BOCA
BOLETIM DE CONJUNTURA

O USO DAS LLM'S PARA OTIMIZAR O DESENVOLVIMENTO DE APLICAÇÕES WEB CONSTRUÍDAS EM DJANGO: DESENVOLVENDO ANÁLISE COMPARATIVA ENTRE LLM'S NA IDENTIFICAÇÃO DE BUGS E QUALIDADE DE RESPOSTA

THE USE OF LLMs TO OPTIMIZE THE DEVELOPMENT OF WEB APPLICATIONS BUILT WITH DJANGO: DEVELOPING A COMPARATIVE ANALYSIS BETWEEN LLMs IN BUG IDENTIFICATION AND RESPONSE QUALITY

EL USO DE LLM PARA OPTIMIZAR EL DESARROLLO DE APLICACIONES WEB CREADAS CON DJANGO: DESARROLLO DE UN ANÁLISIS COMPARATIVO ENTRE LLM EN LA IDENTIFICACIÓN DE ERRORES Y LA CALIDAD DE LA RESPUESTA

10.56238/bocav25n78-032

Cristian Abreu Berredo
Graduando em Sistemas de Informação
Instituição: Centro Universitário Santa Terezinha (CEST)

Gilbert Correia Fernandes
Graduando em Sistemas de Informação
Instituição: Centro Universitário Santa Terezinha (CEST)

Dadilton Bastos Melo
Orientador do Curso de Sistemas de Informação
Instituição: Centro Universitário Santa Terezinha (CEST)

RESUMO
O desenvolvimento de aplicações web no framework Django, apesar de sua estrutura robusta e alta performance, ainda enfrenta desafios relacionados à identificação de bugs e problemas de manutenção de código. Com o avanço da inteligência artificial, os modelos de linguagem de grande escala (LLMs) surgem como uma alternativa promissora para otimizar tarefas como geração de código, revisão e correção de falhas. Este artigo tem como objetivo avaliar como as LLMs podem auxiliar no desenvolvimento de software construídos no framework Django Python. Para isso, foi desenvolvida uma pesquisa de natureza quantitativa, na qual duas LLMs foram submetidas a problemas relacionados a bugs e melhorias em um protótipo de sistema de cadastro de produtos desenvolvido em Django. As variáveis testáveis consideradas foram: tempo de resposta, número de erros na resposta, qualidade da resposta (classificação de 1 a 5), clareza da explicação e justificativa da decisão da resposta. Os resultados foram organizados em uma tabela comparativa, permitindo analisar o desempenho, a eficiência e a qualidade das respostas geradas por cada modelo. A pesquisa foi conduzida no ambiente Visual Studio Code com Python e Django, respeitando as licenças de software e a integridade científica. Os dados obtidos indicam diferenças significativas entre os modelos testados, contribuindo para uma escolha mais criteriosa de LLMs no ciclo de desenvolvimento de software com Django.

Palavras-chave: LLM'S. Programação. Django. Python.

ABSTRACT
The development of web applications using the Django framework, despite its robust structure and high performance, still faces challenges related to bug identification and code maintenance. With the advancement of artificial intelligence, Large Language Models (LLMs) have emerged as a promising alternative for optimizing tasks such as code generation, review, and bug fixing. This article aims to evaluate how LLMs can assist in software development using the Django Python framework. To achieve this, a quantitative research study was conducted in which two LLMs were subjected to problems related to bugs and improvements in a prototype product registration system developed in Django. The variables analyzed were response time, number of errors in the response, response quality (rated from 1 to 5), clarity of explanation, and justification for the response decision. The results were organized into a comparative table, enabling the analysis of the performance, efficiency, and quality of the responses generated by each model. The research was conducted in the Visual Studio Code environment using Python and Django, while respecting software licenses and scientific integrity. The obtained data indicate significant differences between the tested models, contributing to a more careful selection of LLMs in the Django software development cycle.

Keywords: LLMS'S. Programming. Django. Python.

RESUMEN
El desarrollo de aplicaciones web con el framework Django, a pesar de su robusta estructura y alto rendimiento, aún presenta desafíos relacionados con la identificación de errores y el mantenimiento del código. Con el avance de la inteligencia artificial, los modelos de lenguaje a gran escala (LLM) se presentan como una alternativa prometedora para optimizar tareas como la generación, revisión y corrección de errores del código. Este artículo tiene como objetivo evaluar cómo los LLM pueden contribuir al desarrollo de software basado en el framework Django de Python. Para ello, se realizó un estudio cuantitativo en el que dos LLM fueron sometidos a problemas relacionados con errores y mejoras en un prototipo de sistema de registro de productos desarrollado en Django. Las variables evaluables fueron: tiempo de respuesta, número de errores en la respuesta, calidad de la respuesta (calificación de 1 a 5), claridad de la explicación y justificación de la decisión de respuesta. Los resultados se organizaron en una tabla comparativa, lo que permitió analizar el rendimiento, la eficiencia y la calidad de las respuestas generadas por cada modelo. La investigación se llevó a cabo en el entorno Visual Studio Code con Python y Django, respetando las licencias de software y la integridad científica. Los datos obtenidos indican diferencias significativas entre los modelos probados, lo que contribuye a una selección más cuidadosa de los modelos de lógica de negocio (MLG) en el ciclo de desarrollo de software con Django.

Palabras clave: MLD. Programación. Django. Python.

