* e 75,68% de *F1-Score*.

Métodos BaseAdapter				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	4	50%	100%	66,67%
Gemini	3	33%	100%	50%
DeepSeek	5	40%	100%	57,14%
Total na documentação oficial: 2				

Tabela 44 – Comparativo dos métodos da classe BaseAdapter entre diferentes LLMs

Na Tabela 44, todos os modelos listaram quantidades superiores às presentes na documentação oficial. O ChatGPT apresentou uma precisão de 50%, *recall* de 100%, e *F1-Score* de 66,67%, o Gemini mostrou uma precisão de 33%, *recall* de 100%, e *F1-Score* de 50%, e o DeepSeek 40% de precisão, 100% de *recall* e 57,14% de *F1-Score*.

Métodos RequestsCookieJar				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	13	76,92%	38,46%	51,28%
Gemini	8	87,5%	30,77%	45,53%
DeepSeek	12	100%	46,15%	63,16%
Total na documentação oficial: 26				

Tabela 45 – Comparativo dos métodos da classe RequestsCookieJar entre diferentes LLMs

Na Tabela 45, o modelo DeepSeek mesmo sendo o segundo a listar uma boa quantidade de métodos, se destacou garantindo 100% de precisão, 46,15% de *recall* e 63,16% de *F1-Score*.

Métodos AuthBase				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	1	100%	100%	100%
Gemini	1	100%	100%	100%
DeepSeek	1	100%	100%	100%
Total na documentação oficial: 1				

Tabela 46 – Comparativo dos métodos da classe AuthBase entre diferentes LLMs

Na Tabela 46, todos as LLMs avaliadas conseguiram um ótimo desempenho, alcançando 100% em todas as métricas. Mesmo a classe possuindo apenas um único método, os modelos conseguiram identificá-lo, sem retornar outros que não estão na documentação oficial.

Métodos HTTPDigestAuth				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	5	100%	62,50%	76,92%
Gemini	1	100%	12,50%	22,22%
DeepSeek	4	100%	50%	66,67%
Total na documentação oficial: 8				

Tabela 47 – Comparativo dos métodos da classe HTTPDigestAuth entre diferentes LLMs

Na Tabela 47, o ChatGPT foi o modelo que mais se aproximou do total de métodos presentes na classe, com 100% de precisão e um *recall* de 62,50% resultando em um *F1-Score* de 76,92%. O DeepSeek identificou 4 métodos, com 100% de precisão, com um *recall* de 50% e *F1-Score* de 66,67%. Enquanto o Gemini apresentou uma maior limitação, reconhecendo apenas um método, o que manteve sua precisão em 100%, mas fez com que o *recall* (12,50%) não fosse tão alto, resultando em um *F1-Score* de 22,22%.

Métodos HTTPBasicAuth				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	4	100%	100%	100%
Gemini	1	100%	25%	40%
DeepSeek	4	100%	100%	100%
Total na documentação oficial: 4				

Tabela 48 – Comparativo dos métodos da classe HTTPBasicAuth entre diferentes LLMs

Na Tabela 48, os modelos ChatGPT e DeepSeek conseguiram alcançar um ótimo desempenho, identificando todos os métodos presentes na classe, com 100% de precisão, *recall*, e *F1-Score*. Enquanto o Gemini teve um desempenho mais limitado, reconhecendo apenas 1 dos 4 métodos, mantendo sua precisão em 100%, mas com um *recall* baixo, o que consequentemente afetou o resultado do *F1-Score*.

Média dos Métodos Requests				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	7,10	81,36%	61,31%	65,98%
Gemini	4,80	85,83%	46,84%	52,27%
DeepSeek	7,50	86,08%	69,10%	71,54%
Média da quantidade total na documentação oficial: 12,8				

Tabela 49 – Média dos métodos listados da API Requests entre diferentes LLMs

Na Tabela 49, foi analisado a média geral do desempenho das LLMs com a API *Requests*. Nela mostrou que o DeepSeek foi o que conseguiu melhores resultados, com

média de 7,50 métodos listados, 86,08% de precisão, 69,10% de *recall* e 71,54% de *F1-Score*. Em segundo lugar está o ChatGPT, com 7,10 de média dos métodos listados, 81,36% de precisão, 61,31% de *recall* e 65,98% de *F1-Score*. Enquanto o Gemini, mesmo apresentando uma segunda melhor precisão (85,83%), foi inferior nas demais métricas.

