Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Sistemas de Informação

Engenharia de Cenários de Teste End-to-End para APIs RESTful

Uma Ferramenta em Python com Arquitetura Modular para Análise de Cobertura de Parâmetros e Simulação de Fluxos de Comunicação

Fabrício de Sousa Guidine

JUIZ DE FORA
JANEIRO, 2026

Engenharia de Cenários de Teste End-to-End para APIs RESTful

Uma Ferramenta em Python com Arquitetura Modular para Análise de Cobertura de Parâmetros e Simulação de Fluxos de Comunicação

Fabrício de Sousa Guidine

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Sistemas de Informação

Orientador: Victor Ströele de Andrade Menezes

JUIZ DE FORA

JANEIRO, 2026

ENGENHARIA DE CENÁRIOS DE TESTE END-TO-END PARA APIS RESTFUL

Uma Ferramenta em Python com Arquitetura Modular para Análise de Cobertura de Parâmetros e Simulação de Fluxos de Comunicação

Fabrício de Sousa Guidine

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM SISTEMAS DE INFORMAÇÃO.

Aprovada por:

Victor Ströele de Andrade Menezes
Doutor em Engenharia de Sistemas e Computação (UFRJ)

André Luiz de Oliveira
Doutor em Ciência da Computação (USP)

José Maria Nazar David
Doutor em Engenharia de Sistemas e Computação (UFRJ)

JUIZ DE FORA

20 DE JANEIRO, 2026

Dedico às minhas mecenas,

Denise ❀ Margarida

Resumo

O crescimento exponencial de arquiteturas baseadas em microsserviços aumentou significativamente a complexidade dos testes de APIs REST, particularmente devido à validação de múltiplos *endpoints* e à explosão combinatória de cenários de teste. Além disso, garantir a rastreabilidade entre requisitos de negócio e operações de API é fundamental para assegurar que os testes validem efetivamente as funcionalidades prioritárias, evitando lacunas de cobertura e testes redundantes. Neste contexto, este trabalho apresenta uma ferramenta automatizada para geração de cenários de teste de APIs REST a partir de especificações OpenAPI/Swagger e documentos de requisitos de negócio. A abordagem proposta realiza análise estrutural automatizada de especificações de API nos formatos Swagger 2.0 e OpenAPI 3.0 e 3.1, combinada com processamento de linguagem natural de requisitos de negócio por meio da integração com múltiplos provedores de LLM. Como resultado, a ferramenta sintetiza cenários de teste comportamentais no formato Gherkin, compatíveis com *frameworks* BDD, computa métricas de cobertura cruzada entre requisitos de negócio e *endpoints*, e deriva métricas de complexidade por análise algorítmica, empregando *chunking* adaptativo para processar eficientemente APIs de larga escala. O sistema adota arquitetura modular que facilita a incorporação de novos provedores LLM e formatos de saída. Resultados de um estudo de caso utilizando a API Weather.gov demonstram a viabilidade da abordagem proposta. A ferramenta gerou 127 cenários de teste em aproximadamente 45 segundos, cobrindo 59 *endpoints* e 342 parâmetros, alcançando 100% de cobertura de requisitos com consumo total de 5.172 *tokens*. Esses resultados indicam que a solução é aplicável a ambientes reais de desenvolvimento de *software*, oferecendo custos operacionais viáveis e integração com *pipelines* de integração contínua. Adicionalmente, a ferramenta é disponibilizada como código aberto, contribuindo para a comunidade de garantia de qualidade.

