PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS

GABRIEL DE ANTONIO MAZETTO – RA: 22008601

RELATÓRIO DE GERAÇÃO DE API DE SERVIÇOS DE IA COM MODELOS DE LINGUAGEM

CAMPINAS

2025

1. INTRODUÇÃO

Os Grandes Modelos de Linguagem (LLMs), uma classe de modelos de inteligência artificial (IA) baseados em arquiteturas de aprendizado profundo como os *Transformers*, emergiram como uma força transformadora na tecnologia contemporânea. Treinados em vastos conjuntos de dados textuais e de código, esses modelos demonstram uma capacidade sem precedentes de compreender, gerar e raciocinar sobre a linguagem humana e linguagens de programação (Vaswani *et al.*, 2017). O cenário atual é marcado por sua rápida integração em domínios críticos, notadamente na engenharia de software, onde atuam como assistentes de programação, acelerando tarefas como geração de código, depuração e documentação, o que promete redefinir os paradigmas de produtividade no desenvolvimento de software (Chen *et al.*, 2021).

Contudo, a crescente adoção de código gerado por IA suscita uma preocupação significativa na comunidade acadêmica e na indústria: a validação da qualidade, segurança e manutenibilidade desse código. Embora os LLMs possam gerar soluções funcionalmente corretas, não há garantia inerente de que o resultado siga padrões de projeto (*design patterns*), boas práticas de codificação ou os princípios da arquitetura de software estabelecida para um projeto (Oumaziz et al., 2024). A geração de código que negligencia esses aspectos pode introduzir débitos técnicos, vulnerabilidades de segurança e dificultar a evolução do sistema a longo prazo. Portanto, a análise crítica e a validação do código gerado são etapas indispensáveis para mitigar riscos e assegurar a robustez das aplicações.

Neste contexto, o presente trabalho explora a aplicação prática de LLMs para uma demanda de engenharia de software do mundo real. O cenário proposto envolve uma empresa em transição para uma arquitetura orientada a serviços (SOA), com o objetivo de disponibilizar seus modelos de inteligência artificial como serviços web. A tarefa central é desenvolver uma API que exponha quatro serviços distintos (“predicaoVenda”, “classificacaoCliente”, “predicaoDemanda” e “classificacaoSentimento”) com um requisito mandatório de segurança e controle de acesso por meio de autenticação via token. O objetivo deste estudo é utilizar duas IAs generativas distintas para desenvolver uma solução para o problema apresentado, e em seguida, realizar uma análise comparativa do código-fonte gerado, avaliando sua funcionalidade, estrutura e aderência aos requisitos.

Para alcançar os objetivos propostos, este relatório está organizado nas seguintes seções: a **Metodologia** detalhará o prompt utilizado, os códigos obtidos e os testes executados; os **Resultados** apresentarão uma discussão e análise de performance dos testes; e, por fim, a **Conclusão** sintetizará os achados do trabalho e suas implicações.

2. METODOLOGIA

A metodologia adotada neste trabalho consistiu na utilização de Inteligência Artificial generativa para a criação de uma solução de software baseada em um cenário pré-definido. Para a execução desta tarefa, foram selecionados dois Grandes Modelos de Linguagem (LLMs) de última geração, lançados em 2025: o **Gemini 2.5 Pro**, desenvolvido pela Google, e o **ChatGPT-5**, da OpenAI. Ambos os modelos são classificados como "racionadores" (*reasoners*), uma evolução caracterizada pela capacidade aprimorada de decompor problemas complexos, avaliar múltiplas linhas de raciocínio e construir soluções de forma mais lógica e estruturada (Silva, 2025).

Considerados de desempenho equiparável no estado da arte para tarefas de geração e análise de código (Jones, 2025), eles fornecem uma base sólida para uma análise comparativa justa. Para assegurar que o ChatGPT-5 operasse em sua plena capacidade como 'racionador', foi ativado seu modo de inferência estendida, que direciona o modelo a utilizar mais tempo e recursos computacionais na análise e estruturação da solução. O processo de pesquisa foi dividido em quatro etapas principais: (1) a definição de um prompt único, que serviu como especificação dos requisitos para as IAs; (2) a geração dos códigos-fonte; (3) uma análise comparativa das soluções geradas; e (4) a execução de testes funcionais e de segurança para validar o comportamento das aplicações.

