UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

CAIO ROBERTO RAMOS DA SILVA

Redução do Débito Técnico de Testes em Código com o Auxílio de Ferramentas de Inteligência Artificial Generativa

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof[a]. Dr[a]. Karina Kohl Silveira

Porto Alegre
2025

RESUMO

O débito técnico de testes constitui um desafio crítico no desenvolvimento ágil de software, decorrente, frequentemente, da priorização de novas funcionalidades em detrimento da qualidade e da validação rigorosa. Embora as ferramentas de Inteligência Artificial Generativa (GenAI) tenham apresentado rápida difusão no suporte ao desenvolvimento, sua aplicação em contextos corporativos enfrenta barreiras significativas. Tais limitações incluem o risco à confidencialidade do código-fonte e a carência de conhecimento específico, visto que os modelos pré-treinados não possuem contexto sobre o projeto ou apresentam informações desatualizadas.

Este trabalho, estruturado como um estudo de caso em uma *startup* de tecnologia, propõe uma solução baseada na arquitetura de Geração Aumentada por Recuperação (RAG). O objetivo central é desenvolver e avaliar um sistema capaz de gerar testes automatizados *end-to-end* (E2E) para reduzir o débito técnico de testes. O protótipo, implementado em Python, utiliza a plataforma Ollama para a execução local de Grandes Modelos de Linguagem (LLMs), assegurando a privacidade do código. A base de conhecimento do RAG foi composta pelo código-fonte em React Native e pela documentação técnica da ferramenta Maestro. Os resultados demonstram que a arquitetura RAG local é uma estratégia viável e segura para auxiliar equipes de QA na redução do débito técnico, superando as limitações de modelos genéricos ao fornecer comandos contextualmente precisos.

Palavras-chave: Inteligência Artificial. Débito Técnico. RAG. LLM. Teste de Código. Ollama. LangChain. Crawler.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

LLM	Large Language Model
E2E	End-to-end
RAG	Retrieval-Augmented Generation
TS	TypeScript
JS	JavaScript
AI	Artificial Intelligence
Gen-AI	Generative Artificial Intelligence
IA	Inteligência Artificial
GPT	Generative Pre-trained Transformer
IDE	Integrated Development Environment
DevOps	Development Operations

SUMÁRIO

1 INTRODUÇÃO

Atualmente, observa-se um cenário de rápidas transformações tecnológicas, impulsionado tanto pelo aumento do poder de processamento disponível quanto pela ampliação do acesso à rede mundial de computadores. Estima-se que aproximadamente 67% da população mundial esteja conectada, segundo dados da International Telecommunication Union (UNION, 2024). No contexto brasileiro, esse percentual atinge 92,5% dos domicílios, conforme aponta o IBGE (IBGE, 2024), o que evidencia um ambiente altamente propício à disseminação e adoção de novas tecnologias digitais.

Nesse panorama de ampla conectividade, o lançamento do ChatGPT pela OpenAI (OPENAI, 2022), em dezembro de 2022, marcou um ponto de inflexão na popularização de sistemas baseados em Inteligência Artificial Generativa. A ferramenta obteve adesão massiva, atingindo o recorde de 100 milhões de usuários em apenas dois meses, de acordo com Hu (HU, 2023). Esse fenômeno impactou diretamente o ecossistema de desenvolvimento de software, alterando práticas e reduzindo a dependência de fontes tradicionais de consulta técnica. Um indicador notável desse movimento, apontado por Orosz (OROSZ, 2025), foi o declínio acentuado nas postagens da plataforma StackOverflow, sugerindo uma migração de desenvolvedores para o uso de LLMs na resolução de dúvidas e problemas de código.

Este trabalho consiste em uma pesquisa-ação conduzida em uma *startup* de tecnologia na qual o autor desempenha funções de desenvolvimento. Constatou-se que a organização carece de uma política formal para a definição e manutenção de testes automatizados, apesar de possíveis cláusulas contratuais de qualidade. No âmbito do projeto analisado - um aplicativo móvel em React Native -, a implementação de testes é inexistente devido a prazos reduzidos e à priorização de demandas de negócio. Tal lacuna amplia o risco de regressões e compromete a detecção precoce de falhas críticas.

Diante desse cenário, o objetivo deste trabalho é propor e implementar uma estratégia de automação de testes fim a fim (end-to-end), com foco em apoiar o time de Quality Assurance (QA) na elaboração de testes de regressão automatizados integrados ao pipeline de CI/CD. Espera-se, com essa iniciativa, elevar o nível de confiabilidade do sistema, reduzir retrabalho e fortalecer a cultura de qualidade dentro do ciclo de desenvolvimento de software da empresa.

A aplicação de IA mostra-se pertinente neste contexto, pois o time de QA não tem conhecimento em ferramentas de testes para aplicativos móveis e, ao mesmo tempo, o

time de desenvolvimento não tem experiência em escrever casos de teste - no sentido de planejar as possibilidades relevantes para testes. A possibilidade de gerar testes usando IA, tende a otimizar significativamente o processo para as equipes escreverem testes, seja algo feito inteiramente por uma LLM ou com uma boa base de template.

É importante destacar que os LLMs são treinados com grandes volumes de dados amplamente diversificados, resultando em uma base de conhecimento geral e potencialmente desatualizada. Por essa razão, torna-se necessária a integração de uma abordagem de *Retrieval-Augmented Generation* (RAG), capaz de incorporar conhecimento específico sobre o código do aplicativo e a documentação da ferramenta de testes utilizada, como o Maestro (INC, 2022), que é abordado em 2.2.1.

Sem o uso de um RAG, seria necessário fornecer o código relevante para uma LLM como o Github Copilot (MICROSOFT; GITHUB, 2021), o Gemini (GOOGLE; ALPHABET, 2023) ou ChatGPT (OPENAI, 2022), e como ressaltado anteriormente, existe a possibilidade de as LLMs terem usado a documentação do Maestro, nos respectivos treinamentos, porém, teriam usado uma documentação antiga; sendo assim, também teríamos que fornecer toda a documentação da ferramenta.

Outra questão sobre usar LLMs de terceiros é a confidencialidade do código. Atualmente, diversas empresas ainda estão construindo políticas sobre o uso de IAs e LLMs pelos funcionários. A empresa em que o autor atua tem uma política muito restritiva, mesmo que um funcionário utilize uma ferramenta que declara não usar os dados das conversas para treinamento ele é impedido de usá-la para trabalho se houver dados sensíveis.

1.1 Objetivos

1.1.1 Geral

Este trabalho tem por objetivo geral desenvolver e avaliar um sistema RAG para gerar testes automatizados de código, visando reduzir o débito técnico.

1.1.2 Específicos

Os objetivos específicos consistem em:

- Identificar na literatura casos comuns de débito técnico de testes;
- Mapear o débito técnico de testes na empresa;
- Avaliar o suporte de LLMs e RAGs na geração e manutenção de testes;
- Projetar uma arquitetura RAG para geração dos testes automatizados identificados como débito técnico;
- Avaliar a qualidade dos testes gerados medindo o impacto em métricas como cobertura, legibilidade e manutenibilidade dos testes;
- Implementar e integrar o protótipo ao pipeline de desenvolvimento;

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Engenharia de Software e Testes

De acordo com Ham e Vocke (VOCKE, 2018), a disponibilização de um *software* em ambiente de produção deve ser precedida por processos de teste que assegurem o cumprimento de requisitos funcionais e não funcionais. A automação desses testes reduz substancialmente o tempo necessário para a identificação de falhas em comparação com a execução manual, além de constituir um pilar para o desenvolvimento ágil e práticas de DevOps.

Uma das abordagens fundamentais para a qualidade do código é o Desenvolvimento Orientado por Testes (TDD, do inglês *Test-Driven Development*), metodologia sistematizada por Beck (BECK, 2002). O TDD preconiza que o teste deve ser escrito antes da implementação funcional, forçando a falha inicial para validar a lógica de verificação. Conforme observa Washizaki (WASHIZAKI, 2025), essa prática induz o desenvolvedor a refletir sobre o *design* e os requisitos precocemente.

O movimento de DevOps (BUCHANAN, s.d.) iniciou entre 2007 e 2008, com o objetivo de resolver problemas que, mesmo com o agile, persistiam, principalmente em relação à comunicação entre time de desenvolvimento e operações (ou infraestrutura). A base do DevOps é a integração e entrega contínuas (CI/CD), ou seja, práticas como o agendamento prévio e rígido de implantações deixam de ser aceitáveis em ambientes modernos, são necessárias esteiras (pipelines) que executem diversos testes em cima do código que foi submetido, e passando nesses testes, seja entregue em produção. Com isso, é possível realizar múltiplas entregas diárias, sem paralisar setores ou equipes e assegurando que o ambiente de produção seja continuamente validado pelos testes automatizados.

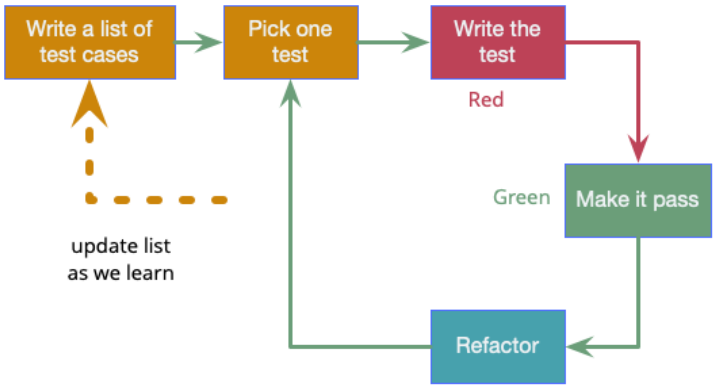
Outra prática fundamental para o DevOps é o versionamento de código. Ele permite rastrear a origem de todas as alterações, desde o desenvolvimento até o que está em produção, garantindo a rastreabilidade. Além disso, o versionamento possibilita estratégias de entrega mais controladas, como o versionamento semântico (*Semantic Versioning*), que agrupa um conjunto de *commits* em um pacote de versão fechado para ser validado e implantado.

Como dito anteriormente, os testes feitos nas esteiras de entrega são o que garantem o funcionamento do software, porém, é preciso entender que a garantia é diretamente

proporcional à qualidade e abrangência dos testes.

Uma abordagem possível para a escrita de testes é a TDD (Test Driven Development), desenvolvida por Kent Beck e apresentada oficialmente em (BECK, 2002), ela preconiza que, antes de escrever qualquer código, se escrevam os testes, eles devem falhar, pois a implementação não existe, então o código é escrito e os testes são executados novamente, caso os testes sejam bem-sucedidos, prossegue-se à escrita dos próximos testes, caso haja alguma falha, o código deve ser corrigido. Um fluxo desses passos pode ser visto abaixo na Figura 2.1.

Figura 2.1: Fluxo TDD



Fonte: (FOWLER, 2023)

Essa sequência de passos, segundo (WASHIZAKI, 2025), leva os desenvolvedores a refletirem sobre os requisitos e o design antes de escrever o código, expondo problemas de requisitos ou design mais cedo. Apesar dessas vantagens, o TDD apresenta baixa adoção. Segundo o report da Practitest (PRACTITEST, 2024), apenas 23% dos participantes da pesquisa disseram que a sua organização segue práticas do TDD.

Nicolaescu (NICOLAESCU,) acredita que o TDD esteja em declínio por conta das LLMs que são muito boas para a geração de casos de testes, principalmente porque escrever testes é entediante e envolve tarefas repetitivas, o que pode levar o ser humano a falhas.

Outro ponto é a necessidade de atualizar os testes sempre que algo for alterado, no código ou nos requisitos, e acaba sendo um esforço adicional.

A abrangência dos testes, chamada cobertura de código, usualmente tem métricas de fácil entendimento. Diversas formas ou categorias podem ser usadas para avaliar a cobertura, de acordo com a Atlassian (ATLASSIAN,):

- Cobertura de Função: quantas das funções declaradas foram chamadas

- Cobertura de Comandos: quantos comandos foram executados no programa
- Cobertura de Ramificações: quantos ramos das estruturas de controle foram executados
- Cobertura de Condição: Quantas subexpressões booleanas foram testadas com true e false
- Cobertura de Linhas: Quantas linhas do código fonte foram testadas

É comum ver os resultados desses testes em formato de porcentagem ou

$$\frac{\text{\# de <tipo> executados}}{\text{\# de <tipo> encontrados no código}}$$

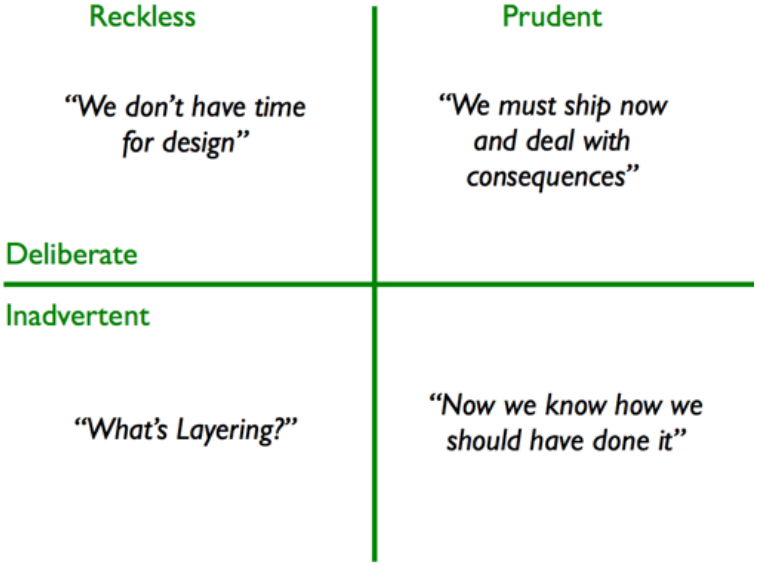
Os testes podem ser classificados em diferentes níveis, como os de unidade (validando componentes individuais) e os de integração (verificando a interação entre componentes), ambos citados em (BECK, 2002). Este trabalho, no entanto, foca em uma categoria de teste de sistema: o **teste *End-to-End* (E2E)**. O teste E2E valida o fluxo completo de uma funcionalidade da perspectiva do usuário, simulando sua interação desde a interface (UI) até os serviços de *backend* e bancos de dados. Embora cruciais para garantir que o sistema funcione como um todo, os testes E2E são historicamente complexos de escrever e manter, tornando-os um ponto central de acúmulo de débito técnico.