4.2.5 Média das LLMs por Linguagem

Uma nova análise foi feita com base nas linguagens de programação do presente estudo (*Python* e Java), obtendo a média de dados que cada uma das LLMs conseguiram alcançar, envolvendo os resultados das classes juntamente com os dos métodos.

Primeiramente considerando a linguagem Java, a média de todos os resultados geraram a seguinte tabela:

Média da Linguagem Java				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	17,95	98,22%	37,16%	43,92%
Gemini	12,14	100%	34,93%	41,38%
DeepSeek	28	99,50%	39,38%	48,83%

Tabela 50 – Média dos resultados que utilizaram a linguagem Java

Na Tabela 50, o modelo DeepSeek obteve resultados significativos, apresentando uma maior média na quantidade de itens listados, com precisão de 99,50%, *recall* de 39,38% e *F1-Score* de 48,83%. Enquanto o ChatGPT veio em seguida, com média de 17,95 itens listados, precisão de 37,16% e *F1-Score* de 43,92%. Embora o Gemini tenha atingido 100% de precisão, deixou a desejar na média de itens listados, no *recall* (34,93%) e no *F1-Score*.

Média da Linguagem *Python*				
LLM	**Quantidade**	**Precisão**	**Recall**	**F1-Score**
ChatGPT	9,82	86,24%	39,45%	46,85%
Gemini	5,14	93,56%	29,08%	36,93%
DeepSeek	11,98	93,08%	43,30%	51,32%

Tabela 51 – Média dos resultados que utilizaram a linguagem *Python*

Na Tabela 51, o modelo DeepSeek se realçou novamente, listando uma média de 11,98 resultados, uma precisão de 93,08%, *recall* de 43,30% e *F1-Score* de 51,32%. O ChatGPT veio em seguida, com média de listagem de 9,82, precisão de 86,24%, *recall* de 39,45% e *F1-Score* de 46,85%. Mais uma vez o Gemini mesmo apresentando uma maior precisão (93,56%), foi a IA que demonstrou menor desempenho nas demais métricas.

Sendo assim, os dados indicam que houve um maior número extraindo resultados da linguagem Java, possivelmente por ser mais comumente utilizada pelos desenvolvedores e por estar a mais tempo na área. Além de que dentre as LLMs em estudo, nota-se que o modelo Gemini preserva cobrir um menor número de resultados mas garantindo que estes sejam precisos, enquanto os outros buscam cobrir uma quantidade maior, o que acaba afetando na sua integridade.

4.2.6 Média das LLMs conforme a quantidade de métodos das classes

Com o objetivo de avaliar o desempenho das LLMs na identificação dos métodos presentes nas classes, foram calculadas as mesmas métricas (Precisão, *Recall*, e *F1-Score*) considerando diferentes intervalos de quantidades de métodos. Essa análise vai permitir observar como cada modelo se comporta diante das classes de maior e menor número. A seguir serão apresentadas as tabelas com os resultados obtidos para cada uma das IAs (ChatGPT, Gemini, DeepSeek):

ChatGPT			
Intervalos	**Precisão**	**Recall**	**F1-Score**
1-10	88,89%	87,31%	86,48%
11-20	84,95%	44,52%	57,25%
21-30	93,53%	28,71%	42,70%
31-60	100%	26,42%	39,14%
+60	100%	3,47%	6,69%

Tabela 52 – Média das métricas do ChatGPT, considerando a quantidade de métodos nas classes das APIs avaliadas, agrupadas por intervalos de número de métodos.