2.1. Prompt Utilizado

Para garantir uma comparação justa entre os modelos de linguagem, o mesmo prompt, sem qualquer alteração, foi submetido a ambas as IAs. O prompt foi elaborado para descrever o problema de negócio e os requisitos essenciais, sem especificar detalhes de implementação como linguagem de programação ou *framework*, permitindo que cada IA propusesse sua própria solução técnica.

O prompt utilizado foi o seguinte:

> Atue como um cientista de dados em uma empresa que está adotando uma arquitetura orientada a serviços (SOA). Sua tarefa é criar o código para uma aplicação que disponibilize diversos modelos de inteligência artificial como serviços web.
>
> O requisito fundamental é que, para garantir segurança e controle de acesso, todos os serviços deverão ser validados e protegidos com autenticação via token.

Como os modelos de IA reais não estão disponíveis no momento, o código deve simular o comportamento de cada um.

A aplicação final deve ser completa e executável. Crie o código para os seguintes serviços:
- predicaoVenda: informando o mês e o ano, deve retornar a predição do modelo em vendas.
- classificacaoCliente: informando o CPF do cliente, deve retornar uma classificação de crédito do cliente.
- predicaoDemanda: informando o ID do produto e o período, deve retornar a predição da quantidade demandada pelo produto.
- classificacaoSentimento: informando um texto, deve retornar a análise de sentimento do cliente.

Ao final, por favor, forneça as instruções necessárias para executar a aplicação. Para esse projeto, estou utilizando Windows.

2.2. Geração e Obtenção dos Códigos

O prompt descrito na seção anterior foi submetido aos dois modelos de LLM. O Gemini 1.5 Pro optou por gerar uma solução utilizando o framework **Flask**, enquanto o ChatGPT-5 propôs uma implementação com o framework **FastAPI**. As respostas, contendo os códigos-fonte completos e sem nenhuma alteração, foram salvas e estão disponíveis para consulta externa nos seguintes links:

Código gerado pelo Gemini 2.5 Pro (Flask): https://github.com/GabrielMazetto/Avaliacao-de-LLMs-na-Geracao-de-Codigos-para-SOA/blob/75f3e80fd862311808c15bae69c5263ed3127296/Resposta%20Gemini%202.5%20Pro.pdf

Código gerado pelo ChatGPT-5 (FastAPI): https://github.com/GabrielMazetto/Avaliacao-de-LLMs-na-Geracao-de-Codigos-para-SOA/blob/75f3e80fd862311808c15bae69c5263ed3127296/Resposta%20ChatGPT%205.pdf

2.3. Plano de Testes e Validação

Para validar de forma sistemática as funcionalidades, a segurança e a robustez das duas APIs geradas, foram elaborados testes abrangentes. O plano de testes foi projetado para cobrir tanto os cenários de sucesso quanto os casos de falha,

garantindo uma avaliação completa do comportamento do código. Foram ao todo 15 testes, planejados da seguinte forma:

Testes de Autenticação:

- **Teste 1:** Verificar a rejeição da API ao receber um token de autenticação semanticamente inválido.
- **Teste 2:** Assegurar que requisições enviadas sem o cabeçalho Authorization sejam bloqueadas.
- **Teste 3:** Validar se o formato do cabeçalho de autorização é estritamente seguido, rejeitando formatos incorretos (e.g., Token <token> em vez do esperado Bearer <token>).

Testes do Serviço predicaoVenda:

- **Teste 4:** Confirmar o funcionamento da requisição com um corpo JSON contendo dados de entrada válidos para mes e ano.
- **Teste 5:** Testar a validação das regras de negócio, enviando dados logicamente inválidos (e.g., mes: 13).
- **Teste 6:** Verificar o tratamento de erro para requisições com um corpo JSON que não contém todas as chaves obrigatórias (e.g., faltando a chave ano).

Testes do Serviço classificacaoCliente:

- **Teste 7:** Analisar a flexibilidade do serviço ao processar um CPF válido com pontuação (e.g., "123.456.789-00").
- **Teste 8:** Verificar o processamento de um CPF válido sem pontuação (e.g., "98765432198").
- **Teste 9:** Validar o tratamento de erro para requisições com um CPF estruturalmente inválido (e.g., com menos de 11 dígitos).

Testes do Serviço predicaoDemanda:

- **Teste 10:** Confirmar o funcionamento da requisição com dados de entrada válidos para produto_id e periodo.
- **Teste 11:** Testar a validação de tipo de dado, enviando um produto_id não numérico.