Palavras-chave: Testes de API, OpenAPI, Swagger, LLM, Gherkin, Automação, BDD

Abstract

The exponential growth of microservice-based architectures has significantly increased the complexity of REST API testing, particularly due to the need to validate multiple endpoints and the combinatorial explosion of test scenarios. Furthermore, ensuring traceability between business requirements and API operations is essential to guarantee that tests effectively validate priority functionalities, avoiding coverage gaps and redundant tests. In this context, this work presents an automated tool for generating REST API test scenarios from OpenAPI/Swagger specifications and business requirements documents. The proposed approach performs automated structural analysis of API specifications in Swagger 2.0 and OpenAPI 3.0 and 3.1 formats, combined with natural language processing of business requirements through integration with multiple large language model (LLM) providers. As a result, the tool synthesizes behavioral test scenarios in Gherkin format, compatible with established BDD frameworks, computes cross-coverage metrics between business requirements and API endpoints, and derives complexity metrics through algorithmic analysis, employing an adaptive chunking strategy to process large-scale APIs efficiently. The system adopts a modular architecture that facilitates the incorporation of new LLM providers and output formats. Results from a case study using the Weather.gov API demonstrate the feasibility of the proposed approach. The tool generated 127 test scenarios in approximately 45 seconds, covering 59 endpoints and 342 parameters, achieving 100% requirement coverage with a total token consumption of 5,172. These findings indicate that the proposed solution applies to real-world software development environments, offering viable operational costs and seamless integration with continuous integration pipelines. Additionally, the tool is made available as open-source software, contributing to the broader quality assurance community.

Keywords: API testing, OpenAPI, Swagger, LLM, Gherkin, Automation, BDD

Agradecimentos

À Denise, que me fez uma de suas prioridades, que me educou para ser uma pessoa que valida os sentimentos em primeiro lugar e me ensinou a ser um bom menino e, agora, um homem. Não consigo imaginar meios de te retribuir tudo o que me foi provido. Ao meu pai, Hélio, que a seu modo pôde se fazer presente e sempre útil e prestativo quanto ao meu crescimento; é meu companheiro, e eu, seu panheirinho. À minha irmã, Larissa, que, além de eu amar muito, trouxe ao mundo outra parte de si para eu amar ainda mais: Mateus, que veio com meus traços.

À minha rocha, vó Margarida (in memoriam), que viveu sua vida trabalhando em prol da nossa família e daqueles que faziam parte da sua vida, salvo exceções, de acordo com suas ideias irrevogáveis. Foi a primeira a me fazer acreditar que as coisas dariam certo na minha caminhada acadêmica. À minha tia, Rosângela (in memoriam), que estará eternamente em meus pensamentos sobre como cresci sob seus cuidados em Cataguarino. Sempre vou te amar. Ao restante da minha família: avô Cosme (in memoriam), tios e tias, afilhados e afilhadas, primos e primas, cunhado e sua família, conforta-me saber que entendem que a distância e os sacrifícios que tive seriam insuportáveis sem o seu apoio.

Ao meu ex-futuro marido, Erick: poder estar em sua companhia é tão natural e me fez experimentar a felicidade em seu mais calmo e puro estado. Eu amei compartilhar a minha vida com você.

À Alana, ao Fabrício, Milena e à Thaís: eu amo vocês. Aos meus amigos da faculdade em geral, cujos nomes não caberiam aqui. Em especial, à Débora, que me forneceu uma lanterna da lua[1] para seguir no umbral. Aos docentes do meu departamento, que me proveram a metodologia e os meios para meu desenvolvimento acadêmico. Em especial, ao Luciano, ao André e ao Victor. Essas pessoas viram em mim algo que nem eu acreditava: sou capaz.

[1] Lanterna da Lua: ⟨https://bg3.wiki/wiki/Moonlantern⟩

Ontem fui feliz, excessivamente feliz, como não se pode sê-lo mais! Até que enfim, uma vez na vida, você, sempre tão inacessível, satisfez os meus desejos! Eram cerca de oito horas, já quase noite, quando acordei da soneca que costumo dormir todos os dias, depois do trabalho. Acendi a luz e tinha já os papéis em ordem, faltando-me apenas aguçar a pena, quando, de súbito, levantei, casualmente, os olhos e deparou-se-me um espetáculo extraordinário, que me fez pular o coração. Decerto adivinhou já do que se trata, compreendeu o motivo do meu alvoroço!

Fiódor Dostoiévski (Gente Pobre)

Conteúdo

Lista de Figuras **iii**

Lista de Tabelas **iv**

Lista de Abreviações **1**

1 Introdução **2**

2 Fundamentação Teórica **7**
2.1 Arquitetura REST e APIs Web 7
2.2 Especificação OpenAPI 8
2.3 Critérios de Cobertura para APIs REST 11
2.4 Rastreabilidade de Requisitos 12
2.5 Desenvolvimento Orientado a Comportamento 14
2.6 Modelos de Linguagem de Grande Porte em Engenharia de Software 16
2.7 Métricas de Qualidade em Automação de Testes 18
2.8 Síntese do Capítulo 19