A Tabela 52 mostra que o desempenho do *ChatGPT* varia de acordo com a quantidade de métodos presentes nas classes. Observa-se que, para as classes menores (1 a 10 métodos), há um equilíbrio entre as métricas, o que mostra uma boa identificação dos métodos. Porém, à medida que o número de classes aumenta, mesmo com a precisão crescendo, o *recall* diminui, afetando também os valores do *F1-score*.

Gemini			
Intervalos	**Precisão**	**Recall**	**F1-Score**
1-10	87,04%	73,43%	71,88%
11-20	97,50%	38,22%	52,89%
21-30	98,75%	28,24%	42,93%
31-60	100%	17,21%	27,48%
+60	100%	2,67%	5,18%

Tabela 53 – Média das métricas do Gemini, considerando a quantidade de métodos nas classes das APIs avaliadas, agrupadas por intervalos de número de métodos.

Pode-se observar, na Tabela 53, que o desempenho do Gemini também foi afetado com o aumento da quantidade de métodos das classes. Para as classes pequenas (1 a 10 métodos), o modelo mantém um equilíbrio razoável entre as métricas. No entanto, conforme aumenta o número dos métodos das classes, observa-se um comportamento parecido com o do ChatGPT, criando um maior número de precisão, porém com o *recall* caindo, afetando também o *F1-score*.

DeepSeek			
Intervalos	**Precisão**	**Recall**	**F1-Score**
1-10	85,93%	88,52%	84,13%
11-20	97,50%	45,92%	60,82%
21-30	100%	36,09%	51,07%
31-60	100%	26,27%	40,06%
+60	98,90%	10,03%	18,01%

Tabela 54 – Média das métricas do DeepSeek, considerando a quantidade de métodos nas classes das APIs avaliadas, agrupadas por intervalos de número de métodos.

A Tabela 54 mostra que o DeepSeek apresentou um comportamento semelhante aos demais. No intervalo de 1 a 10 métodos, o modelo mostrou um *recall* (88,52%) e uma precisão (85,93%) elevados, o que resultou em um *F1-score* alto (84,13%). Contudo, conforme os intervalos vão aumentando, o modelo melhora a precisão, mas acaba reduzindo o número do *recall*, e assim atingindo os valores do *F1-score*.

Sendo assim, a análise desses resultados permite concluir que os três modelos apresentam um padrão, onde alcançam resultados equilibrados em classes com menores quantidades de métodos, com bom desempenho na precisão e no *recall*, garantindo valores de *F1-score* elevados. Entretanto, conforme aumenta o número de métodos nas classes, a precisão tende a aumentar enquanto o número do *recall* diminui, reduzindo também o *F1-score*. Isso mostra que, mesmo que os modelos tenham apresentado resultados precisos, eles ainda possuem limitações com relação ao *recall*, uma possível explicação para esse

problema pode ser a quantidade limitada de *tokens* das LLMs gratuitas, o que pode ser minimizado por meio do uso de LLMs pagas. Essa hipótese será investigada em trabalhos futuros.

Capítulo 5

Conclusão e Trabalhos Futuros

Este capítulo apresenta a conclusão do estudo, bem como recomendações para a continuidade dos trabalhos nesta área de pesquisa.

5.1 Conclusão

O presente trabalho teve como objetivo principal analisar a geração de documentação automática para APIs utilizando Modelos de Linguagem de Grande Escala LLMs para seu desenvolvimento. Para isso foram avaliados os modelos ChatGPT, Gemini e *DeepSeek*, com foco na análise de seus resultados quanto aos níveis de precisão e cobertura das pesquisas geradas.

Os resultados obtidos evidenciaram que cada uma das LLMs possuem características e limitações diferentes em suas abordagens. O Gemini demonstrou um maior foco na precisão das informações, mesmo que isso fizesse com que sua cobertura fosse mais limitada. O ChatGPT, por sua vez, tentou cobrir uma quantidade maior, ainda que isso, em alguns casos, o tenha feito correr riscos em retornar resultados imprecisos. Já o *DeepSeek*, sendo um modelo de IA mais recente, buscou manter um equilíbrio entre abrangência e integridade, entregando um maior número de dados com menos irregularidades.