2.2 Débito Técnico de Testes

Dado que diversos autores citaram a necessidade de testes em projetos de software para o controle de débitos técnicos, a IEEE tem um capítulo dedicado a testes na versão mais recente do livro lançado por eles, o Software Engineer Book of Knowledge (SWEBOK), disponível gratuitamente em https://www.computer.org/education/bodies-of-knowledge/software-engineering. Segundo IEEE a importância de testes vem de fatores como mitigação de riscos, confiança, conformidade, satisfação do usuário, otimização e economia financeira, é possível perceber que temos fatores técnicos e de negócio, os testes não servem só para uma área ou outra, mas para o produto como um todo. Ao mesmo tempo, existem desafios para a introdução ou manutenção dessa prática, como ambientes de teste que reproduzam fielmente ambientes reais, cobertura de testes adequada e escalabilidade, visto que pode haver problemas que só ocorrem em escala.

A seguir, reproduzo parcialmente os tipos de teste que são apresentados pelos

Figura 2.2: Tipos de débito técnico



Fonte: Martin Fowler, 2009

autores:

- Teste funcional – Avaliação que o software funciona como o esperado.
- Teste não funcional – Testa atributos qualitativos como segurança, desempenho, confiabilidade.
- Teste de usabilidade – Testa a facilidade de uso da perspectiva de um usuário final.
- Teste de aceitação – Validação que o software está de acordo com o critério especificado e funciona de forma satisfatória para os usuários alvo.
- Teste de regressão – Retesta o software após modificações para garantir que não foram criados novos problemas.
- Teste de acessibilidade – Testa a facilidade de uso por pessoas com deficiência.

2.2.1 Testes End-to-End (E2E) e a Ferramenta Maestro

Um tipo de teste que frequentemente engloba várias das categorias acima, como testes funcionais e de regressão, é o teste *End-to-End* (E2E). O E2E simula a jornada completa de um usuário real através da aplicação, validando o fluxo de ponta a ponta - desde a interface do usuário (UI) até bancos de dados e APIs.

No contexto de aplicações móveis (como o React Native, foco deste estudo), a escrita de testes E2E robustos é historicamente complexa, exigindo ferramentas como Appium ou Detox, que demandam conhecimento em linguagens de programação (ex: Java,

JavaScript) e manutenção intensiva. Essa complexidade técnica é uma barreira frequente para equipes de Quality Assurance (QA) e contribui para o acúmulo de débito técnico.

Para endereçar esse problema, surgiram ferramentas modernas como o Maestro (INC, 2022). O Maestro é uma ferramenta de código aberto projetada para simplificar drasticamente os testes E2E em iOS, Android e Web. Sua principal vantagem é a abordagem declarativa, onde os fluxos de teste são escritos em arquivos de formato YAML, utilizando comandos simples e legíveis em linguagem natural (ex: `tapOn`, `assertVisible`, `scrollUntilVisible`).

Essa simplicidade reduz a fragilidade (*flakiness*) e a complexidade da escrita de testes, permitindo que sejam criados e mantidos mais rapidamente. Conforme observado neste trabalho, embora essa abordagem seja adequada para equipes mistas, ainda exige conhecimento da sintaxe específica do Maestro e dos identificadores do aplicativo. O protótipo RAG desenvolvido nesta pesquisa tem como objetivo automatizar a geração desses arquivos YAML do Maestro, alimentando o LLM com o código-fonte e a documentação da ferramenta.

Nesse contexto, e diante da crescente popularização de ferramentas de IA, inclusive específicas para o desenvolvimento de software, o trabalho proposto busca enfrentar um problema recorrente em projetos de software: o débito técnico associado aos testes de código. Esse débito pode decorrer de fatores como a ausência de manutenção adequada, a baixa cobertura de cenários ou falta de compreensão, por parte dos envolvidos no projeto, acerca do papel e da contribuição dos testes para a qualidade do produto.

O conceito de débito técnico - ou dívida técnica - refere-se a decisões de desenvolvimento que, embora agilizem a entrega a curto prazo, geram pendências que exigem resolução futura. Segundo Radigan (RADIGAN, s.d.), a utilização de práticas ágeis e de definições de pronto (*Definition of Done*) rigorosas auxilia na prevenção do acúmulo dessa dívida. Complementarmente, Mucci (MUCCI, 2025) destaca que o débito técnico pode ser inevitável sob pressões de prazo, mas sua gestão deve ser planejada para evitar a degradação da arquitetura.

2.3 GenAI e LLMs na Engenharia de Software

De acordo com Cole Stryker (STRYKER, s.d.), LLMs (*Large Language Models*) são uma categoria de modelos de deep learning (LECUN; BENGIO; HINTON, 2015), treinados em uma quantidade imensa de dados, o que os torna capazes de entender e gerar

conteúdo em linguagem natural. As LLMs são construídas sobre uma arquitetura de rede neural chamada Transformer (VASWANI et al., 2017), que é excelente para lidar com sequências de palavras e capturar padrões em texto.

Uma LLM funciona prevendo o próximo token com base no contexto anterior, aprendendo padrões linguísticos complexos e reproduzindo-os de forma coerente. Com essa capacidade, chegou-se a um ponto em que os usuários não precisam adaptar significativamente sua forma de interação com sistemas computacionais, como acontece com buscas tradicionais, que tentam fazer casamento de palavras e relacionar com outras palavras semanticamente próximas, Stryker (STRYKER, s.d.). As LLMs conseguem entender o contexto de forma a conseguir realizar tarefas como resumir um conteúdo, ajudar na depuração de código e até criar um template ou texto base que o usuário peça.

Hoje em dia, diversos modelos de LLM são acessíveis ao público, como o ChatGPT (OPENAI, 2022) da OPENAI, o Claude (ANTHROPIC, s.d.) da ANTHROPIC e o Gemini (GOOGLE; ALPHABET, 2023) do GOOGLE; ALPHABET. Conforme citado na Seção 1, o ChatGPT teve uma adoção extremamente rápida, o que impactou diretamente plataformas de desenvolvedores. Um exemplo notável foi o declínio nos acessos ao StackOverflow, um conhecido fórum de programação, indicando que essas ferramentas estão se tornando satisfatórias para as necessidades dos usuários, pelo menos em relação a dúvidas de código.

O pré-treinamento de uma LLM começa com bilhões ou até trilhões de palavras retiradas de livros, websites, artigos, código e outras fontes de texto. Cientistas de dados são responsáveis por fazer a limpeza dos dados para remover duplicatas, conteúdo indesejado e erros. Depois, o texto é quebrado em unidades menores chamadas de tokens, no processo conhecido como tokenização, Stryker (STRYKER, s.d.). Tokens podem variar entre um caractere até uma palavra, e existem para haver uma padronização que facilitará o manuseio consistente deles.

O Transformer, foi introduzido em (VASWANI et al., 2017) e trouxe grandes mudanças na forma como o processamento de texto é feito; antes, ele era processado palavra por palavra, de forma sequencial, o que era lento e tornava muito difícil para o modelo conectar palavras que estivessem muito distantes entre si na frase. O transformer propõe um modelo baseado em mecanismos de atenção, que permite ao modelo analisar uma frase inteira de uma só vez, dessa forma, ele consegue “prestar atenção” e avaliar a importância de todas as outras palavras da frase simultaneamente, independente da posição, para entender o contexto. Ele não precisa esperar o processamento de uma palavra para pro-

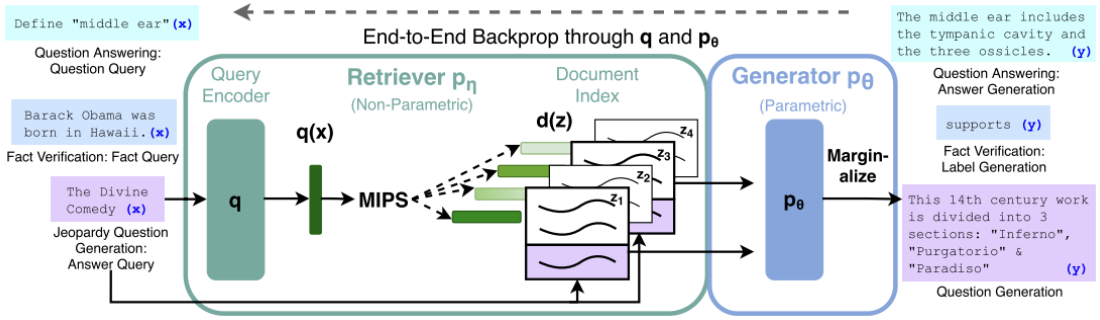
cessar a próxima, todas podem ser processadas em paralelo, o que aumenta drasticamente a velocidade de treinamento.

2.4 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) foi apresentado em (LEWIS et al., 2020) como uma arquitetura híbrida projetada para tarefas de processamento de linguagem natural (PLN, ou NLP em inglês) que exigem bastante conhecimento de algum domínio, como documentação interna de uma empresa. Ele combina dois tipos de memória: paramétrica, que é uma LLM já treinada, e não paramétrica, que é um banco de dados externo acessível.

O funcionamento do RAG utiliza dois componentes principais, um recuperador (retriever) e um gerador (generator). Dado um input, o recuperador primeiro consulta o índice de dados não paramétrico (como um banco de dados) para encontrar K trechos de textos mais relevantes para a consulta. Esses trechos são então adicionados à entrada original e são então fornecidos ao gerador (LLM). Na Figura 2.3

Figura 2.3: Diagrama RAG



Fonte: (LEWIS et al., 2020)

A principal inovação do RAG é conseguir gerar respostas mais factuais, específicas e diversas em comparação com modelos puramente paramétricos, que frequentemente alucinam, principalmente se a entrada pergunta algo que não estava presente, ou com relevância muito baixa, no conjunto de dados em que foi treinado. Outra diferença do RAG é que ele pode ter seu conhecimento atualizado facilmente, basta substituir ou atualizar o índice de documentos por uma versão mais nova ou diferente, sem a necessidade de retreinar o modelo com esses documentos.

Por último, também é possível obter a citação dos documentos em que a saída se baseou, podendo ser conferido facilmente por um ser humano.

2.5 Engenharia de Prompt

A Engenharia de Prompt (ou *Prompt Engineering*) é um campo focado na criação e otimização de instruções, chamadas "*prompts*", para extrair os resultados mais precisos, relevantes e seguros de Modelos de Linguagem Amplos (LLMs) (SARAVIA, 2022).

Dada a natureza dos LLMs, que geram respostas com base no contexto fornecido, a forma como uma pergunta é feita ou uma tarefa é solicitada impacta diretamente a qualidade da saída. Uma instrução bem formulada pode guiar o modelo a produzir respostas factuais.

A literatura, consolidada em guias como o *Prompting Guide* (SARAVIA, 2022), documenta diversas técnicas que evoluíram da simples instrução para métodos complexos de raciocínio. As técnicas fundamentais mais relevantes para este trabalho são:

1. **Prompt de Zero-shot (Zero-shot Prompting)**

   A abordagem *zero-shot* é a forma mais básica de interação. Ela consiste em fornecer uma instrução direta ao LLM, sem oferecer exemplos prévios de como a tarefa deve ser executada. O modelo deve se basear inteiramente em seu treinamento prévio para compreender e realizar a tarefa (SARAVIA, 2022).

   Exemplo: `Resuma o seguinte texto:`

   Esta técnica funciona bem para tarefas genéricas, mas falha quando o formato de saída desejado é muito específico ou a tarefa é complexa.

2. **Prompt de Few-shot (Few-shot Prompting)**

   Para superar as limitações do *zero-shot*, a técnica *few-shot* demonstrou ser altamente eficaz. Nela, o *prompt* inclui, além da instrução, alguns (geralmente de 1 a 5) exemplos de pares entrada/saída que demonstram o padrão esperado. O modelo utiliza esses exemplos como "aprendizado em contexto" (*in-context learning*) para adaptar sua resposta ao formato desejado (SARAVIA, 2022).

   ```
   Traduza a gíria para linguagem formal:
   Entrada:  "Tô de boa."
   Saída:  "Estou me sentindo bem."
   Entrada:  "Essa ideia é osso."
   Saída:  "Essa ideia é problemática."
   Entrada:  "Não tankei."
   Saída:
   ```

3. **Cadeia de Pensamento (Chain-of-Thought - CoT) Prompting**

   A técnica de Cadeia de Pensamento (CoT) foi um avanço significativo para melhorar a capacidade de raciocínio dos LLMs. Ela instrui o modelo a não fornecer

a resposta final imediatamente, mas sim a "pensar passo a passo" e articular uma cadeia de raciocínio lógico que leva à solução (SARAVIA, 2022).