1 INTRODUÇÃO

O desenvolvimento de aplicações web exige soluções eficientes, seguras e inteligentes para garantir qualidade, desempenho e facilidade de manutenção dos sistemas. O Django, um framework consolidado da linguagem Python, destaca-se pela sua estrutura bem definida e alta performance, permitindo a criação de aplicações robustas. No entanto, apesar dessas vantagens, a identificação de bugs e problemas de performance ainda representa um grande desafio para os desenvolvedores.

Com o avanço da inteligência artificial, os modelos de linguagem de grande escala (LLMs) surgem como uma estratégia promissora para otimizar o desenvolvimento de software. De acordo com o Forbes Technology Council, a detecção de defeitos em fases avançadas pode ser 100 vezes mais cara do que encontrar bugs no início do desenvolvimento. Estima-se que, em 2002, bugs de software tenham causado um prejuízo de US 59,5 bilhões à economia dos Estados Unidos (PRESSMAN, 2016). Já em 2022, um relatório do Consortium for Information & Software Quality (CISQ) apontou que a má qualidade de software custou US 2,41 trilhões à economia americana (CISQ, 2022).

As Large Language Models, como o GPT da OpenAI, revolucionaram a forma como a linguagem natural é interpretada e gerada por sistemas computacionais. Essas redes neurais, treinadas com bilhões de parâmetros, são capazes de compreender comandos em linguagem natural, produzir trechos de código, revisar padrões lógicos e até sugerir refatorações para trechos de software. Segundo Sá (2024), as LLMs podem ser utilizadas como suporte à correção de bugs, reduzindo o tempo necessário para identificar falhas no código.

A integração de LLMs com aplicações Django oferece uma nova abordagem para antecipar esses desafios. Silva (2024) destaca que assistentes baseados em LLMs têm sido utilizados com êxito tanto para escrever trechos de código quanto para sugerir soluções durante a construção de sistemas. Com isso, tarefas como criação de rotas, views, serializers, validações e até ORM queries podem ser auxiliadas por LLMs. Além disso, há aplicações na identificação de padrões problemáticos no código, especialmente no desacoplamento de lógica de negócios e interface. O uso preditivo de LLMs nesse contexto permite antecipar falhas que, se não corrigidas precocemente, geram alto custo em fases posteriores. Assim, conforme ressalta Silva (2024), essa sinergia entre LLM e Django representa uma ferramenta estratégica para elevar a qualidade do software e a produtividade das equipes.

Diante desse contexto, este artigo tem como objetivo geral avaliar como as LLMs podem auxiliar no desenvolvimento de software construídos no framework Django Python. Para alcançar essa proposta, foram estabelecidos objetivos específicos que orientam a condução da pesquisa: comparar o tempo de resposta entre duas LLMs distintas; analisar e comparar a forma como cada modelo resolve os problemas propostos e a clareza da explicação de sua decisão de resposta; e, por fim, comparar a qualidade das respostas geradas e a capacidade de compreensão do problema proposto por cada uma das duas LLMs analisadas.

2 REFERENCIAL TEÓRICO

O referencial teórico deste artigo tem como propósito fornecer os fundamentos conceituais necessários à compreensão da pesquisa, organizando-se em quatro subseções: a subseção 2.1 apresenta os princípios de funcionamento das Large Language Models (LLMs) e sua arquitetura baseada em transformadores; a subseção 2.2 descreve as características do framework Django e seu ecossistema para desenvolvimento web em Python; a subseção 2.3 discute a aplicação de ferramentas baseadas em LLMs no auxílio à codificação e revisão de software; e a subseção 2.4 aborda o papel da inteligência artificial na automatização de testes de software.

2.1 FUNDAMENTAÇÃO SOBRE LLM'S E INTELIGÊNCIA ARTIFICIAL

As LLMs (Large Language Models) são modelos de inteligência artificial treinados em enormes volumes de texto para entender, gerar e processar linguagem natural. Eles funcionam prevendo a próxima palavra em uma sequência com base no contexto anterior, permitindo tarefas complexas de linguagem. Esses modelos processam texto dividindo-o em tokens, que são unidades menores como palavras ou caracteres. Cada token é convertido em números e depois em vetores que capturam seu significado. O modelo analisa esses vetores usando redes transformadoras para entender o contexto e prever os próximos tokens, gerando a saída final, que é convertida de volta em texto.

Os LLMs são um tipo de modelo de aprendizado de máquina treinado em uma vasta quantidade de dados textuais. Esses modelos são capazes de compreender, gerar e manipular texto de maneira sofisticada, graças a milhões (ou até bilhões) de parâmetros que ajustam suas previsões com base nos dados de treinamento. Os LLMs são usados em uma variedade de aplicações, desde assistentes virtuais até a criação de conteúdos automatizados. Parâmetros em uma LLM são os valores ajustáveis que o modelo aprende durante o treinamento, determinando como ele processa e gera textos (MELO, Guilherme LM et al., s.d.)

Os modelos de linguagem representam um marco significativo no avanço da inteligência artificial aplicada ao processamento de linguagem natural. Essas arquiteturas, treinadas em grandes volumes de dados, são capazes de compreender, gerar e adaptar texto de maneira contextualizada, aproximando-se cada vez mais da comunicação humana.

2.2 DESENVOLVIMENTO DE APLICAÇÕES DE SOFTWARE CONSTRUÍDAS EM DJANGO PYTHON

O Django é um framework de desenvolvimento web de alto nível, escrito em Python, que promove o desenvolvimento rápido e limpo de aplicações web. Conforme cita a Django Software Foundation (2025, n.p.): "Incentiva o desenvolvimento rápido e o design limpo e pragmático. Criado por desenvolvedores experientes, ele cuida de grande parte do incômodo do desenvolvimento web,

para que você possa se concentrar em escrever seu aplicativo sem precisar reinventar a roda. É gratuito e de código aberto."