Além disso, a análise feita separadamente para cada uma das LLMs em estudo, deixou claro que elas apresentam um padrão, onde mesmo mostrando resultados precisos ainda possuem limitações com relação à capacidade de cobertura. Isso podendo ser, como hipótese, devido ao número limitado de *tokens* que elas possuem por serem versões gratuitas.

Sendo assim, o estudo mostra a relevância dos LLMs como ferramentas complementares no processo de documentação. Destacando que elas possuem um grande potencial para ampliar acesso às informações técnicas, desde que sua aplicação seja acompanhada de critérios de avaliação e revisão, garantindo qualidade dos conteúdos gerados.

5.2 Trabalhos Futuros

O presente estudo obteve resultados significativos, conforme mostrado no Capítulo 4. No entanto, diversos aspectos ainda podem ser explorados em pesquisas futuras, contribuindo para o avanço na área.

Esta pesquisa concentrou-se no uso de apenas uma LLM, a Gemini, para a geração automática de documentação por meio de uma *interface web*, o que limitou a perspectiva dos dados retornados. Assim, a utilização de outras interfaces e diferentes LLMs pode ser uma alternativa interessante para verificar quais modelos melhor atendem às necessidades dos usuários.

Nas análises feitas no Capítulo 4, foram consideradas apenas três LLMs, gratuitas, além de utilizar apenas um pequeno grupo de APIs nas linguagens Java e *Python*, escolhidas por serem amplamente utilizadas na programação. Portanto uma pesquisa futura que abranja uma escala maior de APIs e linguagens pode representar um avanço significativo.

Outro ponto a ser investigado é se modelos mais recentes e/ou versões pagas, que suportam uma maior quantidade de *tokens*, podem alcançar melhores resultados na métrica de *recall*.

Também recomenda-se conduzir uma pesquisa com desenvolvedores/usuários para avaliar a eficácia prática da *interface web* desenvolvida, bem como a qualidade dos exemplos de código gerados pelas LLMs.

Referências

ALEMU, M. B. Rest api: Implementation with flask-python. Lapin ammattikorkeakoulu, 2014. Citado na página 19.

Alura. **Pacote java.io**. 2024. <https://www.alura.com.br/apostila-java-orientacao-objetos/apendice-pacote-java-io>. Acesso em: 28 julho 2025. Citado na página 50.

APRILIA, S. et al. Penerapan api gemini dalam layanan peminjaman novel online pada website cozybook. **Jurnal Informatika dan Teknik Elektro Terapan**, v. 12, n. 3, 2024. Citado na página 18.

ARIMATSU, Y. et al. Enriching api documentation by relevant api methods recommendation based on version history. In: **2018 IEEE Third International Workshop on Dynamic Software Documentation (DySDoc3)**. [S.l.: s.n.], 2018. p. 15–16. Citado na página 21.

BASHAN, G. et al. Enriquecendo a documentação da interface de programação de aplicativos (api) usando uma arquitetura de modelo de linguagem grande (llm) para melhorar a interpretabilidade da api llm. **Technical Disclosure Commons**, August 2024. Disponível em: <https://www.tdcommons.org/dpubs_series/7293>. Citado na página 22.

CASELI, H. d. M.; NUNES, M. d. G. V. Processamento de linguagem natural: conceitos, técnicas e aplicações em português. 2024. Citado na página 17.

DHYANI, P. et al. Automated api docs generator using generative ai. In: **2024 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS)**. [S.l.: s.n.], 2024. p. 1–6. Citado na página 23.

FAN, Q. et al. Why api documentation is insufficient for developers: an empirical study. **Science China. Information Sciences**, Springer Nature BV, v. 64, n. 1, p. 119102, 2021. Citado na página 17.