Isso é particularmente eficaz em problemas que exigem múltiplas etapas, como questões de matemática, lógica ou planejamento. Ao forçar o modelo a decompor o problema, a probabilidade de erros diminui. O CoT pode ser ativado por *zero-shot* (ex: "Pense passo a passo") ou por *few-shot* (mostrando exemplos que incluem o raciocínio) (SARAVIA, 2022).

4. **Autoconsistência (Self-Consistency)**

   Uma evolução do CoT é a técnica de Autoconsistência (Self-Consistency). Em vez de gerar apenas um raciocínio, o modelo é instruído a gerar múltiplas cadeias de pensamento (ex: 3 ou 5) para o mesmo problema e, em seguida, utiliza uma votação majoritária para determinar a resposta final mais provável.

Vale ressaltar que a Engenharia de Prompt desempenha um papel fundamental em arquiteturas como o RAG (apresentado na Seção 2.4). No RAG, a qualidade da resposta depende não apenas da recuperação dos documentos, mas também de como esses documentos são inseridos e formatados dentro do prompt do sistema, instruindo a LLM a utilizar aquele contexto específico para gerar a resposta.

2.6 *Embbedding*

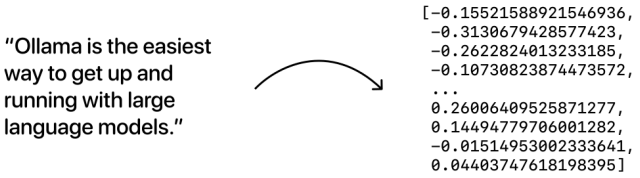
Em essência, *embeddings* são representações numéricas de dados complexos, como texto, imagens ou áudio (HASSANI, 2025).

2.6.1 Modelos de Embedding (*Embedding Models)*

Um *embedding model* (modelo de *embedding*) é um modelo de aprendizado de máquina, geralmente uma rede neural, que converte esses dados não estruturados em vetores numéricos.

O objetivo principal desse processo é traduzir o texto para um formato numérico que os modelos de *machine learning* possam processar e entender. Uma forma visual é apresentada na Figura 2.4

Figura 2.4: Embedding



Fonte: (OLLAMA, 2024)

2.6.2 Funcionamento de *Embedding Models*

2.6.2.1 Treinamento e Significado Semântico

O *embedding model* é treinado em um conjunto massivo de dados de texto, como a internet (HASSANI, 2025). Durante esse treinamento, o modelo aprende a capturar a "essência" ou o "significado semântico" das palavras e frases, analisando os contextos em que elas aparecem.

2.6.2.2 O Espaço Vetorial

O resultado desse treinamento é um "espaço vetorial" de alta dimensão, onde palavras e frases com significados semelhantes são mapeadas para pontos "próximos" nesse espaço (OLLAMA, 2024; HASSANI, 2025). Por exemplo, o modelo aprende a colocar o vetor da palavra "cachorro" perto do vetor de "cão", mas longe do vetor de "computador".

Um exemplo clássico dessa capacidade é a realização de operações vetoriais que aproximam relações semânticas, como:

Vetor("Rei") - Vetor("Homem") + Vetor("Mulher")

O resultado dessa operação é um vetor muito próximo ao *Vetor("Rainha")* (HASSANI, 2025).

2.6.3 Aplicação Principal: Retrieval-Augmented Generation (RAG)

Atualmente, o uso mais comum para *embedding models* é em sistemas de *Retrieval-Augmented Generation* (RAG), que permitem que Modelos de Linguagem Amplos (LLMs)

respondam perguntas usando uma base de conhecimento privada (OLLAMA, 2024; LLAMAINDEX,).

O processo de RAG é composto por duas fases complementares:

1. **Fase 1: Indexação (Armazenamento do Conhecimento)**
    - O sistema é alimentado com os documentos da base de conhecimento.
    - Os documentos são divididos em pedaços menores.
    - O *embedding model* é usado para criar um vetor numérico (um *embedding*) para cada pedaço de texto.
    - Esses vetores são armazenados em um banco de dados vetorial (*vector database*).
2. **Fase 2: Recuperação**
    - Quando um usuário faz uma pergunta (um *query*), o *mesmo embedding model* é usado para converter essa pergunta em um vetor.
    - O sistema então compara o vetor da pergunta com todos os vetores armazenados no banco de dados para encontrar os vetores de documentos mais "próximos" ou semanticamente similares.
    - Os pedaços de texto correspondentes a esses vetores similares são "recuperados" (*retrieved*).
    - Finalmente, esses textos recuperados são enviados ao LLM (como Llama 3 ou Gemma) junto com a pergunta original, servindo como "contexto" para que o LLM possa gerar uma resposta precisa e baseada nos documentos fornecidos.

2.6.4 O Papel em Ferramentas como LangChain e LlamaIndex

Ferramentas como LangChain [1] e LlamaIndex [2] usam os *embedding models* como um componente modular essencial. Elas fornecem uma interface padronizada que permite aos desenvolvedores trocar facilmente entre diferentes modelos (sejam eles da OpenAI, Cohere, Hugging Face ou locais via Ollama).

Essas ferramentas geralmente possuem duas funções distintas para *embeddings*:

- **`embed_documents`**: Usada para criar os vetores para os documentos que serão

[1] https://www.langchain.com/
[2] https://www.llamaindex.ai/

armazenados (a base de conhecimento).

- **`embed_query`**: Usada para criar o vetor para a pergunta do usuário.

Essa distinção existe porque alguns modelos são otimizados de forma diferente para criar vetores de documentos (que precisam ser bons em *serem encontrados*) versus vetores de consultas (que precisam ser bons em *encontrar* outros vetores).

3 TRABALHOS RELACIONADOS

A aplicação de Inteligência Artificial na Engenharia de Software, especificamente na automação de testes, tem sido objeto de diversos estudos recentes. Esta seção discute trabalhos que investigam o uso de Grandes Modelos de Linguagem (LLMs) e arquiteturas RAG (*Retrieval-Augmented Generation*) para mitigar desafios na geração e manutenção de testes.

3.1 Uso de IA em Testes de Software

Uma investigação fundamental sobre a capacidade de LLMs na geração de testes foi conduzida por Guilherme e Vincenzi (GUILHERME; VINCENZI, 2023). Os autores avaliaram o desempenho do ChatGPT (`gpt-3.5-turbo`) na criação de testes de unidade para programas Java, comparando-o com ferramentas tradicionais como o EvoSuite. O estudo concluiu que, embora os LLMs demonstrem um potencial comparável às ferramentas de mercado, a qualidade do *prompt* é um fator determinante para o sucesso.

Prompts genéricos resultaram em baixas taxas de compilação, enquanto instruções detalhadas (engenharia de *prompt*) elevaram significativamente a cobertura de código e a pontuação de mutação. A Figura 3.1 mostra um exemplo de prompt simples, enquanto a Figura 3.2 demonstra o prompt detalhado utilizado no estudo. A relevância deste estudo para o presente trabalho reside na validação dos LLMs como ferramentas viáveis de teste, ao mesmo tempo que evidencia a necessidade de fornecer contexto rico — uma lacuna que nossa proposta busca preencher via RAG.

Figura 3.1: Prompt v1 Guilherme e Vicenzi

```
Generate test cases just for the {cut}
Java class in one Java class file with
imports using JUnit 4 and Java 8:

{code}
```

Fonte: (GUILHERME; VINCENZI, 2023)

3.2 IA e RAG Aplicados a Testes de Software

Avançando para abordagens que utilizam contexto externo, Wang et al. (WANG; GUO; TAN, 2025) propuseram o "Copilot for Testing", um sistema que integra detecção de *bugs* e geração de testes utilizando uma arquitetura RAG baseada em contexto. Diferente de abordagens estáticas, o trabalho de Wang modela a base de código como um grafo dinâmico, atualizando os *embeddings* conforme o código evolui e o cursor do desenvolvedor se move (Figura 3.3). Essa estratégia resultou em uma melhoria de 31,2% na precisão da detecção de *bugs*.

Embora Wang et al. validem o uso de RAG para testes, este trabalho se diferencia pela sua aplicação e restrições de arquitetura. Enquanto Wang foca na detecção síncrona de *bugs* com um grafo dinâmico, nossa pesquisa propõe um RAG focado na redução do débito técnico de testes E2E já existente, utilizando uma base de conhecimento híbrida (código-fonte e documentação técnica da ferramenta Maestro). Além disso, diferentemente das soluções em nuvem comuns na literatura, nossa implementação prioriza a privacidade e a segurança através do uso de modelos locais via Ollama, atendendo a restrições corporativas reais.

Figura 3.2: Prompt v2 Guilherme e Vicenzi

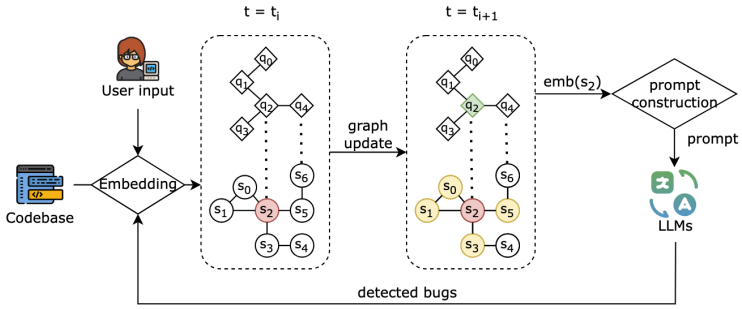
```
I need functional test cases to cover all
decisions in the methods of the class
under testing.
All conditional expressions must assume
true and false values.
Tests with Boundary Values are also
mandatory. For numeric data, always use
positive and negative values.
All tests must be in one Java class file.
Include all necessary imports.
It is mandatory to throws Exception
in all test method declarations.
It is mandatory to include timeout=1000
in all @Test annotations.
It is mandatory to test for the default
constructor.
Each method in the class under test must
have at least one test case.
Even simple or void methods must have a
test calling it with valid inputs.
@Test(expected= must be used only if the
method under testing explicitly throws
an exception.
Test must be in JUnit 4 framework format.
Test set heather package and import
dependencies:
package ds;
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.*;
import ds.*;
The class under testing is {clazz}.
The test class must be {cut}Test

Class under testing
*******************

{code}
```

Fonte: (GUILHERME; VINCENZI, 2023)

Figura 3.3: RAG baseado em contexto



Fonte: Wang et al.

4 METODOLOGIA

Esta seção detalha os procedimentos adotados para a realização deste trabalho, que visa desenvolver e avaliar um sistema RAG (Retrieval-Augmented Generation) para auxiliar na redução do débito técnico de testes.

4.1 Tipo de Pesquisa

A presente pesquisa classifica-se, quanto à sua natureza, como aplicada, e, quanto aos seus objetivos, como exploratória, sendo estruturada sob o método de pesquisa-ação. Caracteriza-se como aplicada por buscar o desenvolvimento de uma solução tecnológica concreta - o protótipo RAG - para mitigar o problema do débito técnico de testes em um ambiente operacional real. A dimensão exploratória justifica-se pela investigação de tecnologias emergentes, como Grandes Modelos de Linguagem (LLMs) e arquiteturas de recuperação, aplicadas à garantia de qualidade de software. Por fim, o estudo configura-se como um estudo de caso ao analisar profundamente o contexto de uma *startup* de tecnologia específica.

4.2 Etapas do Desenvolvimento

O trabalho foi estruturado nas seguintes etapas: diagnóstico do problema, coleta e preparação de dados, implementação do protótipo RAG e avaliação dos resultados.

4.3 Diagnóstico e Contextualização do Problema

A primeira etapa consistiu em diagnosticar o ambiente e o problema. O estudo foi conduzido em um projeto de aplicativo móvel desenvolvido em React Native e TypeScript, no qual o autor atua profissionalmente. O aplicativo é focado na gestão e acompanhamento de eventos e apresenta as seguintes funcionalidades principais:

- Telas de autenticação, login e redefinição de senha.
- Processo de *onboarding* exibido na primeira execução.
- Listas de eventos, disponíveis, em andamento ou finalizados, com telas de filtros e

de detalhes.

- Funcionalidades de interação no evento, como registro de disponibilidade, *check-in/check-out*, envio de fotos e resposta a tarefas.
- Tela de perfil para atualização de dados pessoais, imagem, preferências e configuração de login biométrico.

Foi identificado que o projeto apresenta um elevado débito técnico relacionado à ausência de testes automatizados, decorrente da despriorização dessa atividade no cronograma. Os testes são realizados manualmente por uma equipe de QA reduzida, que também é responsável pela validação de outras frentes no projeto. Essa lacuna já resultou em falhas críticas (*crashes*) que poderiam ter sido evitadas por *smoke tests* e aumenta o esforço necessário para testes de regressão à medida que o sistema evolui.

4.4 Justificativa da Abordagem Técnica

A implementação de um sistema RAG para a geração de testes *end-to-end* (E2E) fundamenta-se na necessidade de gerar código contextualizado e seguro. A opção por modelos locais, executados via plataforma Ollama, foi motivada por dois pilares estratégicos:

1. **Segurança e Conformidade:** Em virtude das políticas restritivas da organização quanto ao uso de serviços de IA de terceiros, a execução local previne o tráfego de código-fonte confidencial para servidores externos, mitigando riscos de vazamento de dados proprietários.
2. **Desempenho e Viabilidade:** Testes preliminares indicaram que modelos de código aberto, como o Granite e o Gemma3, apresentam latência e qualidade de resposta satisfatórias para a automação de scripts de teste quando operados em infraestrutura local dedicada.