Como o Django foi desenvolvido em um ambiente de redação em ritmo acelerado, foi projetado para tornar as tarefas comuns de desenvolvimento web rápidas e fáceis (DJANGO SOFTWARE FOUNDATION, 2025, n.p.).

Portanto, o framework Django se consolidou como um dos frameworks mais completos e robustos para o desenvolvimento web em Python, oferecendo uma ampla gama de recursos prontos para uso, o que acelera a criação de aplicações seguras, escaláveis e de alta qualidade.

2.3 O USO DE FERRAMENTAS BASEADAS EM LLMS PARA AUXÍLIO NO DESENVOLVIMENTO DE SOFTWARE

As LLMs podem ser aplicadas em diversas áreas para criar soluções inteligentes baseadas em linguagem natural. Em atendimento ao cliente, elas alimentam chatbots e assistentes virtuais capazes de responder dúvidas e resolver problemas de forma automática. Em desenvolvimento de software, auxiliam na geração de código, revisão, documentação e testes, acelerando o trabalho dos programadores (SILVA, Lucas Marques, 2024). Já em negócios, são usadas para análise de dados, geração de relatórios, criação de conteúdo de marketing e automação de tarefas administrativas.

Esses modelos de linguagem também têm grande impacto em educação, atuando como tutores inteligentes, resumos de textos e criação de materiais de apoio. Em aplicações criativas, podem gerar textos, roteiros, histórias e até auxiliar em tradução e revisão. Dessa forma, as LLMs funcionam como um núcleo de inteligência em diferentes aplicações, trazendo produtividade, automação e inovação para múltiplos setores.

Existem ferramentas que utilizam LLMs como núcleo para oferecer funcionalidades práticas em diferentes contextos. Segundo Silva, Lucas Marques (2024, p. 19): "Ferramentas como Copilot, CodeLlama, e Tabnine estão se tornando extremamente sofisticadas nas tarefas de analisar códigos, encontrar padrões, compreender contexto, e oferecer ajuda instantânea, portanto elevando a produtividade e qualidade do software." O autor complementa que algoritmos de IA conseguem antecipar próximos segmentos de código, oferecendo assinaturas de métodos e identificando potenciais problemas, habilitando desenvolvedores a gerar código mais idiomático, ou seja, conciso, fácil de verificar e fácil de expandir.

As ferramentas baseadas em LLMs vêm transformando a forma como lidamos com informações, automatizamos tarefas e potencializamos a criatividade. Essas soluções permitem desde a geração de texto, código e resumos até análise de grandes volumes de dados de maneira inteligente. Plataformas de atendimento automatizado, sistemas de apoio e agentes de produtividade são alguns exemplos prátricos desse impacto.

2.4 O USO DA IA NO AUXÍLIO A TESTES DE APLICAÇÕES DE SOFTWARE

As LLMs podem ser usadas em testes de software para automatizar e otimizar várias etapas. Elas conseguem gerar automaticamente casos de teste a partir de requisitos escritos em linguagem natural. Além disso, podem auxiliar em testes de regressão, identificando partes do código afetadas por mudanças (SILVA, Bruno Augusto Cota, 2023).

Segundo Silva, Bruno Augusto Cota (2023, p. 14):

À medida que aplicativos de software mais complexos são desenvolvidos, o tempo está se tornando um fator crítico para lançar aplicativos que devem ser completamente testados e estar em conformidade com os Requisitos de Negócios. A IA desempenha um papel fundamental no Teste de Software, proporcionando resultados mais precisos e economizando tempo. Este artigo discute os pilares-chave da Inteligência Artificial que podem ser utilizados no Teste de Software. Ele também lança uma visão sobre como o futuro será em termos de Inteligência Artificial e Teste de Software. Os resultados mostram que a IA pode alcançar melhores resultados no Teste de Software e que os testes impulsionados pela IA liderarão a nova era do trabalho de garantia de qualidade (QA) em um futuro próximo. O Teste de Software baseado em IA reduzirá o tempo de lançamento no mercado e aumentará a eficiência da organização para produzir software mais sofisticado, criando testes automatizados mais inteligentes.

As LLMs têm se mostrado cada vez mais relevantes no apoio à área de testes de software, oferecendo recursos que vão desde a geração automática de casos de teste até a análise de cenários complexos que demandariam tempo considerável de uma análise humana. Com a capacidade de compreender requisitos, interpretar código e sugerir melhorias, esses modelos podem acelerar a detecção de falhas e contribuir para uma maior cobertura de testes.

3 METODOLOGIA

3.1 TIPO DE PESQUISA

Esta pesquisa foi de natureza quantitativa, pois buscou compreender os impactos das Large Language Models (LLMs) no desenvolvimento de aplicações construídas no framework Django a partir da análise comparativa entre duas LLMs, utilizando respostas para problemas propostos em um protótipo de sistema de cadastro de produtos e da coleta de dados numéricos sobre as respostas geradas. A abordagem quantitativa possibilitou mensurar a adoção e o impacto das LLMs no desenvolvimento de software.

A pesquisa teve o intuito de compreender de que forma as LLMs podem auxiliar os desenvolvedores de software e de que forma isso tem acontecido até o momento, a partir da análise de informações relevantes e da coleta de dados numéricos. Para isso, foi desenvolvido um protótipo em

Django de um sistema de cadastro de produtos para efetuar os testes nas LLMs no contexto da aplicação protótipo.