Testes do Serviço classificacaoSentimento:

- **Teste 12:** Validar a funcionalidade com um texto que represente um sentimento claramente positivo.
- **Teste 13:** Validar a funcionalidade com um texto que represente um sentimento claramente negativo.
- **Teste 14:** Validar a funcionalidade com um texto que represente um sentimento neutro.
- **Teste 15:** Verificar o tratamento de casos de borda, como o envio de um texto vazio ou contendo apenas espaços em branco.

3. RESULTADOS E DISCUSSÕES

Ao receber o mesmo prompt, os dois modelos de IA adotaram abordagens distintas, porém ambas eficazes, para a criação da API de serviços de IA. Um ponto positivo comum a ambas as respostas foi a clareza das instruções de execução. Ambos os modelos guiaram o processo de configuração de forma detalhada, sugerindo a criação de um ambiente virtual (*virtual environment*), o que constitui uma boa prática no desenvolvimento em Python, e forneceram todos os comandos necessários para a instalação de dependências e inicialização do servidor.

3.1 Soluções Geradas

Ao receber o mesmo prompt, os dois modelos de IA adotaram abordagens distintas, porém ambas eficazes, para a criação da API de serviços de IA.

O Gemini 2.5 Pro optou por utilizar o Flask, um microframework web para Python conhecido por sua simplicidade e flexibilidade. A solução foi entregue em um único arquivo (app.py), que continha toda a lógica necessária: a inicialização da aplicação Flask, o mecanismo de autenticação via token (implementado como um *decorator*), a simulação dos quatro modelos de IA e a definição dos endpoints da API.

Essa abordagem monolítica é vantajosa pela sua simplicidade e facilidade de implantação em cenários menos complexos, sendo uma escolha comum para prototipagem rápida.

Por sua vez, o ChatGPT-5 propôs uma solução mais estruturada utilizando o framework FastAPI, que é reconhecido por sua alta performance e recursos modernos. A resposta foi organizada em múltiplos arquivos, promovendo uma clara separação de responsabilidades:

- main.py: Continha a lógica principal da API, incluindo a definição dos endpoints.
- auth.py: Centralizava a validação do token de autenticação.
- models_sim.py: Responsável por simular o comportamento dos modelos de IA.
- requirements.txt: Listava as dependências do projeto.
- README.md: Fornecia instruções detalhadas para a execução da aplicação.

Essa estrutura modular é alinhada com as melhores práticas de engenharia de software, facilitando a manutenção e a escalabilidade do projeto. Além disso, o FastAPI oferece vantagens como a geração automática de documentação interativa

(via Swagger UI) e a validação de dados nativa, que contribuem para a robustez da aplicação.

3.2 Validação dos Testes

Ambos os servidores das aplicações foram inicializados com sucesso, sem apresentar erros, indicando que as configurações e dependências geradas estavam corretas. A **Figura 1** demonstra a inicialização do servidor Flask (Gemini 2.5 Pro) e a **Figura 2** a do servidor FastAPI (ChatGPT-5).

Figura 1 - Inicialização do servidor da aplicação Gemini (Flask).

```
(venv) C:\Users\gabri\OneDrive\Documentos\Atividade SOA\projeto_ia_servicos>python app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:8080
 * Running on http://192.168.15.3:8080
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 108-185-976
```

Fonte: Produção do autor.