4.4.1 Coleta e Preparação de Dados (Base de Conhecimento)

A constituição da base de conhecimento envolveu a extração de dados de duas fontes primárias:

1. **Código-Fonte do Projeto:** Utilizou-se a base de código do aplicativo em React

Native, processando arquivos com extensões `.ts` e `.tsx`. Foram aplicados filtros de exclusão em diretórios sem relevância semântica para testes, tais como `node_modules`, `android` e `ios`.

2. **Documentação Técnica:** Implementou-se um *crawler* em linguagem Python, utilizando a biblioteca Crawlee, para a extração automatizada de conteúdos da documentação oficial da ferramenta Maestro.

Esses dados foram processados, divididos em blocos (*chunks*) e vetorizados usando um modelo de *embedding* para criar o índice (banco de vetores) que o RAG consulta.

4.4.2 Implementação e Experimentação do Protótipo

O protótipo RAG foi desenvolvido em Python, utilizando o *framework* LangChain para orquestrar o *pipeline* e o Ollama para executar os modelos de LLM e *embedding* localmente.

Uma etapa metodológica crucial foi a experimentação e o ajuste dos componentes do RAG para otimizar os resultados. Foram testadas diferentes combinações de:

- **LLMs:** (ex: `deepseek-coder:6.7b`, `ibm/granite4.0-preview:tiny`, `gemma3:4b`).
- **Modelos de Embedding:** (ex: `granite-embedding:278m`, `embeddinggemma:300m`).
- **Parâmetros de Divisão de Texto:** (ex: *chunk size* de 1000, 512; *chunk overlap* de 1000, 100, 60, 128).
- **Engenharia de Prompt:** O *prompt* do sistema foi refinado para especializar o assistente (ex: "especializado em React, React Native, Typescript e Maestro") e controlar o formato das saídas.

4.4.3 Avaliação

A avaliação da eficácia do protótipo RAG foi planejada em duas frentes:

- **Análise Quantitativa:** Medição do impacto da ferramenta em métricas de débito técnico, como o aumento percentual da cobertura de testes.
- **Análise Qualitativa:** Avaliação da qualidade dos testes gerados, considerando a

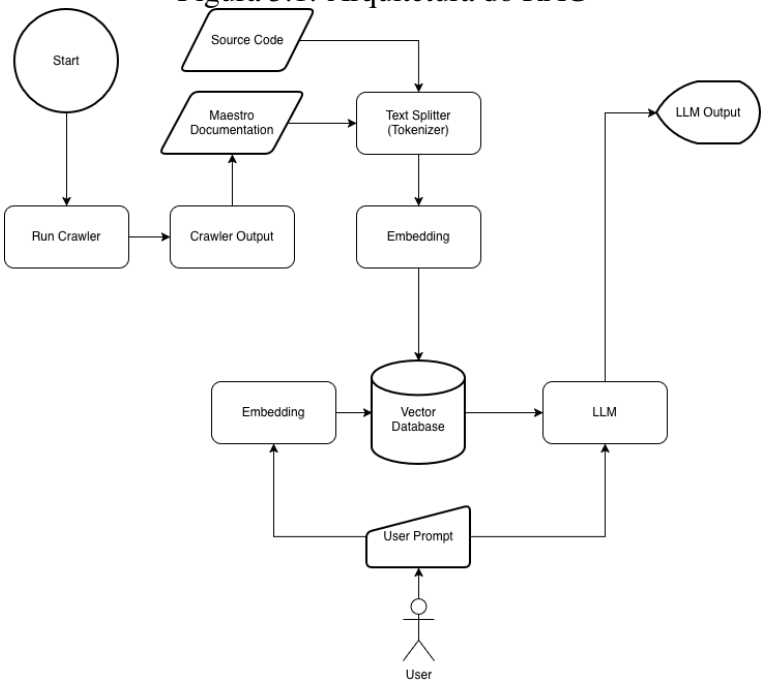
legibilidade e a manutenibilidade do código produzido.

5 IMPLEMENTAÇÃO

5.1 Arquitetura da Solução

Este capítulo detalha a construção e a descrição dos componentes do RAG.

Figura 5.1: Arquitetura do RAG



Fonte: O autor

A arquitetura do sistema implementado compreende um *pipeline* dividido em duas fases operacionais: indexação e recuperação. O fluxo inicia-se com a ingestão do código-fonte e da documentação técnica, processados respectivamente pelo `DirectoryLoader` do LangChain e pelo *crawler* customizado. Após a etapa de *chunking*, os dados são vetorizados e armazenados em um banco de dados persistente Chroma. Na fase de recuperação, o *prompt* do usuário é submetido a uma busca de similaridade semântica, cujo contexto resultante é injetado no LLM via `ChatOllama` para a geração do script de teste final.

A arquitetura do protótipo, ilustrada na Figura 5.1, segue um fluxo padrão de RAG, adaptado para as necessidades deste trabalho. O processo é dividido em duas fases: indexação (criação da base de conhecimento) e recuperação (geração de resposta).

O fluxo de dados da arquitetura implementada é o seguinte:

1. **Coleta de Dados:** O processo inicia com a coleta de dados de duas fontes distintas: a documentação oficial da ferramenta Maestro e o código-fonte do projeto em React Native.

2. **Processamento (Crawl e Carga):** Um *crawler* processa a documentação do Maestro. O código-fonte (`.ts`, `.tsx`) é carregado diretamente do disco usando o `DirectoryLoader` do LangChain, conforme (SILVA, 2025).
3. **Divisão (Chunking):** Os documentos e o código são passados para um `TextSplitter` (`RecursiveCharacterTextSplitter`), que divide os textos longos em pedaços menores (*chunks*).
4. **Vetorização e Indexação:** Um modelo de *embedding* (ex: `granite-embedding:278m`) converte cada *chunk* de texto em um vetor numérico. Esses vetores são armazenados em um banco de dados vetorial `Chroma`, criando um índice persistente.
5. **Recuperação e Geração:** Quando o usuário envia um *prompt*, o sistema primeiro vetoriza essa pergunta. Em seguida, ele busca no `Vector Database` os *chunks* semanticamente mais similares.
6. **Saída do LLM:** Os *chunks* recuperados (o "contexto") são inseridos no *prompt* final, que é então enviado ao LLM (`ChatOllama`) para gerar a resposta.

5.1.1 Base de Conhecimento e Processo de Indexação

A eficácia de um sistema RAG depende diretamente da qualidade de sua base de conhecimento. Para este trabalho, a base foi composta por duas fontes de dados cruciais:

- **Código-Fonte do Projeto:** Todo o código TypeScript (`.ts` e `.tsx`) do projeto React Native foi utilizado como fonte. Conforme detalhado no código em (SILVA, 2025), foram excluídos diretórios que não agregam valor semântico para a geração de testes, como `node_modules`, `android` e `ios`.
- **Documentação da Ferramenta:** Foi executado um *crawler* para extrair todo o conteúdo textual da documentação oficial da ferramenta Maestro.

Para garantir a eficiência operacional e evitar o reprocessamento redundante, o protótipo utiliza um mecanismo de atualização incremental baseado em um arquivo de manifesto (`manifest.json`). Este componente monitora o estado de integridade dos arquivos na base de conhecimento, permitindo que apenas as modificações recentes no código-fonte sejam vetorizadas e integradas ao banco de dados vetorial a cada execução.

5.1.2 Experimentação e Evolução dos Componentes (Prompts e Modelos)

A definição dos componentes do *pipeline* (LLM, modelo de *embedding* e parâmetros do *splitter*) não foi trivial e exigiu um processo iterativo de experimentação, conforme descrito nas notas de implementação.

5.1.2.1 Iteração 1: Configuração Inicial

Para a validação do protótipo proposto, a base de conhecimento do RAG foi composta por aproximadamente 300 arquivos de código-fonte (.ts e .tsx) e 125 arquivos extraídos da documentação técnica do Maestro. A avaliação utilizou o modelo `gemma3:4b`, identificado em etapas anteriores como o melhor equilíbrio entre latência e precisão.

Previamente à análise principal, conduziu-se um teste de controle sem o auxílio da arquitetura RAG. Nesta etapa, o modelo gerou scripts em linguagem Python utilizando bibliotecas incompatíveis com o projeto, o que evidenciou a ineficácia de LLMs genéricos na ausência de contexto específico para ferramentas de automação móvel.

5.1.2.2 Iteração 2: Troca de Componentes

Seguindo a sugestão da orientação, os modelos foram alterados para `LLM: ibm/granite4.0-preview:tiny` e `Embedding Model: embeddinggemma:300m`, com `chunk_size=512` e `chunk_overlap=100`. A resposta para o mesmo *prompt* melhorou, mas foi genérica, afirmando que o arquivo "provavelmente contém um React hook customizado".

5.1.2.3 Iteração 3: Otimização da Combinação

Alterando o gerador `LLM: gemma3:4b`, o modelo de *embedding* foi revertido para `granite-embedding:278m`. Esta combinação provou ser a ideal. A resposta ao *prompt* foi precisa e altamente detalhada, descrevendo corretamente o gerenciamento de estado (`useState`), as operações assíncronas (`useMutation`), o tratamento de erros e a lógica de *refetching* de dados.

5.1.2.4 Iteração 4: Teste de Contexto (Documentação)

Com os modelos validados, o foco mudou para a recuperação de dados da documentação do Maestro. Foi definido um *system prompt* especializado e feita a pergunta: `give me a list of maestro commands...`. O sistema falhou, alucinando comandos como `run_cloud_tests` e afirmando que a informação não estava no contexto.

5.1.2.5 Iteração 5: Ajuste Fino do Splitter

O problema da Iteração 4 foi identificado como um erro na recuperação de contexto. O `TextSplitter` foi ajustado de `chunk_overlap=60` para `chunk_overlap=128`. Com essa mudança, o mesmo *prompt* sobre o Maestro retornou uma lista correta e exaustiva de comandos (ex: `assertVisible`, `tapOn`, `inputText`, `scrollUntilVisible`, etc.).

Este processo validou a configuração final do *pipeline* e permitiu que o foco do trabalho avançasse para a engenharia de *prompt* focada na geração de testes.

5.1.3 Desafios e Decisões Técnicas

A implementação apresentou desafios técnicos que exigiram decisões específicas:

- **Ajuste de Hiperparâmetros do RAG:** Como demonstrado na seção anterior, a seleção de `chunk_size` e `chunk_overlap` não é trivial. Um *overlap* muito pequeno (ex: 60) falhou em recuperar informações de comandos do Maestro que poderiam estar divididas entre *chunks*, um problema que um *overlap* maior (128) resolveu.
- **Controle de Alucinação:** Para um sistema RAG, a fidelidade ao contexto é mais importante que a criatividade. Por isso, o parâmetro `temperature` do LLM foi mantido em `0` na maioria dos testes, garantindo respostas mais determinísticas e factuais.
- **Especialização do Prompt:** O *prompt* inicial do LangChain é genérico. Foi necessário criar um *system prompt* especializado (`You are an assistant... specialized in React, React Native, Typescript and Maestro...`) para forçar o LLM a priorizar o contexto fornecido e a atuar como um especialista.

- **Fluxo de Desenvolvimento:** A integração com o IDE (como VSCode) foi considerada, mas classificada como um trabalho futuro. A implementação atual funciona como uma ferramenta de linha de comando (CLI), onde o desenvolvedor interage via terminal. O código em (SILVA, 2025) reflete isso através do *loop* `while True:` que aguarda pelo `input()` do usuário.

5.1.4 Análise de Performance de Modelos Locais

Um dos principais requisitos deste trabalho é a utilização de modelos de LLM que possam ser executados localmente, via Ollama, para garantir a privacidade do código-fonte da empresa. Esta restrição impõe um desafio de engenharia: encontrar um modelo que ofereça o melhor **equilíbrio entre capacidade de raciocínio (qualidade) e performance de inferência (velocidade)**.

Para um sistema RAG, a performance é medida em duas métricas principais:

- **Prompt Eval Rate (Velocidade de Processamento):** Medido em *tokens*/segundo (t/s), indica a rapidez com que o modelo "lê"e processa o *prompt* de entrada, que inclui o contexto recuperado. Uma taxa alta é crucial para que o RAG não se torne lento ao processar grandes volumes de contexto (código e documentação).
- **Eval Rate (Velocidade de Geração):** Também em t/s, mede a rapidez com que o modelo gera a resposta (o código de teste). Esta é a velocidade percebida pelo usuário.

Foram realizados testes de *benchmark* em diversos modelos de código aberto disponíveis no Ollama para avaliar sua viabilidade. A Tabela 5.1 apresenta um subconjunto representativo dos resultados obtidos, destacando a relação entre o tamanho do modelo (em bilhões de parâmetros) e sua performance de geração.

Também foi feito um experimento com um mesmo prompt para as LLMs acima com variação do modelo de embedding entre `granite-embedding:278m`, `embeddinggemma:300m` e `qwen3-embedding:0.6b`. O prompt utilizado foi:

- *System Prompt: "You are an assistant for question-answering tasks specialized in React, React Native, Typescript and Maestro. Use the following pieces of retrieved context to answer the question. If you don't know the answer, just say that you don't know. Provide full code examples when asked, without placeholders like '...'. When asked about Maestro, answer only with commands that are given on the context"*

Tabela 5.1: Benchmark de performance de LLMs locais via Ollama.