3.2 VARIÁVEIS TESTÁVEIS

Para atender aos objetivos específicos da pesquisa, foram definidas cinco variáveis testáveis:

- Tempo de resposta (variável quantitativa contínua): mensurado em segundos, correspondente ao intervalo temporal decorrido entre o envio do prompt e a conclusão da resposta gerada pelo modelo, considerando-se o processamento integral da saída;
- Número de erros na resposta (variável quantitativa discreta): contabilização sistemática de ocorrências de erros de sintaxe, de lógica ou de incompatibilidade semântica com o contexto da aplicação Django, incluindo-se falsos positivos e omissões;
- Qualidade da resposta (variável qualitativa ordinal): avaliada segundo uma escala hierárquica de cinco níveis, atribuindo-se conceitos descritivos aos respectivos valores numéricos: 1 (Muito Ruim), 2 (Ruim), 3 (Regular), 4 (Bom) e 5 (Excelente). Esta variável considera a precisão técnica da solução proposta, a adequação ao contexto do problema e a utilidade prática do código ou orientação fornecida;
- Clareza da explicação da resposta (variável qualitativa ordinal): análise subjetiva, porém sistemática, convertida na mesma escala ordinal de cinco níveis (1 a 5), com os respectivos conceitos: Muito Ruim, Ruim, Regular, Bom e Excelente. Foram considerados critérios como legibilidade, organização estrutural, didática, uso de exemplos ilustrativos e coerência argumentativa da resposta gerada;
- Explicação da decisão de resposta (variável qualitativa nominal): verificação da presença e da qualidade da fundamentação apresentada pelo modelo para sustentar sua solução, incluindo-se a explicitação do raciocínio lógico, a menção a boas práticas de engenharia de software e a referência a princípios ou padrões adotados.

3.3 COLETA DE DADOS

A coleta de dados foi realizada aplicando problemas para as LLMs no contexto da aplicação, relacionados a bugs e melhorias no protótipo, para que as duas LLMs os resolvessem. As respostas geradas foram registradas em uma planilha, contendo informações sobre tempo de execução, correção do código e clareza das instruções.

Foi desenvolvida uma análise comparativa das LLMs para verificar qual modelo apresentou os melhores resultados, os modelos utilizados foram gemini-2.5-flash e llama-3.3-70b-versatile, atraves de requisição a api utilizando o sdk google genai, exemplo de uso, import google.generativeai as genai. Os dados foram organizados em uma tabela com os seguintes campos:

- Nome do modelo: identificação inequívoca da LLM testada (gemini-2.5-flash e llama-3.3-70b-versatile);
- Prompt utilizado: transcrição integral da consulta enviada ao modelo;
- Tempo de resposta: medição em segundos, realizada por meio de instrumentação da chamada à API;
- Qualidade da resposta: classificação segundo a escala ordinal de cinco níveis (1 a 5), com os respectivos conceitos descritivos (Muito Ruim, Ruim, Regular, Bom, Excelente), abrangendo tanto a precisão técnica quanto a adequação da solução ao problema proposto;
- Números de erros: contabilização discreta de erros de sintaxe, lógica ou incompatibilidade semântica;
- Clareza da explicação: classificação análoga na escala ordinal (1 a 5), consideração legibilidade, organização textual, didática e eficácia comunicativa da resposta;
- Data do teste: registro cronológico da execução, para fins de rastreabilidade e reprodutibilidade.

Todos os testes foram realizados em condições controladas, com temperatura do modelo ajustada para 0,2 (parâmetro de criatividade) e demais configurações padronizadas entre as duas LLMs, de modo a minimizar vieses decorrentes de variações estocásticas nas respostas. As seções de código geradas foram submetidas a verificação sintática e execução em ambiente de testes isolado, para validação objetiva das soluções propostas.

3.4 ANÁLISE DOS DADOS

Todos os dados adquiridos nos testes foram verificados em uma tabela comparativa, exibindo possíveis destaques e problemas nas respostas dos modelos, com o objetivo de comparar o desempenho, a eficiência e a qualidade das respostas geradas pelos modelos de linguagem. Foram considerados, por exemplo, o tempo de resposta da pergunta no contexto da aplicação protótipo, erros de sintaxe, problemas para compreender as perguntas e se as respostas realmente resolveram o problema proposto, permitindo identificar vantagens, limitações e padrões de comportamento de cada modelo.

3.5 LOCAL DE PESQUISA

A pesquisa foi delimitada ao contexto do desenvolvimento de software utilizando Django. Os testes da pesquisa foram desenvolvidos em um ambiente virtual de desenvolvimento de software, a fim de obter a melhor qualidade dos testes dos modelos.

Todos os testes foram realizados no ambiente de desenvolvimento utilizando a IDE VsCode e um Ambiente virtual python criado com o comando “python -m venv venv” na versão 12.1 isolando o

ambiente de teste do ambiente geral da maquina utilizada para o experimento. Este estudo foi desenvolvido utilizando o ambiente de software integrado Visual Studio Code, tendo como linguagem principal Python com o framework Django. Foi configurado um ambiente para suportar práticas de desenvolvimento ágil e integração contínua, permitindo a implementação e o teste das funcionalidades do sistema.

3.6 ASPECTOS ÉTICOS

A pesquisa do presente artigo respeitou todas as licenças de software utilizadas, garantindo que ferramentas e códigos analisados fossem empregados dentro dos limites legais. O estudo foi conduzido com integridade científica, assegurando que a coleta, a análise e a apresentação dos dados refletissem fielmente o ambiente de desenvolvimento estudado.