FAZZINI, M.; XIN, Q.; ORSO, A. Atualização automatizada de uso de api para aplicativos android. In: **Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis**. New York, NY, USA: Association for Computing Machinery, 2019. (ISSTA 2019), p. 204–215. ISBN 9781450362245. Disponível em: <https://doi.org/10.1145/3293882.3330571>. Citado na página 21.

FERNANDO, S. et al. Leveraging large language models for the automated documentation of hardware designs. In: **2024 13th Mediterranean Conference on Embedded Computing (MECO)**. [S.l.: s.n.], 2024. p. 1–6. Citado na página 23.

JORELLE, Y. Geração de documentação de api usando grandes modelos de linguagem: rumo a apis autoexplicativas. 2024. Citado na página 22.

KAMBHAMPATI, S. et al. **LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks**. 2024. Disponível em: <https://arxiv.org/abs/2402.01817>. Citado na página 14.

KIM, J. et al. Adding examples into java documents. In: **2009 IEEE/ACM International Conference on Automated Software Engineering**. [S.l.: s.n.], 2009. p. 540–544. Citado 2 vezes nas páginas 15 e 20.

KRAUSE, J. **Introducing Web Development**. 1. ed. Berkeley, CA: Apress, 2016. ISBN 978-1-4842-2499-1. Citado 2 vezes nas páginas 18 e 19.

LIN, F. et al. **SOEN-101: Code Generation by Emulating Software Process Models Using Large Language Model Agents**. 2024. Disponível em: <https://arxiv.org/abs/2403.15852>. Citado na página 14.

MAURER, P. G. G.; ZAMBERLAN, A. d. O. Trabalho de Conclusão de Curso em Sistemas de Informação, **Protótipo de Interação Humano-Computador para Processamento da Língua Natural em LLMs**. Santa Maria, RS, Brasil: [s.n.], 2024. <https://tfgonline.lapinf.ufn.edu.br>. Citado na página 18.

MIKKONEN, T.; TAIVALSAARI, A. **Using JavaScript as a real programming language**. [S.l.]: Citeseer, 2007. Citado na página 19.

PROCKO, T. T.; COLLINS, S. Automatic code documentation with syntax trees and gpt. 2023. Citado na página 22.

RACHMAT, N.; KESUMA, D. P. Implementasi llm gemini pada pengembangan aplikasi chatbot berbasis android. **Jurnal Ilmu Komputer (JUIK)**, v. 4, n. 1, p. 40–52, 2024. Citado na página 18.

ROCHA, A. M.; MAIA, M. A. Automated api documentation with tutorials generated from stack overflow. In: **Proceedings of the XXX Brazilian Symposium on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2016. (SBES '16), p. 33–42. ISBN 9781450342018. Disponível em: <https://doi.org/10.1145/2973839.2973847>. Citado 3 vezes nas páginas 14, 21 e 39.

SOUZA, L. de; CAMPOS, E.; MAIA, M. On the extraction of cookbooks for apis from the crowd knowledge. In: **Proceedings of the 28th Brazilian Symposium on Software Engineering (SBES 2014)**. [S.l.: s.n.], 2014. p. 21–30. Citado 2 vezes nas páginas 14 e 20.

SRINATH, K. Python–the fastest growing programming language. **International Research Journal of Engineering and Technology**, v. 4, n. 12, p. 354–357, 2017. Citado na página 19.

STYLOS, J. et al. Improving api documentation using api usage information. In: . [S.l.: s.n.], 2009. p. 119–126. Citado na página 20.

SUBRAMANIAN, S.; INOZEMTSEVA, L.; HOLMES, R. Live api documentation. In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 643–652. ISBN 9781450327565. Disponível em: <https://doi.org/10.1145/2568225.2568313>. Citado na página 14.

WANG, S.; TIAN, Y.; HE, D. gdoc: Automatic generation of structured api documentation. In: **Proceedings of the ACM Web Conference 2023 (WWW '23 Companion)**. Association for Computing Machinery, 2023. p. 53–56. ISBN 9781450394192. Disponível em: <https://doi.org/10.1145/3543873.3587310>. Citado na página 22.