Modelo (Ollama ID)	Tamanho (Parâmetros)	Proc. Prompt (t/s)	Geração (t/s)
`granite4:350m`	0.35B	∼4270 t/s	∼195 t/s
`deepseek-coder:1.3b`	1.3B	∼1855 t/s	∼148 t/s
`qwen2.5-coder:3b`	3B	∼365 t/s	∼73 t/s
`ibm/granite4.0-preview:tiny`	∼1B	∼189 t/s	∼101 t/s
`gemma3:4b`	4B	∼155 t/s	∼52 t/s
`deepseek-coder:6.7b`	6.7B	∼323 t/s	∼32 t/s
`llama3:8b`	8B	∼102 t/s	∼34 t/s
`gemma3:12b`	12B	∼34 t/s	∼17 t/s
`gpt-oss:20b`	20B	∼72 t/s	∼9 t/s

Fonte: Dados de benchmark coletados pelo autor no equipamento de teste (MacBook Pro, M4 Pro, 24GB). As taxas de t/s são aproximadas, baseadas na média das últimas execuções de cada teste.

- *User Prompt: "What are all maestro commands?"*

O resultado "alvo" é que o modelo liste todos os 41 comandos que o Maestro possui no momento do teste. É possível visualizar todos os resultados deste prompt no arquivo "all_content.txt" do repositório do projeto[1].

Tabela 5.2: Embedding Model vs. LLM Performance Score (0–3)

	LLM Model								
Embedding Model	`gemma3n:e4b`	`granite-code:8b`	`gemma3:12b`	`gemma3:4b`	`granite4:1b`	`ibm/granite4.0-preview:tiny`	`granite4:350m`	`granite4:3b`	`granite4:tiny-h`
`granite-embedding:278m`	1	1	3	3	2	1	0	1	1
`embeddinggemma:300m`	2	3	3	3	3	1	0	2	1
`qwen3-embedding:0.6b`	2	0	3	1	2	0	0	2	2

Score Key: 3 = Excelente, 2 = Bom, 1 = Ruim, 0 = Falha/Sem Resposta

Fonte: Dados de benchmark coletados pelo autor no equipamento de teste (MacBook Pro, M4 Pro, 24GB).

Onde:

- 0: demonstrou algum tipo de falha na saída, disse que não sabia ou a resposta estava corrompida
- 1: problemas para listar os comandos ou produziu valores confusos

[1] (SILVA, 2025)

- 2: listou pelo menos 35 comandos
- 3: todos os comandos esperados foram listados

5.1.4.1 Justificativa da Escolha

Os dados da Tabela 5.1 ilustram um claro *trade-off*:

- **Modelos Pequenos (0.35B - 1.3B):** Apresentam velocidades de geração e processamento extremamente altas (acima de 140 t/s), mas sua capacidade de raciocínio é limitada, tendendo a produzir código de baixa qualidade ou alucinações.
- **Modelos Grandes (12B - 20B):** Possuem maior capacidade de raciocínio, porém sua performance de geração local é muito baixa (9 a 17 t/s), tornando-os inviáveis para um fluxo de trabalho interativo.
- **Modelos Médios (3B - 8B):** Representam o ponto de equilíbrio ideal para este trabalho. Modelos como `gemma3:4b` e `deepseek-coder:6.7b` mostraram uma excelente velocidade de processamento de *prompt* (155-323 t/s), essencial para o RAG, e uma velocidade de geração (32-52 t/s) que é rápida o suficiente para o uso prático.

Com base nessa análise, os modelos `gemma3:4b` e `gemma3:12b` foram selecionados para as experimentações detalhadas na Seção 5.1.2, pois oferecem o melhor balanço entre performance de execução local e capacidade de geração de código. E o modelo de embedding `qwen3-embedding:0.6b` foi descartado.

6 AVALIAÇÃO E RESULTADOS

6.1 Configuração Experimental

Para a validação do protótipo proposto, a base de conhecimento do RAG foi alimentada com aproximadamente 300 arquivos de código-fonte (extensões `.ts` e `.tsx`) do projeto e 125 arquivos resultantes do *crawler* da documentação do Maestro. A avaliação utilizou o modelo `gemma3:4b`, selecionado nas etapas anteriores de implementação. Inicialmente, realizou-se um teste de controle sem o uso de RAG (descrito no Apêndice 9.9), onde o modelo, desprovido de contexto, gerou código em Python utilizando uma biblioteca incorreta, demonstrando a ineficácia de LLMs genéricos para esta tarefa específica sem o devido contexto.

6.2 Análise das Métricas de Resultado

Os resultados obtidos validam a hipótese de que um sistema RAG local é uma estratégia viável para a redução do débito técnico de testes em ambientes corporativos. A principal contribuição deste trabalho reside no "empoderamento"da equipe de QA: profissionais sem especialização profunda em desenvolvimento tornam-se capazes de gerar automações complexas a partir de requisitos em linguagem natural.

Diferente das abordagens baseadas em nuvem, a utilização de modelos locais via Ollama provou ser uma solução que concilia produtividade e segurança da informação. Esta arquitetura atende às exigências de confidencialidade da organização, demonstrando que é possível extrair valor de Grandes Modelos de Linguagem sem comprometer a integridade da base de código proprietária.

A eficácia da ferramenta foi mensurada pelo impacto na cobertura de fluxos críticos e na eficiência temporal do desenvolvimento. Realizou-se um levantamento de 62 cenários de uso do aplicativo, dos quais apenas 16 (aproximadamente 26%) possuíam cobertura E2E antes da intervenção. Com a aplicação do sistema RAG, a cobertura foi expandida para 42 fluxos, atingindo um patamar de 68%. Este incremento demonstra que a ferramenta mitiga a barreira técnica inicial, facilitando a criação de novos casos de teste.

No que tange ao esforço de desenvolvimento, observou-se uma redução expressiva no tempo de escrita. Enquanto a elaboração manual de fluxos complexos, como o "Primeiro Acesso", demandava cerca de uma hora de análise técnica, a utilização do pro-

Tabela 6.1: Matriz de Rastreabilidade - Login, Onboarding e Primeiro acesso

Fluxo	**Descrição**	**Criticidade**	**Antes**	**Depois**
Login (Success)	Usuário insere credenciais válidas e acessa o sistema.	Crítica	✓	✓
Password Recovery (Valid Email)	Usuário solicita a redefinição de senha com um e-mail válido e o link é enviado.	Crítica	×	✓
Password Recovery (Resend Link)	Usuário solicita o reenvio do link de recuperação de senha.	Alta	×	×
Password Recovery (Open Email Client)	Usuário clica no botão "Abrir e-mail"após solicitar a redefinição.	Média	×	×
Onboarding (Button Navigation)	Usuário visualiza as telas de onboarding pela primeira vez, avançando pelos botões.	Crítica	✓	✓
First Access (Complete Profile)	Usuário loga pela primeira vez e preenche dados de perfil obrigatórios.	Crítica	×	✓
Terms & Conditions (From Login)	Usuário clica no link de Termos e Condições na tela de Login e visualiza o conteúdo.	Média	×	×

Fonte: O autor

Tabela 6.2: Tempo para escrever e ajustar os testes manualmente

Fluxo	**Tempo (minutos)**	Tempo ajustes (minutos)
Edit Profile	12	19
View Booked Event	3	12
First Access Flow	4	8

Fonte: O autor

tótipo permitiu que o esforço humano fosse deslocado da escrita estrutural para a revisão e o ajuste fino, otimizando o ciclo de QA.

Uma parte dos fluxos cobertos e sua criticidade pode ser vista na tabela 6.1.

Os demais fluxos podem ser conferidos nos apêndices 9.1, 9.2, 9.3 e 9.4.

Em relação ao esforço temporal, a escrita manual dos passos de teste para fluxos complexos, como o "Primeiro Acesso", demandou cerca de uma hora de dedicação exclusiva para análise de código e escrita. Em contrapartida, fluxos subsequentes, como o "Edit Profile", beneficiaram-se do reaproveitamento de código, reduzindo o tempo de escrita manual para cerca de 12 minutos, com mais 19 minutos dedicados a ajustes e correções. A Tabela 6.2 detalha os tempos aferidos.

6.2.1 Análise Qualitativa: Testes Gerados vs. Corrigidos

As saídas podem ser consultadas no repositório do projeto (SILVA, 2025). A comparação entre o código gerado pelo RAG e a versão final funcional (corrigida manualmente) revelou limitações importantes do protótipo atual. A análise dos *diffs* (Figuras 6.1 a **??**) indicou que, embora a lógica geral dos testes fosse gerada corretamente, o modelo apresentou dificuldades em duas categorias principais: complexidade semântica e sintaxe.

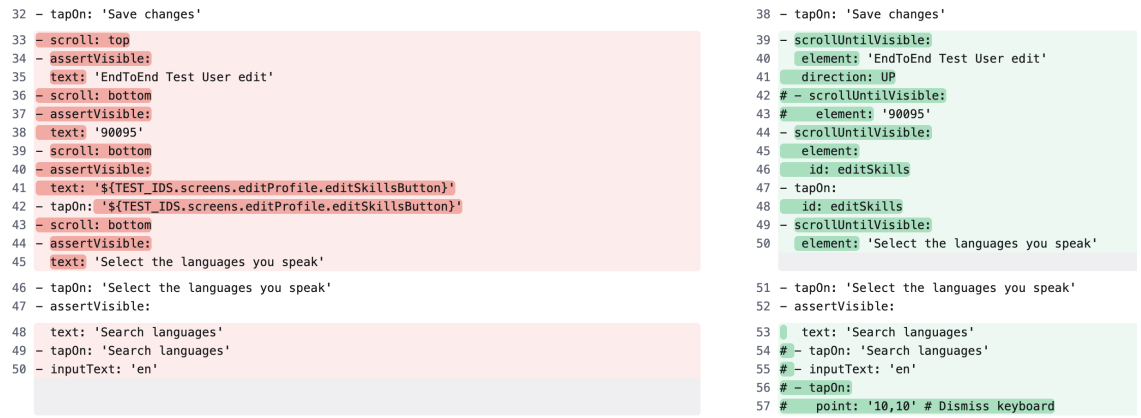
Figura 6.1: Diff Edit Profile(1)

```
- text: '${TEST_IDS.screens.editProfile.editPersonalInfoButton}'
- tapOn: '${TEST_IDS.screens.editProfile.editPersonalInfoButton}'
- assertVisible:
- text: 'EndToEnd Test User'
- tapOn: 'EndToEnd Test User'
- inputText: ''
- inputText: 'EndToEnd Test User edit'
- tapOn: '10,10' # Dismiss keyboard
- scroll: 100
- assertVisible:
- text: 'UCLA'
- tapOn: 'UCLA'
- assertVisible:
- text: 'Search address'
- tapOn: 'Search address'
- inputText: 'UCLA Luskin'
- assertVisible:
- text: 'UCLA Luskin School of Public Affairs'
- tapOn: 'UCLA Luskin School of Public Affairs'
- tapOn: 'Confirm'
- assertVisible:
- text: '90095'
- scroll: bottom
- assertVisible:
- text: 'Save changes'
```

```
+ id: editPersonalInfo
+ tapOn:
+ id: editPersonalInfo
+ assertVisible:
+ text: 'EndToEnd Test User'
+ tapOn: 'EndToEnd Test User'
+ eraseText
+ inputText: 'EndToEnd Test User edit'
+ tapOn:
+ point: '10,10' # Dismiss keyboard
+ scrollUntilVisible:
+ element:
+ id: 'addressButton'
+ tapOn:
+ id: 'addressButton'
# + assertVisible:
#   text: 'Search address'
# + tapOn: 'Search address'
# + inputText: 'UCLA Luskin'
# + assertVisible:
#   text: 'UCLA Luskin School of Public Affairs'
# + tapOn: 'UCLA Luskin School of Public Affairs'
# + tapOn: 'Confirm'
# + assertVisible:
#   text: '90095'
+ scrollUntilVisible:
+ element: 'Save changes'
```

Fonte: O autor

Figura 6.2: Diff Edit Profile(2)



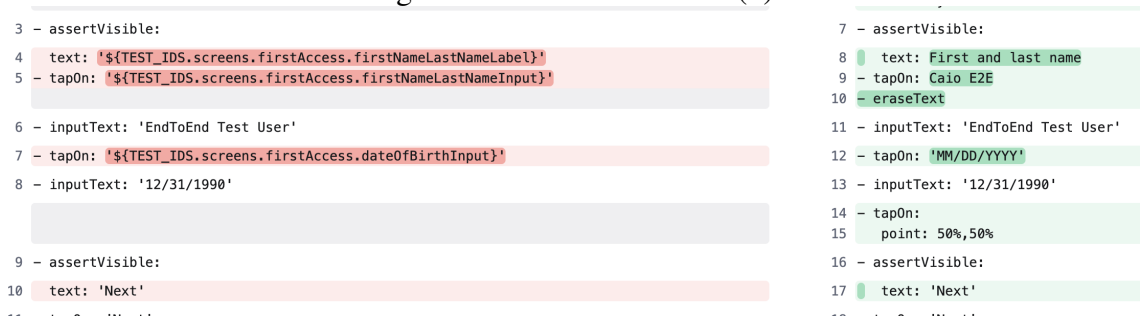
```
- tapOn: 'Save changes'
- scroll: top
- assertVisible:
- text: 'EndToEnd Test User edit'
- scroll: bottom
- assertVisible:
- text: '90095'
- scroll: bottom
- assertVisible:
- text: '${TEST_IDS.screens.editProfile.editSkillsButton}'
- tapOn: '${TEST_IDS.screens.editProfile.editSkillsButton}'
- scroll: bottom
- assertVisible:
- text: 'Select the languages you speak'
- tapOn: 'Select the languages you speak'
- assertVisible:
- text: 'Search languages'
- tapOn: 'Search languages'
- inputText: 'en'
```

```
+ tapOn: 'Save changes'
+ scrollUntilVisible:
+ element: 'EndToEnd Test User edit'
+ direction: UP
# + scrollUntilVisible:
#   element: '90095'
+ scrollUntilVisible:
+ element:
+ id: editSkills
+ tapOn:
+ id: editSkills
+ scrollUntilVisible:
+ element: 'Select the languages you speak'
+ tapOn: 'Select the languages you speak'
+ assertVisible:
+ text: 'Search languages'
# + tapOn: 'Search languages'
# + inputText: 'en'
# + tapOn:
#   point: '10,10' # Dismiss keyboard
```