3.7 PROCEDIMENTO EXPERIMENTAL

O protótipo utilizado nos testes consiste em um sistema de cadastro de produtos desenvolvido com Django 5.2 e Python 3.12.10. A aplicação contempla as seguintes funcionalidades básicas: listagem de produtos com paginação; busca por nome de produto; ordenação por diferentes campos (nome, preço, quantidade em estoque); cadastro, edição e exclusão de produtos. O código-fonte do protótipo é composto por aproximadamente 15 arquivos, distribuídos entre models, views, templates, urls e arquivos de configuração. O banco de dados utilizado foi o SQLite, padrão do Django para ambientes de desenvolvimento.

3.7.1 Descrição dos bugs e prompts utilizados

Foram elaborados três cenários de teste, cada um com um prompt específico e um bug ou melhoria a ser implementada, conforme descrito na tabela 1.

Tabela 1 – Cenários de teste, prompts e objetos de avaliação

Objeto de Avaliação	Objeto de Avaliação	Prompt (resumido)
Bug Sintático	Identificação e correção de 5 erros de sintaxe	"Corrija os erros de sintaxe no código abaixo, forneça o menor patch possível no formato git diff"
Teste Unitário	Geração de caso de teste para o modelo Produto	"Crie um teste unitário para o modelo Produto que valide a criação de um produto com nome e preço"
Refatoração	Refatoração de view monolítica	"Refatore a view abaixo aplicando boas práticas do Django, separação de responsabilidades e princípios DRY"

Fonte: O autor (2026)

Os testes foram conduzidos nas seguintes condições:

- Hardware: Processador Intel(R) Celeron(R) N4020C, 4GB RAM, SSD 115GB;
- Software: Visual Studio Code 1.121, Python 3.12.1, Django 4.2;
- Modelos avaliados: gemini-2.5-flash (API Google) e llama-3.3-70b-versatile (API Groq);
- Número de execuções: cada prompt foi submetido 2 (duas) vezes para cada modelo, totalizando 12 execuções (2 modelos × 3 cenários × 2 repetições);
- Medição de tempo: realizada por meio da função time.time() do Python, registrando o intervalo entre o envio da requisição e o recebimento da resposta completa.

3.7.2 Critérios de avaliação

Para minimizar a subjetividade inerente à avaliação qualitativa, adotaram-se os seguintes critérios objetivos:

- Qualidade da resposta: avaliada com base na correção sintática do código gerado; execução bem-sucedida do código sem erros; aderência às restrições explícitas do prompt (ex: "menor patch possível"); aplicação de boas práticas do Django.
- Clareza da explicação: avaliada com base na presença de explicação passo a passo; uso de terminologia técnica adequada; inclusão de comandos para execução; exemplos de saída esperada.
- Número de erros: contabilização objetiva de erros de sintaxe; erros de lógica; falsos positivos (apontamento de erros inexistentes); violações de restrições do prompt.

As avaliações foram realizadas pelos dois autores do artigo, de forma independente, e as eventuais divergências foram resolvidas por consenso.

4 RESULTADOS

4.1 TESTE RESOLUÇÃO DE BUG SINTÁTICO

O modelo Gemini 2.5 Flash identificou corretamente todos os erros sintáticos presentes no código: parêntese não fechado na linha 3, digitação incorreta da variável request como reques na linha 8, string literal "nome" sem fechamento de aspas na linha 10, uso incorreto da variável qq em vez de qs na linha 11, e digitação total em vez de totais na linha 22. O patch foi fornecido no formato git diff, corrigindo exclusivamente as linhas problemáticas, conforme solicitado. A explicação detalhou a causa raiz de cada erro, apresentou medidas preventivas e listou comandos de teste completos. Diante disso, o modelo recebeu nota 5 (Excelente) para qualidade da resposta e nota 5 (Excelente) para clareza da explicação.

Já o modelo Llama 3.3 70B Versatile listou corretamente alguns erros, mas incluiu dois falsos positivos: afirmou que a variável busca não estava definida antes do uso (o que é falso, pois busca =

request.GET.get(...) está correta) e que faltava fechamento de parêntese na linha 20, quando o erro real estava na linha 3. O patch incluiu suposições desnecessárias, como substituir ORDENACAO_PERMITIDA por uma lista fixa, ferindo a instrução de fazer o "menor patch possível". A explicação, embora extensa, continha imprecisões que comprometem a confiabilidade do diagnóstico. Assim, o modelo recebeu nota 3 (Regular) para qualidade da resposta e nota 4 (Bom) para clareza da explicação.

```
from django.db import models
from django.utils import timezone

class Produto(models.Model):

nome = models.CharField(max_length=100, null=False, blank=False)    preco =
models.DecimalField(        max_digits=20, decimal_places=2, default=0, null=False,
blank=False    )    quantidade = models.PositiveIntegerField(default=0, null=False,
blank=False)    data_cadastro =models.DateTimeField(        default=timezone.now,        editable=False,    )

class Meta:
      ordering = ["nome"]

def __str__(self):
      return self.nome

@property
def subtotal_estoque(self):
return self.preco * self.quantidade
```

Tabela 2 – Bug sintático no protótipo

MODELOS	Praticidade/execução (1-5)	Solução Correta? (1-5)	Erros sintáticos encontrados na resposta (1-5)	Clareza e explicação da resposta (1-5)	Data do teste	tempo de resposta
gemini-2.5-flash	5	5	0	5	5/4/2026	21 segundos
llama-3.3-70b-versatile	3	3	2	3	5/4/2026	30 segundos

Fonte: O autor (2026)

4.2 TESTE UNITÁRIO:

O modelo Gemini 2.5 Flash criou uma classe TestProdutoModel herdando de django.test.TestCase, com método test_criar_produto_corretamente. Utilizou Produto.objects.create() para criar um produto de exemplo e validou o nome com self.assertEqual(). Adicionalmente, incluiu verificações de boas práticas (assertIsNotNone(produto.pk), contagem de registros). O comando para execução (python manage.py test meuapp) foi apresentado com explicação do funcionamento. Forneceu exemplo da saída esperada no terminal e orientação sobre estrutura de diretórios. O modelo recebeu nota 5 (Excelente) para qualidade da resposta e nota 5 (Excelente) para clareza da explicação.