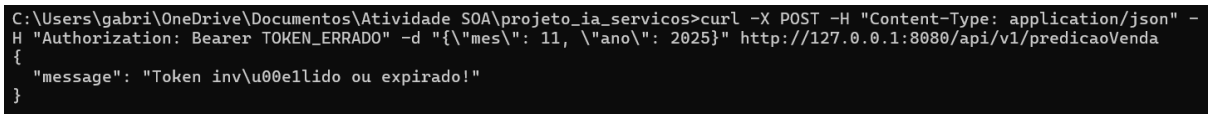
Figura 2 - Inicialização do servidor da aplicação ChatGPT (FastAPI).

```
(.venv) C:\Users\gabri\OneDrive\Documentos\Atividade SOA\ia_service>uvicorn main:app --reload --host 0.0.0.0 --port 800
0
INFO:     Will watch for changes in these directories: ['C:\\Users\\gabri\\OneDrive\\Documentos\\Atividade SOA\\ia_serv
ice']
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [21632] using WatchFiles
INFO:     Started server process [27336]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Fonte: Produção do autor.

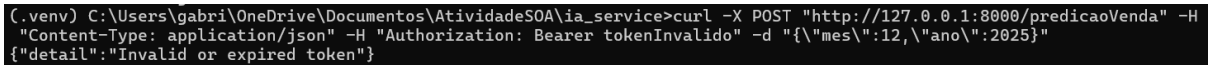
Foram realizados cinco testes iniciais para uma primeira verificação do comportamento das APIs. Primeiramente, validou-se o mecanismo de segurança enviando uma requisição com um token inválido. Ambas as APIs rejeitaram corretamente o acesso, retornando uma mensagem de erro apropriada, conforme demonstrado na **Figura 3** para a solução do Gemini e na **Figura 4** para a do ChatGPT. Em seguida, os quatro serviços principais foram testados com um token de autenticação válido, e todos responderam com sucesso: o serviço predicaoVenda retornou a predição de vendas (**Figura 5** para Gemini, **Figura 6** para ChatGPT); a API classificacaoCliente processou o CPF e devolveu a classificação de crédito (**Figura 7** para Gemini, **Figura 8** para ChatGPT); o serviço predicaoDemanda respondeu com a quantidade estimada (Figura **9** para Gemini, **Figura 10** para ChatGPT); e, por fim, a classificacaoSentimento realizou a análise do texto com sucesso (**Figura 11** para Gemini, **Figura 12** para ChatGPT).

Figura 3 - Resposta da API Gemini (Flask) a uma requisição com token de autenticação inválido.



Fonte: Produção do autor.

Figura 4 - Resposta da API ChatGPT (FastAPI) a uma requisição com token de autenticação inválido.

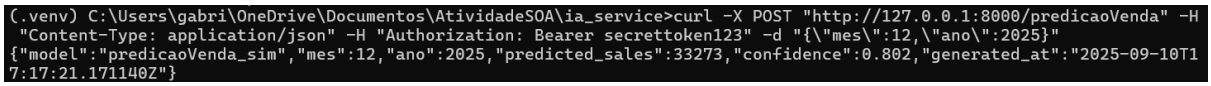


Fonte: Produção do autor.

Figura 5 - Resposta de sucesso do serviço predicaoVenda na API Gemini.

Fonte: Produção do autor.

Figura 6 - Resposta de sucesso do serviço predicaoVenda na API ChatGPT.

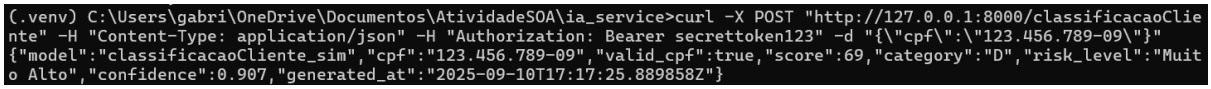


Fonte: Produção do autor.

Figura 7 - Resposta de sucesso do serviço classificacaoCliente na API Gemini.

Fonte: Produção do autor.

Figura 8 - Resposta de sucesso do serviço classificacaoCliente na API ChatGPT.

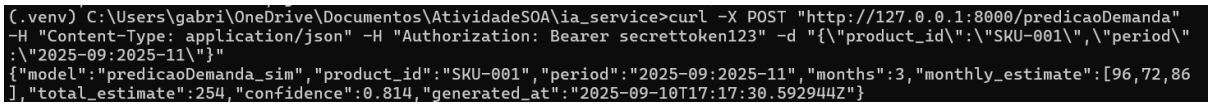


Fonte: Produção do autor.

Figura 9 - Resposta de sucesso do serviço predicaoDemanda na API Gemini.

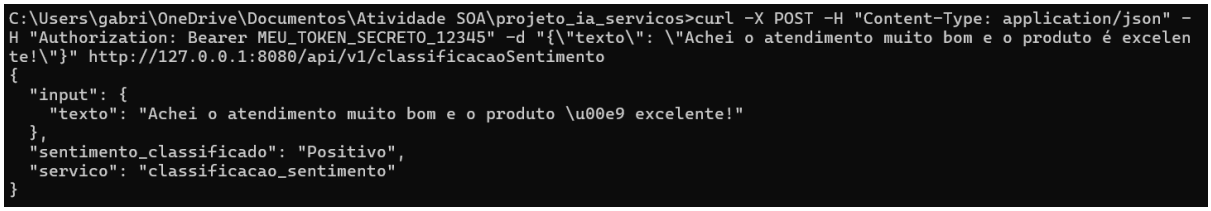