Fonte: O autor

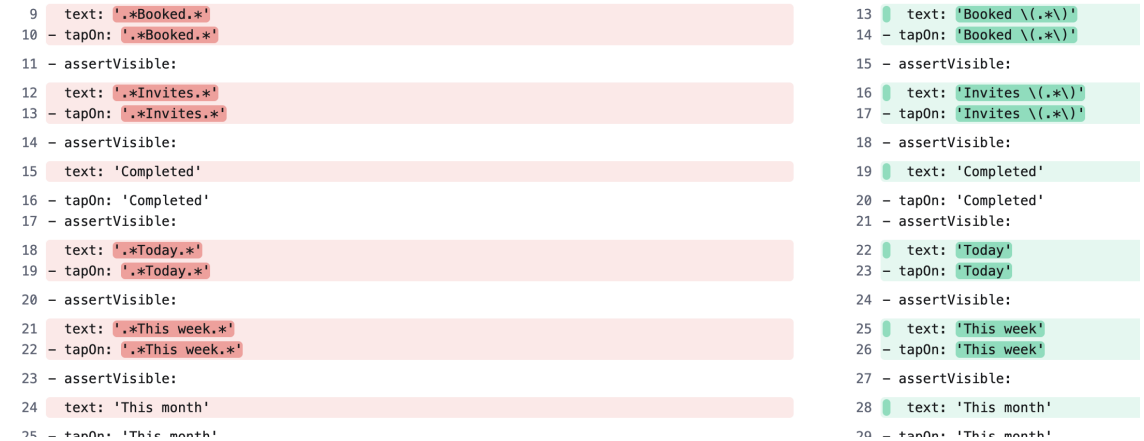
A análise comparativa entre o código gerado e a versão funcional corrigida revelou limitações inerentes ao protótipo atual. Observou-se que o sistema apresenta dificuldades

Figura 6.3: Diff First Access(1)



Fonte: O autor

Figura 6.4: Diff View Booked Event(1)

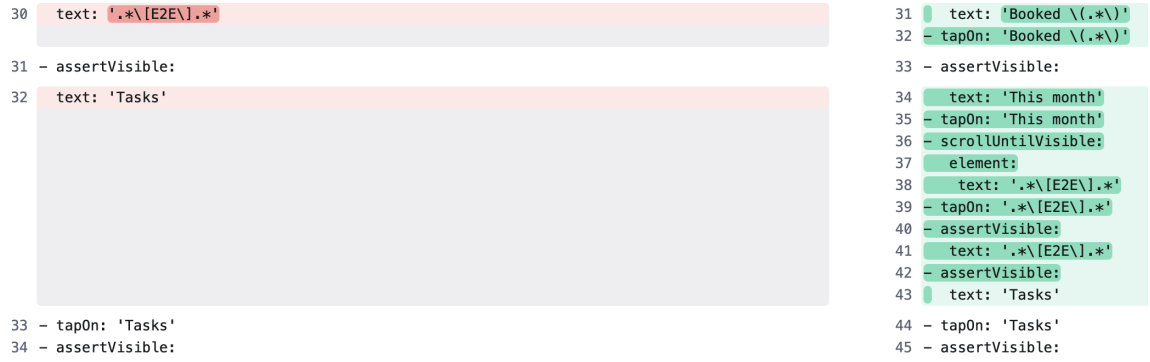


Fonte: O autor

em lidar com complexidades semânticas avançadas, como o uso de expressões regulares (*regex*) para seletores dinâmicos e comandos de rolagem condicional (`scrollUntilVisible`).

Ademais, recorrentes erros de indentação no formato YAML exigiram intervenção manual. Tais achados sugerem que, embora o RAG atue como um assistente eficiente na aceleração da escrita, a supervisão técnica permanece indispensável para garantir a execução correta dos *scripts* e a integridade da suíte de testes.

Figura 6.5: Diff View Booked Event(2)



Fonte: O autor

7 DISCUSSÃO

O protótipo desse trabalho foi apresentado como proposta de melhoria de processo de testes de uma *startup* de 150 funcionários. Os resultados obtidos, especialmente a viabilidade de usar RAG local para gerar testes E2E, também serão compartilhados em um evento técnico organizado pelo Google, reforçando o potencial de aplicação prática e disseminação do conhecimento obtido.

Esta seção discute a interpretação dos resultados apresentados no Capítulo 6, focando em onde o protótipo RAG teve sucesso, quais foram suas limitações e como essas descobertas se posicionam em relação à literatura.

A principal contribuição do trabalho foi a validação empírica de que um RAG local, executado via Ollama, é uma estratégia viável para reduzir o débito técnico de testes. O aumento da cobertura de fluxos de $\sim$ 26% (16 de 62 fluxos) para $\sim$ 68% (42 de 62 fluxos), com um esforço de implementação focado na ferramenta e não na escrita manual de cada teste, demonstra a eficácia prática da abordagem.

7.1 Onde o RAG funcionou bem: O Tradutor de QA para Código

A eficácia do protótipo reside em sua capacidade de atuar como um "tradutor"entre a lógica de negócio, descrita em linguagem natural, e a sintaxe técnica da ferramenta Maestro. A maior barreira identificada na Introdução 1 foi o fato de que a equipe de QA não tem conhecimento em ferramentas de teste móvel, e a equipe de desenvolvimento não tem experiência em planejar os casos de teste.

O RAG resolveu exatamente essa lacuna:

- **Empoderamento do QA:** Uma importante constatação é que um profissional sem especialização na área de desenvolvimento ou não tem conhecimento do código seria capaz de elaborar um *prompt* com os passos necessários (conforme Apêndice 9.1) e gerar um resultado muito próximo do que seria escrito por alguém que tem a documentação aberta ou conhece os comandos e o código.
- **Geração Contextualizada:** O RAG, por ter acesso à documentação do Maestro, gerou os comandos corretos (ex: `assertVisible`, `tapOn`), algo que uma pessoa de QA levaria tempo para aprender.
- **Validação da Arquitetura RAG:** A importância do RAG é evidenciada pelo teste

"sem RAG"(Apêndice 9.9). Ao receber o *prompt* do Onboarding sem contexto, o LLM alucinou e gerou um código em Python, totalmente inútil para o projeto. O RAG, por outro lado, usou o contexto da documentação para gerar o `YAML` correto.

Este ponto é reforçado ao analisar a diferença de tempo. Conforme a Tabela 6.2, o fluxo `Edit Profile` exigiu 12 minutos para a escrita manual do código YAML e 19 minutos para ajustes. O RAG elimina os 12 minutos de escrita inicial, substituindo-os pelo tempo de geração. Embora o RAG ainda exija ajuste manual (discutido a seguir), o esforço para corrigir a indentação ou trocar um comando (Figuras 6.1 a 6.5) é significativamente menor do que escrever o fluxo completo do zero.

7.2 Limitações Identificadas e a Necessidade de Co-piloto

A avaliação dos resultados também expôs que o RAG, na sua forma atual, não é uma ferramenta de "um clique", mas sim um "co-piloto"que exige supervisão humana. O trabalho teve algumas limitações perceptíveis nos testes. As limitações se dividem em duas categorias:

1. **Limitações de Escopo (Inteligência):** O RAG falhou em responder perguntas abertas como "Escreva um teste usando o Maestro para a tela Login.tsx". Mesmo fornecendo o arquivo como contexto, as respostas foram genéricas. Isso indica que o RAG atual não cria a *estratatégia* de teste do zero (o que testar); ele é excelente em *automatizar* uma estratégia que já foi definida pelo humano (os passos do Apêndice 9.1).
2. **Limitações de Geração (Sintaxe e Semântica):** A análise dos *diffs* (Figuras 6.1 a 6.5) mostrou falhas recorrentes. Conforme descrito nos resultados (Seção 6.2.1), o RAG teve dificuldade com:
    - **Complexidade Semântica:** Gerar comandos avançados como `scrollUntilVisible` e o uso correto de regex.
    - **Erro de Sintaxe:** A falha mais comum foi a indentação incorreta do `YAML` para comandos com parâmetros.

Essas limitações, no entanto, não invalidam a abordagem. Elas reforçam que o tempo de ajuste manual (ex: os 19 minutos para `Edit Profile`) é substituído por um tempo de ajuste do RAG (corrigindo *diffs*), o que se provou mais eficiente.

7.3 Diálogo com os Trabalhos Relacionados

Este trabalho se insere na literatura de IA para testes de software, mas com duas contribuições específicas.

Em relação a **(GUILHERME; VINCENZI, 2023)**, que testaram a geração de testes de unidade com o `gpt-3.5-turbo` (GUILHERME; VINCENZI, 2023), nosso trabalho avança ao aplicar o RAG. O estudo deles limitou o contexto do LLM ao código-fonte da classe. Este TCC enriqueceu o RAG com a **documentação técnica da ferramenta** (Maestro). Isso é crucial, pois permitiu ao LLM usar comandos específicos (`assertVisible`, etc.) que ele não conheceria apenas pelo código-fonte, mitigando as falhas que os autores observaram.

Em relação a **Wang et al. (2025)**, que propuseram um RAG dinâmico sobre um grafo de código para detecção de bugs (WANG; GUO; TAN, 2025), este trabalho é complementar. Enquanto eles focam em um RAG dinâmico para *prevenir* bugs, nosso foco foi em um RAG estático (baseado em documentação e código) para *reduzir o débito técnico existente*.

A principal distinção de ambos os trabalhos é a nossa restrição metodológica: o uso de **modelos locais (Ollama)**. Esta decisão, motivada pela política de segurança da empresa, prova que é possível obter resultados práticos na redução de débito técnico sem depender de APIs de terceiros e sem comprometer a confidencialidade do código-fonte.

8 CONCLUSÃO E TRABALHOS FUTUROS

8.1 Disseminação dos resultados

Este trabalho investigou a aplicação de sistemas de Geração Aumentada por Recuperação (RAG) na automação de testes *end-to-end*, visando à mitigação do débito técnico de testes em um ambiente operacional de *startup*. A síntese dos resultados demonstra que a utilização de modelos de linguagem locais, orquestrados pela arquitetura RAG, não apenas é viável tecnicamente, mas também responde eficazmente aos requisitos de segurança e privacidade inerentes ao contexto corporativo.

As principais contribuições deste estudo dividem-se em três frentes principais:

1. **Contribuição Técnica:** O desenvolvimento de um protótipo funcional que integra ferramentas como Ollama e LangChain para a geração contextualizada de *scripts* Maestro, validando a eficácia da execução local de LLMs para tarefas de codificação.
2. **Contribuição Prática:** A elevação da cobertura de testes de 26% para 68% nos fluxos críticos do aplicativo, reduzindo a barreira de entrada para a equipe de QA e otimizando o esforço temporal do ciclo de desenvolvimento.
3. **Contribuição Científica:** A análise empírica de parâmetros de recuperação, como o *chunk overlap*, e a avaliação de desempenho de diferentes modelos de *embedding* em tarefas específicas de engenharia de software.

A relevância prática da pesquisa foi confirmada pela recepção positiva dos resultados. O projeto foi apresentado internamente à equipe técnica da empresa parceira e selecionado para apresentação em um evento técnico organizado pelo Google, evidenciando o potencial de disseminação e aplicação do conhecimento gerado.

8.2 Trabalhos Futuros

Apesar dos resultados satisfatórios, a pesquisa identificou oportunidades para evoluções futuras que podem ampliar o impacto da solução proposta:

- **Integração com Ambientes de Desenvolvimento (IDEs):** O desenvolvimento de extensões para editores como o VSCode permitiria a atualização em tempo real do banco de dados vetorial conforme o código-fonte é alterado, integrando a geração

de testes diretamente ao fluxo de trabalho do desenvolvedor.

- **Mecanismos de Autocorreção (*Self-healing*):** A implementação de um agente capaz de executar os testes gerados e, em caso de erro de sintaxe ou falha de execução, reenviar o erro e o contexto ao LLM para correção automática do *script*.
- **Generalização de Domínios:** A expansão do *crawler* para suportar documentações de outros *frameworks* de teste (como Playwright ou Appium) e diferentes linguagens de programação, validando a versatilidade da arquitetura RAG em diversos ecossistemas tecnológicos.
- **Otimização de Contexto Dinâmico:** A exploração de grafos de código para fornecer um contexto mais rico ao LLM, permitindo que a IA compreenda não apenas arquivos isolados, mas as interdependências entre componentes do sistema.

REFERÊNCIAS

ANTHROPIC. **Claude**. s.d. Available from Internet: <https://claude.ai/new>.

ATLASSIAN. **What is Code Coverage? | Atlassian**. Available from Internet: <https://www.atlassian.com/continuous-delivery/software-testing/code-coverage>.

BECK, K. **Test-driven development : by example**. 1st. ed. [S.l.]: Addison-Wesley, 2002. 240 p. ISBN 9780321146533.