O modelo Llama 3.3 70B Versatile criou a classe ProdutoTestCase e o método test_criar_produto, seguido de produto.save(), o que é funcional, embora menos conciso que create(). A validação com assertEqual está correta. O comando python manage.py test foi mencionado.

Contudo, a resposta carece de clareza quanto à localização obrigatória do arquivo de testes e não apresenta exemplo de saída esperada. Diante disso, o modelo recebeu nota 4 (Bom) para qualidade da resposta e nota 3 (Regular) para clareza da explicação.

```
@property
def  subtotal_estoque(self):
      return self.preco * self.quantidade
```

```
from decimal import Decimal
from django.test import TestCase
from .models import Produto


class ProdutoModelTest(TestCase):
      def test_subtotal_estoque(self):
            produto = Produto(nome="Teclado", preco=Decimal("10.50"), quantidade=3)
self.assertEqual(produto.subtotal_estoque, Decimal("31.50"))
```

Tabela 3 – Teste unitário

MODELOS	Praticidade/execução (1-5)	Solução Correta? (1-5)	Erros sintáticos encontrados na resposta (1-5)	Clareza e explicação da resposta (1-5)	Data do teste	tempo de resposta
gemini-2.5-flash	5	5	0	5	12/4/2026	35 segundos
llama-3.3-70b-versatile	4	4	0	3	12/4/2026	41 segundos

Fonte: O autor (2026)

4.3 TESTE REFATORAÇÃO DE CÓDIGO

O modelo Gemini 2.5 Flash refatorou a view original utilizando django.views.generic.ListView, que é a prática recomendada no Django para listagens com paginação, busca e ordenação. Criou um arquivo constants.py para externalizar valores mágicos (PRODUCTS_PER_PAGE, LOW_STOCK_THRESHOLD, campos de ordenação permitidos), aplicando o princípio DRY. Sobrescreveu os métodos get_queryset() (para aplicar filtros e ordenação) e get_context_data() (para incluir agregados de estoque e variáveis de UI). Criou um método auxiliar privado _get_stock_aggregates() para isolamento da lógica de cálculos. Forneceu template HTML completo com formulário de busca, seletor de ordenação, tabela de produtos, resumo de estoque e links de paginação que preservam os parâmetros da URL. Configurou corretamente o arquivo de URLs. A explicação detalhou a aplicação dos princípios SRP (Single Responsibility Principle), DRY e Clean Code. O modelo recebeu nota 5 (Excelente) para qualidade da resposta e nota 5 (Excelente) para clareza da explicação.

O modelo Llama 3.3 70B Versatile propôs a criação de uma ProdutoService que recebe o objeto request no construtor e contém métodos para obtenção, ordenação, cálculo de totais e paginação. A

view ProdutoView foi mantida com @staticmethod e passou a instanciar o serviço. O principal problema identificado foi o acoplamento inadequado do serviço à requisição HTTP: um serviço deve receber parâmetros explícitos (string de busca, campo de ordenação, número da página), não o objeto request, o que viola o princípio de separação de responsabilidades e dificulta testes unitários. Diante disso, o modelo recebeu nota 2 (Ruim) para qualidade da resposta e nota 3 (Regular) para clareza da explicação.

```
qs = Produto.objects.all()
busca = request.GET.get("q", "").strip()
if busca:
    qs = qs.filter(nome__icontains=busca)

ordenar = request.GET.get("ordenar", "nome")
if ordenar not in ORDENACAO_PERMITIDA:
     ordenar = "nome"
qs = qs.order_by(ordenar)
```

```
subtotal_expr = ExpressionWrapper(
   F("preco") * F("quantidade"),
   output_field=DecimalField(max_digits=24, decimal_places=2),
)

totais = qs.aggregate(
   qtd_produtos=Count("id"),
   itens_em_estoque=Sum("quantidade"),
   valor_total_estoque=Sum(subtotal_expr),
)
```

Tabela 4 – Refatoração de código

MODELOS	Praticidade/execução (1-5)	Solução Correta? (1-5)	Erros sintáticos encontrados na resposta (1-5)	Clareza e explicação da resposta (1-5)	Data do teste	tempo de resposta
gemini-2.5-flash	5	5	0	5	13/4/2026	45 segundos
llama-3.3-70b-versatile	3	3	2	3	13/4/2026	65 segundos

Fonte: O autor (2026)

4.4 DISCUSSÃO

Os resultados apresentados evidenciam disparidade sistemática de desempenho entre os dois modelos avaliados: Gemini 2.5 Flash e Llama 3.3 70B Versatile. A análise comparativa fundamenta-se nos três cenários de teste propostos.

4.4.1 Análise comparativa por cenário

Quanto à correção de bug sintático, o modelo Gemini 2.5 Flash identificou corretamente todos os cinco erros, demonstrando precisão diagnóstica superior. Em contraste, o Llama 3.3 70B Versatile

apresentou dois falsos positivos e desrespeitou a restrição explícita do "menor patch possível". Este resultado corrobora os achados de Xu et al. (2025), segundo os quais LLMs de código aberto tendem a apresentar maior taxa de falsos positivos em tarefas de localização de bugs. Conforme demonstraram os autores, o desempenho desses modelos degrada-se quando a tarefa exige alta precisão diagnóstica.