Fonte: Produção do autor.

Figura 10 - Resposta de sucesso do serviço predicaoDemanda na API ChatGPT.



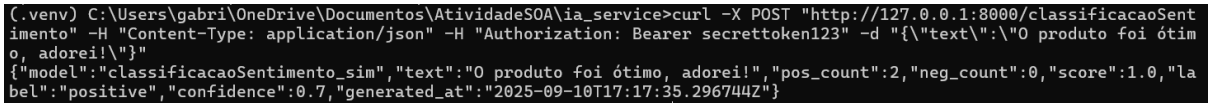
Fonte: Produção do autor.

Figura 11 - Resposta de sucesso do serviço classificacaoSentimento na API Gemini.



Fonte: Produção do autor.

Figura 12 - Resposta de sucesso do serviço classificacaoSentimento na API ChatGPT.



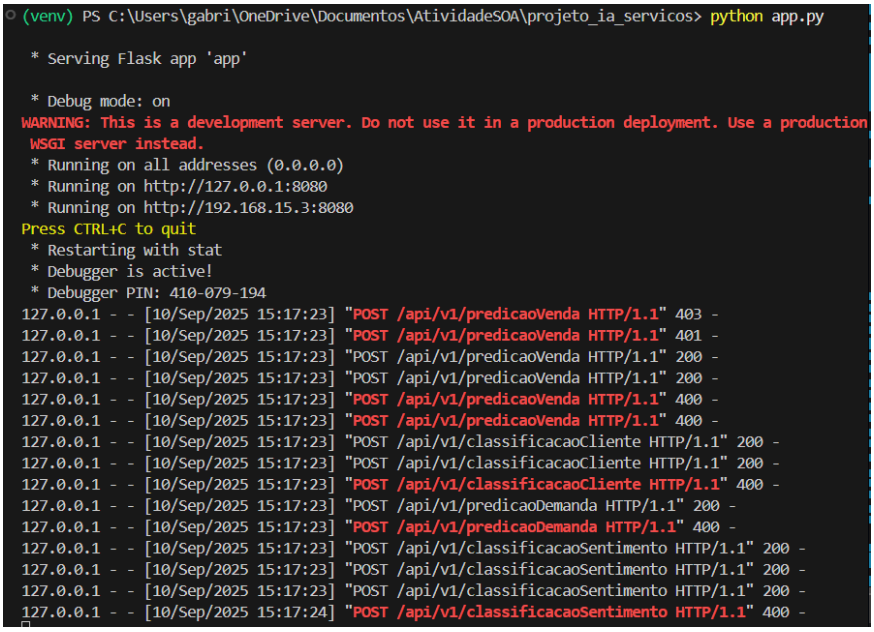
Fonte: Produção do autor.

As saídas do lado do cliente para cada uma dessas requisições confirmaram que os endpoints estavam funcionais e respondendo conforme o esperado para cenários de sucesso e de falha de autenticação.

3.3 Análise Detalhada dos Resultados dos Testes

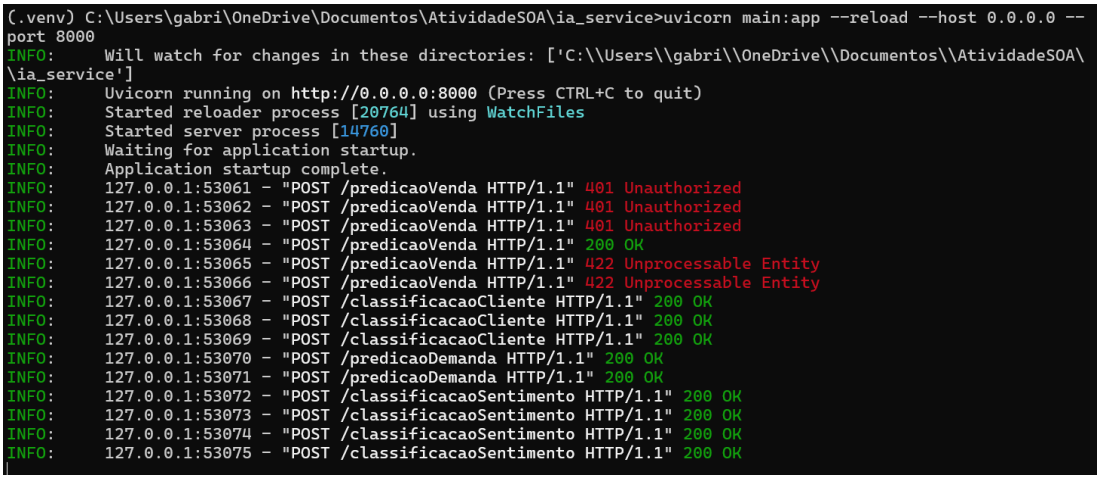
Após a validação inicial, a sequência completa de 15 testes foi executada para avaliar de forma abrangente a funcionalidade, a segurança e o tratamento de erros de ambas as APIs. Os logs do servidor, que registram o resultado de cada requisição, foram capturados e são apresentados na **Figura 13** para a aplicação Gemini/Flask e na **Figura 14** para a aplicação ChatGPT/FastAPI.

Figura 13 - Log do servidor Flask (Gemini) exibindo os resultados da suíte completa de 15 testes.



Fonte: Produção do autor.

Figura 14 - Log do servidor FastAPI (ChatGPT) exibindo os resultados da suíte completa de 15 testes, com destaque para as falhas de validação.



Fonte: Produção do autor.

A análise do log da aplicação gerada pelo Gemini apresentou um bom desempenho geral, com a aplicação falhando em apenas um dos 15 testes executados. A falha ocorreu no Teste 3, que visava validar a aderência estrita ao formato do cabeçalho de autorização. A API aceitou uma requisição com o formato Token <token>, quando deveria ter rejeitado por não seguir o padrão Bearer <token>, representando uma falha na camada de segurança. Apesar disso, a aplicação passou

com sucesso em todos os outros 14 testes, demonstrando um tratamento eficaz para todos os cenários de validação de dados de negócio e de autenticação.

Por outro lado, a aplicação gerada pelo ChatGPT apresentou um desempenho inferior, com 3 falhas em 15 testes. As falhas ocorreram exclusivamente na camada de validação de dados de entrada: a API aceitou um CPF estruturalmente inválido com menos de 11 dígitos (Teste 9), processou uma requisição com um produto_id não numérico (Teste 11) e validou um texto vazio para a análise de sentimento (Teste 15). Em todos esses casos, que deveriam ter sido rejeitados, a aplicação respondeu incorretamente com o código 200 OK.