BUCHANAN, I. **História do DevOps | Atlassian**. s.d. Available from Internet: <https://www.atlassian.com/br/devops/what-is-devops/history-of-devops>.

FOWLER, M. **Test Driven Development**. 2023. Available from Internet: <https://martinfowler.com/bliki/TestDrivenDevelopment.html>.

GOOGLE; ALPHABET. **Google Gemini**. 2023. Available from Internet: <https://gemini.google.com/app>.

GUILHERME, V.; VINCENZI, A. An initial investigation of chatgpt unit test generation capability. In: **ACM International Conference Proceeding Series**. [S.l.]: Association for Computing Machinery, 2023. p. 15–24. ISBN 9798400716294.

HASSANI, H. S. **LLM Embeddings Explained: A Visual and Intuitive Guide - a Hugging Face Space by hesamation**. 2025. Available from Internet: <https://huggingface.co/spaces/hesamation/primer-llm-embedding?section=what_are_embeddings%3F>.

HU, K. ChatGPT sets record for fastest-growing user base - analyst note. 2023. Available from Internet: <https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/>.

IBGE. **Internet foi acessada em 72,5 milhões de domicílios do país em 2023 | Agência de Notícias**. 2024. Section: Estatísticas Sociais. Available from Internet: <https://agenciadenoticias.ibge.gov.br/agencia-noticias/2012-agencia-de-noticias/noticias/41024-internet-foi-acessada-em-72-5-milhoes-de-domicilios-do-pais-em-2023>.

IEEE. **The Importance of Software Testing**. Available from Internet: <https://www.computer.org/resources/importance-of-software-testing/>.

INC, M. D. **Maestro, End-to-End UI Testing for Mobile and Web**. 2022. Available from Internet: <https://maestro.dev/>.

LECUN, Y.; BENGIO, Y.; HINTON, G. **Deep learning**. [S.l.]: Nature Publishing Group, 2015. 436-444 p.

LEWIS, P. et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. **Advances in Neural Information Processing Systems**, Neural information processing systems foundation, v. 2020-December, 5 2020. ISSN 10495258. Available from Internet: <https://arxiv.org/abs/2005.11401v4>.

LLAMAINDEX. **Embeddings | LlamaIndex Python Documentation**. Available from Internet: <https://developers.llamaindex.ai/python/framework/module_guides/models/embeddings/>.

MICROSOFT; GITHUB. **GitHub Copilot · Your AI pair programmer**. 2021. Available from Internet: <https://github.com/features/copilot>.

MUCCI, T. **What is Technical Debt? | IBM**. 2025. Available from Internet: <https://www.ibm.com/think/topics/technical-debt>.

NICOLAESCU, C. **The end of test-driven development - Pragmatic Optimism**. Accessed on 27/11/2025. Available from Internet: <https://cosn.io/posts/tdd>.

OLLAMA. **Embedding models · Ollama Blog**. 2024. Available from Internet: <https://ollama.com/blog/embedding-models>.

OPENAI. **ChatGPT**. 2022. Available from Internet: <https://chatgpt.com/>.

OROSZ, G. **Stack overflow is almost dead**. 2025. Available from Internet: <https://blog.pragmaticengineer.com/stack-overflow-is-almost-dead/>.

PRACTITEST. **Unveiling the 2024 State of Testing | PractiTest**. 2024. Available from Internet: <https://www.practitest.com/resource-center/blog/unveiling-the-2024-state-of-testing>.

RADIGAN, D. **What is Technical Debt? [+How to Manage It] | Atlassian**. s.d. Available from Internet: <https://www.atlassian.com/agile/software-development/technical-debt>.

SARAVIA, E. Prompt Engineering Guide. **https://github.com/dair-ai/Prompt-Engineering-Guide**, 12 2022.

SILVA, C. R. R. da. **Maestro RAG**. 2025. Available from Internet: <https://github.com/caiorrs/tcc>.

STRYKER, C. **What Are Large Language Models (LLMs)? | IBM**. s.d. Available from Internet: <https://www.ibm.com/think/topics/large-language-models>.

UNION, I. T. **Individuals using the Internet - ITU DataHub**. 2024. Available from Internet: <https://datahub.itu.int/data/?e=701&i=11624>.

VASWANI, A. et al. Attention is all you need. p. 1, 6 2017. Available from Internet: <https://arxiv.org/abs/1706.03762v7>.

VOCKE, H. **The Practical Test Pyramid**. 2018. Available from Internet: <https://martinfowler.com/articles/practical-test-pyramid.html>.

WANG, Y.; GUO, S.; TAN, C. W. From code generation to software testing: Ai copilot with context-based rag. **IEEE Software**, v. 42, p. 34–42, 7 2025. ISSN 0740-7459, 1937-4194. Available from Internet: <http://arxiv.org/abs/2504.01866>.

WASHIZAKI, H. **Guide to the Software Engineering Body of Knowledge - SWEBOK**. [S.l.], 2025.

9 APÊNDICES

9.1 Escrita manual dos passos dos testes

Onboarding

- launch app with storage and keychain cleared
- assert that "Simplified gigs"is visible
- assert that "Navigating events has never been easier."is visible
- assert that "Next"button is visible
- tap on "Next"button
- assert that "Embrace the craft"is visible
- assert that "Work at events with premium drinks from over 200 global brands."is visible
- assert that "Next"button is visible
- tap on "Next"button
- assert that "Toast to your earnings"is visible
- assert that "Every hour counts, and we count it for you. Check your hourly earnings for each worked event."is visible
- assert that "Let’s go"button is visible
- tap on "Let’s go"button
- assert that TEST_IDS.screens.login.loginButton is visible

Login

- launch app without clearing storage and keychain
- assert that TEST_IDS.screens.login.loginButton is visible
- tap on "Email"input
- fill email input with "neatdottest+brand_staff@gmail.com"
- tap on "Password"input
- fill password input with "admin123"
- tap on TEST_IDS.screens.login.loginButton
- assert that EVENTS_HEADER is visible

Forgot Password

- launch app without clearing storage and keychain
- assert that TEST_IDS.screens.login.loginButton is visible

- tap on "First access or forgot password"
- assert "Need help?"is visible
- tap on "Email"input
- fill email input with "neatdottest+brand_staff@gmail.com"
- tap on "Send Link"
- assert "Link sent to your email!"is visible
- assert "Open email"is visible

First Access Flow
- run login flow
- assert that "First and Last name"is visible
- tap on "First and Last name"input
- fill "First and Last name"input with "EndToEnd Test User"
- tap on "Date of birth"input
- fill "Date of birth"input with "12/31/1990"
- assert "Next"is visible
- tap on "Next"
- assert "Phone number"is visible
- tap on "Phone number"input
- fill "Phone number"input with "2345678910"
- tap on "Address"
- assert "Search address"is visible
- tap on "Search address"input
- fill "Search address"input with "UCLA Medical Center"
- assert "UCLA Medical Center, Medical Plaza, Los Angeles, CA, USA"is visible
- tap on "UCLA Medical Center, Medical Plaza, Los Angeles, CA, USA"
- tap on "Confirm"
- scroll to the bottom of the screen
- assert "90024"is visible
- assert "Next"is visible
- tap on "Next"
- assert "Select the languages you speak"is visible
- tap on "Select the languages you speak"
- assert "Search languages"is visible

- assert "English"is visible
- tap on "English"
- assert "Spanish"is visible
- tap on "Spanish"
- assert "Confirm"is visible
- tap on "Confirm"
- assert "Select the languages you speak"is visible
- assert "Select one option"is visible
- tap on "Select one option"
- assert "years"is visible
- tap on "0-2 years"
- assert "None"is not displayed
- assert "0-2 years"is visible
- assert "Next"is visible
- tap on "Next"
- assert "Contact information"is visible
- scroll to the bottom of the screen
- assert "Email"is visible
- assert "Phone number"is visible
- assert "Next"is visible
- tap on "Next"
- assert "To start using Neat., please agree to our"is visible
- scroll to the bottom of the screen
- assert "I agree to the terms and conditions and privacy policy"is visible
- tap on "I agree to the terms and conditions and privacy policy"
- assert "Create account"is visible
- tap on "Create account"
- assert "Your Neat. account is ready!"is visible
- assert "Access Neat."is visible
- tap on "Access Neat."
- asserts "Events"is visible

Edit Profile

- run login flow

- assert "Profile"is visible
- tap on "Profile"
- assert testid_edit_personal_info is visible
- tap on testid_edit_personal_info
- assert "EndToEnd Test User"is visible
- tap on "EndToEnd Test User"
- clear input field
- type "EndToEnd Test User edit"
- tap x y position 10, 10 (to dismiss keyboard)
- scroll down 100 pixels or 10% of the screen
- assert "UCLA"is visible
- tap on "UCLA"
- assert "Search address"is visible
- tap on "Search address"input
- fill "Search address"input with "UCLA Luskin"
- assert "UCLA Luskin School of Public Affairs"is visible
- tap on "UCLA Luskin School of Public Affairs"
- tap on "UCLA Luskin School of Public Affairs"
- tap on "Confirm"
- assert "90095"is visible
- scroll down until "Save changes"is visible
- tap on "Save changes"
- scroll to top
- assert "EndToEnd Test User edit"is visible
- scroll down until "90095"is visible
- scroll down until testid_edit_skills is visible
- tap on testid_edit_skills
- scroll down until "Select the languages you speak"is visible
- tap on "Select the languages you speak"
- assert "Search languages"is visible
- tap on "Search languages"
- fill "Search languages"with "en"
- assert "French"is visible
- tap on "French"

- assert "Confirm"is visible
- tap on "Confirm"
- assert "French"is visible
- scroll down until "Bartending experience"is visible
- tap on testid_bartendingxp_input
- assert "5-10 years"is visible
- tap on "5-10 years"
- scroll down until "Save changes"is visible
- tap on "Save changes"
- assert "French"is visible
- assert "5-10 years"is visible
- scroll up until "Settings"is visible
- tap on "Settings"
- assert "Signout"is visible
- tap on "Settings"
- scroll down until visible "FAQ and Contact Us"
- assert "Notify me automatically"is visible
- tap on "Notify me automatically"

Sign out
- run login flow
- assert "Profile"is visible
- tap on "Profile"
- assert "Settings"is visible
- tap on "Settings"
- assert "Signout"is visible
- tap on "Signout"
- assert that TEST_IDS.screens.login.loginButton is visible

View booked events list
- run login flow
- assert "Schedule"is visible
- tap on "Schedule"
- assert "My schedule"is visible

- assert "Booked (*)"is visible
- tap on "Booked (*)"
- assert "Invites (*)"is visible
- tap on "Invites (*)"
- assert "Completed"is visible
- tap on "Completed"
- assert "Today"is visible
- tap on "Today"
- assert "This week"is visible
- tap on "This week"
- assert "This month"is visible
- tap on "This month"
- scroll down until "[E2E]"is visible
- tap on testid_event_card_e2e
- assert "[E2E]"is visible
- assert "Tasks"is visible
- tap on "Tasks"
- assert "To do"is visible
- assert "Photos"is visible
- tap on "Photos"
- assert "Choose photos"is visible
- assert "Details"is visible
- tap on "Details"
- assert "miles"is visible

9.2 Prompt Onboarding

Write a Maestro test flow for Onboarding.

The steps are:

1. launch app with storage and keychain cleared
2. assert that "Simplified gigs"is visible
3. assert that "Navigating events has never been easier."is visible
4. assert that "Next"button is visible

5. tap on "Next"button
6. assert that "Embrace the craft"is visible
7. assert that "Work at events with premium drinks from over 200 global brands."is visible
8. assert that "Next"button is visible
9. tap on "Next"button
10. assert that "Toast to your earnings"is visible
11. assert that "Every hour counts, and we count it
for you. Check your hourly earnings for each worked event."is visible 12. assert that "Let’s go"button is visible
13. tap on "Let’s go"button
14. assert that TEST_IDS.screens.login.loginButton is visible

9.3 Prompt Login

Write a Maestro test flow for Login.

The steps are:
1. launch app without clearing storage and keychain
2. assert that TEST_IDS.screens.login.loginButton is visible
3. tap on "Email"input
4. fill email input with "neatdottest+brand_staff@gmail.com"
5. tap on "Password"input
6. fill password input with "admin123"
7. tap on TEST_IDS.screens.login.loginButton
8. assert that EVENTS_HEADER is visible (Note: a testid needs to be added for this element)

9.4 Prompt Forgot Password

Write a Maestro test flow for Forgot Password.

The steps are:
1. launch app without clearing storage and keychain
2. assert that TEST_IDS.screens.login.loginButton is visible
3. tap on "First access or forgot password"

4. assert "Need help?"is visible
5. tap on "Email"input
6. fill email input with "neatdottest+brand_staff@gmail.com"
7. tap on "Send Link"
8. assert "Link sent to your email!"is visible
9. assert "Open email"is visible

9.5 Prompt First Access Flow

Write a Maestro test flow for the First Access Flow.

Note: This flow assumes a previous login flow has just completed.