Durante teste unitário, ambos os modelos produziram soluções funcionalmente corretas. O Llama, entretanto, omitiu informações relevantes sobre a localização obrigatória do arquivo de testes e não apresentou exemplos de saída esperada. Este achado está alinhado com o estudo de Zhou et al. (2025), no qual se identificou que ferramentas de AI pair programming frequentemente apresentam respostas incompletas ou imprecisas, reduzindo a usabilidade prática para desenvolvedores menos experientes.

A refatoração de código foi o cenário no qual a diferença de desempenho entre os modelos se tornou mais pronunciada. O Gemini 2.5 Flash propôs solução alinhada às práticas recomendadas do Django, incluindo o uso de ListView, externalização de constantes e aplicação dos princípios SRP (Single Responsibility Principle) e DRY (Don't Repeat Yourself). O Llama 3.3 70B Versatile, por sua vez, apresentou solução arquiteturalmente inadequada, caracterizada pelo acoplamento indevido entre a camada de serviço e o objeto request. Tal violação do princípio de separação de responsabilidades compromete a testabilidade e a manutenibilidade do código.

4.4.2 Padrões observados

A precisão diagnóstica mostrou-se como o primeiro diferenciador entre os modelos. O Gemini apresentou zero falsos positivos, enquanto o Llama demonstrou tendência a alucinações diagnósticas. Conforme Xu et al. (2025), LLMs de código aberto apresentam maior variabilidade em tarefas de localização precisa de bugs.

Um segundo aspecto distintivo foi a aderência a restrições. O Gemini demonstrou maior capacidade de seguir instruções restritivas específicas. Zhou et al. (2025) identificaram que problemas de compatibilidade e erros internos estão entre as causas mais frequentes de falhas em ferramentas de AI pair programming.

A qualidade pedagógica das respostas constituiu o terceiro padrão observado. As respostas do Gemini foram mais completas, incluindo orientações complementares. Zhou et al. (2025) destacam que desenvolvedores valorizam LLMs capazes de fornecer explicações claras e orientações práticas.

4.4.3 Implicações práticas

Os resultados sugerem que a escolha da LLM para auxílio no desenvolvimento com Django deve considerar a complexidade da tarefa. Para tarefas simples e bem delimitadas, como a geração de testes unitários, modelos de código aberto como o Llama podem ser suficientes. Para tarefas que

exigem precisão diagnóstica ou conhecimento arquitetural aprofundado, modelos proprietários como o Gemini 2.5 Flash demonstraram superioridade consistente.

Recomenda-se, conforme Zhou et al. (2025), que praticantes avaliem criticamente as sugestões geradas por LLMs, especialmente em contextos de alta complexidade. Processos de verificação devem ser estabelecidos para mitigar os riscos de falsos positivos e violações arquiteturais.

4.4.4 Limitações do estudo

Embora os resultados sejam consistentes com a literatura, algumas limitações devem ser explicitadas. A primeira delas diz respeito à amostra, composta por apenas dois modelos e três cenários de teste, o que restringe a capacidade de generalização dos achados para outras LLMs ou contextos de desenvolvimento. Em segundo lugar, a ausência de repetições estatisticamente significativas impede a análise de variabilidade intra-modelo e a aplicação de testes de significância estatística. A terceira limitação refere-se ao parâmetro de temperatura fixado em 0,2, valor que maximiza a determinismo das respostas, mas que não reflete cenários de uso mais exploratórios ou criativos, nos quais temperaturas mais elevadas são comumente empregadas. Por último, a avaliação da clareza das explicações, embora sistemática e baseada em escala ordinal padronizada com validação por consenso entre os autores, mantém um componente de subjetividade inerente a qualquer análise qualitativa.

5 CONSIDERAÇÕES FINAIS

Esta pesquisa avaliou o potencial das Large Language Models (LLMs) no auxílio ao desenvolvimento de aplicações web construídas com o framework Django Python. A análise comparativa foi conduzida entre dois modelos, Gemini 2.5 Flash e Llama 3.3 70B Versatile, submetidos a três cenários distintos: correção de bugs sintáticos, geração de testes unitários e refatoração de código.

Os resultados obtidos demonstram que ambos os modelos são capazes de auxiliar no desenvolvimento com Django, porém com níveis de eficácia distintos. O Gemini 2.5 Flash apresentou desempenho superior em todos os cenários. Sua precisão diagnóstica atingiu 100%, sem falsos positivos. Sua aderência a restrições explícitas dos prompts foi rigorosa. Suas soluções aplicaram consistentemente boas práticas arquiteturais do Django. Suas respostas apresentaram completude pedagógica, incluindo orientações sobre estrutura de diretórios, comandos de execução e exemplos de saída esperada.

O Llama 3.3 70B Versatile mostrou-se funcional em tarefas de menor complexidade, como evidenciado pelo desempenho Bom (nota 4) no teste unitário. Em cenários mais desafiadores, porém, apresentou limitações significativas. No teste de bug sintático, produziu dois falsos positivos e violou a restrição do "menor patch possível", obtendo qualidade Regular (nota 3). No teste de refatoração,

propôs solução arquiteturalmente inadequada, caracterizada pelo acoplamento indevido entre serviço e requisição HTTP, resultando em qualidade Ruim (nota 2).