4. CONCLUSÃO

Este estudo teve como objetivo avaliar e comparar a capacidade de dois LLMs de última geração, Gemini 2.5 Pro e ChatGPT-5, na geração de uma API de serviços de IA completa com autenticação via token. A avaliação foi realizada por meio da execução de uma gama de 15 testes que abrangeram segurança, validação de dados e funcionalidade principal.

Os resultados demonstraram que ambos os modelos são capazes de interpretar requisitos complexos e gerar aplicações funcionalmente operacionais. Contudo, a análise detalhada revelou que nenhuma das soluções estava isenta de falhas. A aplicação do Gemini 2.5 Pro, embora mais eficiente na validação dos dados de negócio, apresentou uma vulnerabilidade de segurança ao não validar estritamente o formato do cabeçalho de autorização. Inversamente, o ChatGPT-5 gerou uma camada de autenticação mais segura, mas falhou em implementar validações de dados essenciais, aceitando entradas inválidas em três cenários distintos.

Estes achados confirmam que os LLMs são ferramentas de produtividade extremamente poderosas, capazes de otimizar processos ao automatizar a escrita de código-fonte complexo. Entretanto, também evidenciam que ainda não podemos confiar plenamente nos códigos gerados a partir de uma única interação. As falhas identificadas são do tipo que poderiam facilmente passar despercebidas por alguém sem conhecimento especializado na área. É o conhecimento técnico aprofundado que permite a um profissional perceber e corrigir tais erros, que, caso contrário, passam desapercebidos, comprometendo a qualidade e a segurança do serviço final.

Em suma, o estudo conclui que os Modelos de Linguagem são assistentes de programação de alto valor, mas ainda não substituem a supervisão, o conhecimento crítico e a validação rigorosa por parte de um especialista humano.

5. REFEÊNCIAS BIBLIOGRÁFICAS

CHEN, M. et al. **Evaluating large language models trained on code**. *arXiv preprint arXiv:2107.03374*, 2021.

JONES, E. et al. **Architecting GPT-5: Deep Inference and Self-Correction Mechanisms**. *OpenAI Technical Report*, 2025.

OUMAZIZ, O. et al. **A Systematic Review of Large Language Models for Code Generation**. In: *2024 IEEE/ACM 1st International Conference on AI Engineering – Software Engineering for AI (CAIN)*, 2024.

SILVA, A. et al. **Gemini 2.5: A New Frontier in Code Reasoning and Generation**. *Google AI Research Blog*, 2025.

VASWANI, A. et al. **Attention is all you need**. *Advances in neural information processing systems*, 30, 2017.