The steps are:

1. assert that "First and Last name"is visible (use a testid for this)
2. tap on "First and Last name"input
3. fill "First and Last name"input with "EndToEnd Test User"
4. tap on "Date of birth"input (use a testid for this)
5. fill "Date of birth"input with "12/31/1990"
6. assert "Next"is visible
7. tap on "Next"
8. assert "Phone number"is visible
9. tap on "Phone number"input
10. fill "Phone number"input with "2345678910"
11. tap on "Address"input
12. assert "Search address"is visible
13. tap on "Search address"input
14. fill "Search address"input with "UCLA Medical Center"
15. assert "UCLA Medical Center, Medical Plaza, Los Angeles, CA, USA"is visible
16. tap on "UCLA Medical Center, Medical Plaza, Los Angeles, CA, USA"
17. tap on "Confirm"
18. scroll to the bottom of the screen
19. assert "90024"(zip code) is visible
20. assert "Next"is visible
21. tap on "Next"
22. assert "Select the languages you speak"is visible

23. tap on "Select the languages you speak"
24. assert "Search languages"is visible
25. assert "English"is visible
26. tap on "English"
27. assert "Spanish"is visible
28. tap on "Spanish"
29. assert "Confirm"is visible
30. tap on "Confirm"
31. assert "Select the languages you speak"is visible
32. assert "Select one option"is visible
33. tap on "Select one option"
34. assert "years"is visible
35. tap on "0-2 years"
36. assert "None"is not displayed
37. assert "0-2 years"is visible
38. assert "Next"is visible
39. tap on "Next"
40. assert "Contact information"is visible
41. scroll to the bottom of the screen
42. assert "Email"toggle is visible (use testid for the toggle)
43. assert "Phone number"toggle is visible (use testid for the toggle)
44. assert "Next"is visible
45. tap on "Next"
46. assert "To start using Neat., please agree to our"is visible
47. scroll to the bottom of the screen
48. assert "I agree to the terms and conditions and privacy policy"is visible
49. tap on "I agree to the terms and conditions and privacy policy"
50. assert "Create account"is visible
51. tap on "Create account"
52. assert "Your Neat. account is ready!"is visible
53. assert "Access Neat."is visible
54. tap on "Access Neat."
55. assert "Events"is visible

9.6 Prompt Edit Profile

Write a Maestro test flow for Edit Profile.

Note: This flow assumes a previous login.

The steps are:

1. assert "Profile"is visible and tap on it
2. assert testid_edit_personal_info is visible and tap on it
3. assert "EndToEnd Test User"is visible
4. tap on "EndToEnd Test User"
5. clear the input field
6. type "EndToEnd Test User edit"
7. tap at position x:10, y:10 to dismiss keyboard (if needed)
8. scroll down 100 pixels
9. assert "UCLA"is visible
10. tap on "UCLA"
11. assert "Search address"is visible
12. tap on "Search address"input
13. fill "Search address"input with "UCLA Luskin"
14. assert "UCLA Luskin School of Public Affairs"is visible
15. tap on "UCLA Luskin School of Public Affairs"
16. tap on "Confirm"
17. assert "90095"(new zip code) is visible
18. scroll down until "Save changes"is visible
19. tap on "Save changes"
20. scroll to top
21. assert "EndToEnd Test User edit"is visible
22. scroll down until "90095"is visible
23. scroll down until testid_edit_skills is visible
24. tap on testid_edit_skills
25. scroll down until "Select the languages you speak"is visible
26. tap on "Select the languages you speak"
27. assert "Search languages"is visible
28. tap on "Search languages"
29. fill "Search languages"with "en"

30. assert "French"is visible
31. tap on "French"
32. assert "Confirm"is visible
33. tap on "Confirm"
34. assert "French"tag is visible
35. scroll down until "Bartending experience"is visible
36. tap on testid_bartendingxp_input
37. assert "5-10 years"is visible
38. tap on "5-10 years"
39. scroll down until "Save changes"is visible
40. tap on "Save changes"
41. assert "French"is visible
42. assert "5-10 years"is visible
43. scroll up until "Settings"is visible
44. tap on "Settings"
45. assert "Signout"is visible
46. tap on "Settings"to dismiss the overlay
47. scroll down until "FAQ and Contact Us"is visible
48. assert "Notify me automatically"toggle is visible (use testid with state)
49. tap on "Notify me automatically"

9.7 Prompt Sign out

Write a Maestro test flow for Sign Out.

Note: This flow assumes a previous login.

The steps are:

1. assert "Profile"is visible and tap on it
2. assert "Settings"is visible and tap on it
3. assert "Signout"is visible and tap on it
4. assert that TEST_IDS.screens.login.loginButton is visible

9.8 Prompt View booked event

Write a Maestro test flow for View Booked Events List.

Note: This flow assumes a previous login.

The steps are:

1. assert "Schedule"is visible and tap on it
2. assert "My schedule"is visible
3. assert "Booked (*)"is visible (use regex) and tap on it
4. assert "Invites (*)"is visible (use regex) and tap on it
5. assert "Completed"is visible and tap on it
6. assert "Today"is visible (use regex) and tap on it
7. assert "This week"is visible (use regex) and tap on it
8. assert "This month"is visible and tap on it
9. scroll down until an event with "[E2E]"in the title is visible
10. tap on testid_event_card_e2E (use regex for testid if needed)
11. assert "[E2E]"is visible
12. assert "Tasks"is visible and tap on it
13. assert "To do"is visible
14. assert "Photos"is visible and tap on it
15. assert "Choose photos"is visible
16. assert "Details"is visible and tap on it
17. assert "miles"is visible

9.9 Prompt sem RAG

```
import time
from maestro.test_flow import TestFlow
from maestro.page_object import PageObject
from maestro.assertion import Assertion

class OnboardingTestFlow(TestFlow):
    """
    A Maestro test flow to perform the onboarding process.
    """
```

```
def __init__(self, app):
    super().__init__(app)
    self.app = app
    self.page_object = PageObject(self.app)  # Assuming PageObject
    self.assertion = Assertion(self.app)

def run(self):
    """
    Executes the onboarding test flow.
    """
    self.app.launch()
    time.sleep(3)  # Allow app to fully launch

    # 1. Assert that "Simplified gigs" is visible
    self.assertion.is_element_visible(self.page_object.simplified_g
    print("Step 1: 'Simplified gigs' is visible")

    # 2. Assert that "Navigating events has \n never been easier."
    self.assertion.is_element_visible(self.page_object.navigating_e
    print("Step 2: 'Navigating events has \n never been easier.' is
    ...
    ...
```

Example usage (assuming you have a running app instance named 'app'):...

9.10 Listagem de fluxos

Tabela 9.1: Matriz de Rastreabilidade - Módulo de Eventos e Filtros

Fluxo	**Descrição**	**Criticidade**	**Antes**	**Depois**
Events (View List)	Usuário acessa a aba "Eventos"e visualiza a lista principal de eventos disponíveis.	Crítica	✓	✓
Events (View Event Details)	Usuário clica em um evento na lista e visualiza sua tela de detalhes.	Crítica	×	✓
Events (Set Availability)	Usuário marca sua disponibilidade (disponível, indisponível, etc.) para um evento.	Crítica	×	✓
Events (Load More)	Usuário rola até o fim da lista e aciona a paginação para carregar mais eventos.	Alta	×	✓
Events (Refresh)	Usuário clica no botão de refresh para atualizar os eventos.	Alta	×	✓
Events (Details - Toggle Accordions)	Na tela de detalhes, usuário expande e retrai os menus sanfona (accordions).	Média	×	✓
Events (Details - Open Documents)	Na tela de detalhes, usuário clica para abrir documentos/links externos.	Média	×	✓
Events (Details - Remove From Market)	Usuário opta por se remover do programa de marketing de um evento específico.	Alta	×	×
Events (View All Filters)	Usuário abre a tela/modal que exibe todas as opções de filtro disponíveis.	Crítica	✓	✓
Events (Filter by Date)	Usuário aplica um filtro para ver eventos de um dia específico.	Crítica	×	×
Events (Filter by Date Range)	Usuário aplica um filtro para ver eventos em uma faixa de datas.	Crítica	×	×
Events (Search by Name)	Usuário utiliza a barra de busca para encontrar um evento pelo nome.	Baixa	×	×
Filter (Apply Brand)	Usuário aplica com sucesso um filtro de Marca (Brand).	Crítica	✓	×
Filter (Apply Channel)	Usuário aplica com sucesso um filtro de Canal (Channel).	Crítica	×	✓
Filter (Apply Marketing Area)	Usuário aplica com sucesso um filtro de Área de Marketing.	Crítica	×	×
Filter (Apply Availability)	Usuário aplica com sucesso um filtro de Disponibilidade.	Crítica	×	✓
Filter (Apply Event Status)	Usuário aplica com sucesso um filtro de Status do Evento.	Crítica	×	✓
Filter (Apply Radius)	Usuário aplica com sucesso um filtro de Distância/Raio.	Crítica	×	×
Filter (Search Filters)	Usuário busca por um termo dentro da lista de opções de filtro.	Alta	×	×

Tabela 9.2: Matriz de Rastreabilidade - Módulo de Agenda (Schedule)

Fluxo	Descrição	Criticidade	Antes	Depois
Schedule (Booked - Today)	Usuário visualiza a lista de eventos agendados (Booked) para "Hoje".	Crítica	✓	✓
Schedule (Booked - This Week)	Usuário visualiza a lista de eventos agendados (Booked) para "Esta Semana".	Crítica	✓	✓
Schedule (Booked - This Month)	Usuário visualiza a lista de eventos agendados (Booked) para "Este Mês".	Crítica	✓	✓
Schedule (Booked - Previous Month)	Usuário visualiza a lista de eventos agendados (Booked) do mês anterior.	Crítica	×	✓
Schedule (Booked - Next Month)	Usuário visualiza a lista de eventos agendados (Booked) do próximo mês.	Crítica	×	✓
Schedule (Invited - Today)	Usuário visualiza a lista de convites (Invited) para "Hoje".	Crítica	✓	✓
Schedule (Invited - This Week)	Usuário visualiza a lista de convites (Invited) para "Esta Semana".	Crítica	×	✓
Schedule (Invited - This Month)	Usuário visualiza a lista de convites (Invited) para "Este Mês".	Crítica	×	✓
Schedule (Invited - Previous Month)	Usuário visualiza a lista de convites (Invited) do mês anterior.	Crítica	×	✓
Schedule (Invited - Next Month)	Usuário visualiza a lista de convites (Invited) do próximo mês.	Crítica	×	✓
Schedule (Completed - Today)	Usuário visualiza a lista de eventos concluídos (Completed) de "Hoje".	Crítica	✓	✓
Schedule (Completed - This Week)	Usuário visualiza a lista de eventos concluídos (Completed) desta "Semana".	Crítica	✓	✓
Schedule (Completed - This Month)	Usuário visualiza a lista de eventos concluídos (Completed) deste "Mês".	Crítica	✓	✓
Schedule (Completed - Previous Month)	Usuário visualiza a lista de eventos concluídos (Completed) do mês anterior.	Crítica	×	✓
Schedule (Completed - Next Month)	Usuário visualiza a lista de eventos concluídos (Completed) do próximo mês.	Crítica	×	✓
Schedule (View Booked Event Details)	Usuário clica em um evento na sua agenda ("Schedule") e vê os detalhes.	Crítica	×	✓
Schedule (Event Interaction - Check-in/Out)	Usuário realiza ações em um evento agendado (check-in, check-out, etc.).	Crítica	×	✓
s Schedule (Event Interaction - Respond Task)	Usuário responde a uma tarefa (task) associada a um evento agendado.	Crítica	×	✓
Schedule (Event Interaction - Upload Photo)	Usuário envia uma foto solicitada em uma tarefa de um evento agendado.	Crítica	×	✓

Tabela 9.3: Matriz de Rastreabilidade - Módulo de Perfil e Configurações

Fluxo	**Descrição**	**Criticidade**	**Antes**	**Depois**
Profile (View)	Usuário navega até a tela de "Perfil"e visualiza suas informações.	Crítica	✓	✓
Profile (Edit Personal Info)	Usuário edita e salva suas informações pessoais (nome, endereço, etc.).	Crítica	×	✓
Profile (Edit Skills)	Usuário edita e salva suas habilidades e experiências (skills).	Crítica	×	✓
Profile (Add Picture)	Usuário adiciona ou atualiza sua foto de perfil com sucesso.	Alta	×	×
s Profile (Remove Picture)	Usuário remove sua foto de perfil existente.	Média	×	×
Profile (Sign Out)	Usuário realiza o logout da sua conta no aplicativo.	Alta	×	✓
Profile (Toggle Share Email)	Usuário altera a configuração de compartilhamento de e-mail.	Média	×	×
Profile (Toggle Share Phone)	Usuário altera a configuração de compartilhamento de telefone.	Média	×	×
Profile (Toggle Notify Updates)	Usuário altera a configuração de notificação de atualizações.	Média	×	✓
s Profile (Toggle Dark Mode)	Usuário altera a configuração de tema (modo claro/escuro).	Média	×	×
Profile (Check for Updates)	Usuário aciona manualmente a verificação de novas atualizações do app.	Média	×	×

s

Tabela 9.4: Matriz de Rastreabilidade - Módulos Gerais e de Suporte

Fluxo	**Descrição**	**Criticidade**	**Antes**	**Depois**
Contact Us (Text + Attachments Success)	Usuário envia uma mensagem de contato com sucesso, incluindo anexos.	Crítica	×	×
Contact Us (Text Only Success)	Usuário envia uma mensagem de contato com sucesso, apenas com texto.	Crítica	×	×
Contact Us (File Error)	Usuário tenta anexar um arquivo muito grande ou de formato inválido e recebe erro.	Alta	×	×
Update Modal (Get New Version)	Usuário clica para ir ao site para baixar a nova versão.	Alta	✓	✓
Update Modal (Dismiss)	Usuário fecha o modal de atualização forçada/sugerida.	Média	✓	✓
s Update Modal (Do Not Show Again)	Usuário opta por não ver o modal de atualização sugerida novamente.	Média	✓	✓