Estes achados dialogam com a literatura recente sobre programação assistida por IA. Zhou et al. (2025) identificaram que as Operation Issues, incluindo respostas imprecisas ou incompletas, estão entre as principais limitações de ferramentas de AI pair programming, corroborando as dificuldades observadas no modelo Llama. Xu et al. (2025) demonstraram que o desempenho de LLMs de código aberto varia significativamente conforme a complexidade da tarefa, padrão consistente com os resultados aqui apresentados.

Este artigo oferece três contribuições práticas. Primeira, uma metodologia replicável para avaliação comparativa de grandes modelos de linguagem no contexto do desenvolvimento web com Django. Segunda, métricas objetivas, incluindo tempo de resposta, qualidade, clareza e número de erros, que podem subsidiar a escolha de modelos. Terceira, evidências empíricas sobre as capacidades e limitações dos modelos testados, auxiliando desenvolvedores e gestores na tomada de decisão.

Reconhecem-se as seguintes limitações do estudo. A amostra restringiu-se a dois modelos e três cenários de teste. Não foram realizadas repetições estatisticamente significativas. A temperatura foi fixada em 0,2, o que pode não representar cenários de uso mais criativos. A subjetividade residual na avaliação da clareza das explicações foi mitigada pelo uso de escala ordinal padronizada e avaliação por consenso entre os autores.

Para trabalhos futuros, sugerem-se cinco direções de pesquisa. A ampliação da amostra para incluir outros modelos como GPT-4, Claude e CodeLlama. A realização de testes com projetos Django de maior porte e complexidade. A investigação do impacto de diferentes temperaturas no desempenho dos modelos. Estudos longitudinais que avaliem não apenas a correção imediata das respostas, mas também a manutenibilidade do código gerado ao longo do tempo. Pesquisas sobre a segurança e confiabilidade do código produzido por LLMs em contextos críticos.

Conclui-se que os modelos de linguagem de grande escala, especialmente o Gemini 2.5 Flash, representam ferramentas estratégicas para otimizar o desenvolvimento em Django. Seu uso pode reduzir o tempo de depuração, melhorar a qualidade do código e acelerar o aprendizado de boas práticas do framework. Recomenda-se, no entanto, que seu uso seja crítico e complementar à expertise humana, considerando as limitações inerentes a cada modelo, particularmente a maior propensão a falsos positivos e violações arquiteturais observada em LLMs de código aberto como o Llama. Conforme recomendam Zhou et al. (2025), praticantes devem estabelecer processos de verificação para mitigar os riscos associados à dependência excessiva de assistentes de programação baseados em IA.

REFERÊNCIAS

ZHOU, Xiyu; LIANG, Peng; ZHANG, Beiqi; et al. Exploring the problems, their causes and solutions of AI pair programming: A study on GitHub and Stack Overflow. Journal of Systems and Software, v. 219, n. 112204, 2025. Disponível em: https://jyx.jyu.fi/handle/123456789/99112.

XU, Chuyang; et al. FlexFL: Flexible and effective fault localization with open-source large language models. ArXiv:2411.10714, 2025. Disponível em: https://arxiv.org/abs/2411.10714.

LI, Zhenhao; et al. Empirical Evaluation of Large Language Models for Novice Program Fault Localization. In: IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2024. Disponível em: https://ieeexplore.ieee.org/document/10684637.

LIMA, F. R.; AZEVEDO, A. I. R.; BARBOSA, H. F. A influência do GitHub Copilot no ensino e aprendizado de programação: percepções e desafios. In: Anais do XXXIV Simpósio Brasileiro de Informática na Educação (SBIE), 2023, Passo Fundo. p. 123-134. Disponível em: https://sol.sbc.org.br/index.php/sbie/article/view/24567.

CAMPOS, A. G. M.; BARBOSA, M. S. S. Uso de modelos de inteligência artificial na geração de código computacional: um estudo sobre percepção docente. Brazilian Journal of Development, Curitiba, v. 9, n. 10, p. 27997-28013, 2023. DOI: 10.34117/bjdv9n10-007. Disponível em: https://ojs.brazilianjournals.com.br/ojs/index.php/BRJD/article/view/63640.

HUANG, K. et al. A Survey on Evaluation of Large Language Models. arXiv:2307.03109, 2023. Disponível em: https://arxiv.org/abs/2307.03109.

LI, X. et al. Large Language Models for Artificial General Intelligence (AGI): A Survey of Foundational Principles and Approaches. arXiv:2501.03151, 2025. Disponível em: https://arxiv.org/abs/2501.03151.

ZHANG, S. et al. A Survey of GPT-3 Family Large Language Models Including ChatGPT and GPT-4. arXiv:2310.12321, 2023. Disponível em: https://arxiv.org/abs/2310.12321.

AHN, M. et al. Do As I Can, Not As I Say: Grounding Language in Robotic Affordances. arXiv:2204.01691, 2022. Disponível em: https://arxiv.org/abs/2204.01691.

GAO, L. et al. PAL: Program-Aided Language Models. arXiv:2211.10435, 2022. Disponível em: https://arxiv.org/abs/2211.10435.

SCHICK, T. et al. Toolformer: Language Models Can Teach Themselves to Use Tools. arXiv:2302.04761, 2023. Disponível em: https://arxiv.org/abs/2302.04761.

CHEN, M. et al. Evaluating Large Language Models Trained on Code. arXiv:2107.03374, 2021. Disponível em: https://arxiv.org/abs/2107.03374.

FENG, Z. et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155, 2020. Disponível em: https://arxiv.org/abs/2002.08155.

LU, S. et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. arXiv:2102.04664, 2021. Disponível em: https://arxiv.org/abs/2102.04664.