3 Materiais e Métodos **20**
3.1 Visão Geral do Sistema 20
3.2 Arquitetura do Sistema 23
3.3 Fluxo de Funcionamento 25
3.4 Tecnologias Utilizadas 26
3.5 Estrutura do Projeto 28
3.6 Componentes Principais 29

4 Estudo de Caso **30**
4.1 Descrição do Cenário 30
4.2 Configuração da Execução 31
4.3 Métricas da API Processada 31
4.4 Geração e Estrutura do BRD 32
4.4.1 Análise do BRD Gerado 33
4.5 Geração de Cenários de Teste 34
4.5.1 Análise Qualitativa dos Cenários 36
4.6 Exportação e Artefatos Gerados 39
4.7 Análise de Desempenho e Custos 40
4.8 Discussão dos Resultados 41

5 Considerações Finais **42**
5.1 Síntese do Trabalho 42
5.2 Retomada das Questões de Pesquisa 43
5.3 Contribuições 44
5.4 Limitações 45
5.5 Trabalhos Futuros 46

Bibliografia **48**

A Documento de Requisitos de Negócio Gerado **51**

B Amostra de Cenários de Teste Gerados **55**

C Estrutura Completa do Projeto **57**
C.1 Organização de Diretórios de Nível Raiz 57

D Estrutura do Diretório Source **58**

E Cenários para Endpoint de Alertas Meteorológicos **63**

Lista de Figuras

2.1 Exemplo da UI OpenAPI visualizada no Swagger Editor 9
2.2 Exemplo de especificação OpenAPI visualizada no Swagger Editor 10
2.3 Diagrama conceitual de rastreabilidade entre BRD e cenários de teste . . . 13

3.1 Página principal do repositório GitHub do projeto 21
3.2 Continuação da página principal do repositório GitHub do projeto 22
3.3 Arquitetura do sistema organizada em camadas 24
3.4 Fluxograma das sete etapas do processo de geração de cenários 26
3.5 Visão simplificada da estrutura de diretórios principal 28

4.1 Saída do terminal durante geração de cenários via LLM 37

Lista de Tabelas

3.1 Principais tecnologias e dependências do projeto 27
3.2 Resumo dos módulos principais e suas responsabilidades 29

4.1 Métricas estruturais da API weather.gov 32
4.2 Análise de cobertura cruzada BRD vs *Endpoints* 34
4.3 Métricas de geração de cenários via LLM 35
4.4 Análise de custos operacionais da execução 40

B.1 Amostra de cenários de teste gerados para a API weather.gov 55

C.1 Estrutura de diretórios e arquivos de nível raiz 57

D.1 Especificação dos componentes do sistema 58

Lista de Abreviações

API	*Application Programming Interface*
BDD	*Behavior-Driven Development*
BRD	*Business Requirement Document*
CI/CD	*Continuous Integration/Continuous Delivery*
CLI	*Command Line Interface*
CPU	*Central Processing Unit*
CSV	*Comma-Separated Values*
GPT	*Generative Pre-trained Transformer*
HTTP	*Hypertext Transfer Protocol*
IEEE	*Institute of Electrical and Electronics Engineers*
JSON	*JavaScript Object Notation*
LLM	*Large Language Model*
MIT	*Massachusetts Institute of Technology*
NLP	*Natural Language Processing*
OpenAPI	*OpenAPI Specification*
PDF	*Portable Document Format*
RAM	*Random Access Memory*
REST	*Representational State Transfer*
RTM	*Requirements Traceability Matrix*
SDK	*Software Development Kit*
TDD	*Test-Driven Development*
TXT	*Text File*
UFJF	Universidade Federal de Juiz de Fora
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*
YAML	*YAML Ain't Markup Language*

1 Introdução

A arquitetura de *software* contemporânea tem experimentado uma transformação significativa nas últimas décadas, caracterizada pela migração de sistemas monolíticos para arquiteturas distribuídas baseadas em serviços. Nesse contexto evolutivo, as APIs REST emergiram como o principal mecanismo de comunicação entre componentes de *software*, especialmente em ecossistemas orientados a microsserviços (FIELDING, 2000). Dessa forma, essas interfaces expõem *endpoints* HTTP que recebem parâmetros através de mecanismos de transmissão, incluindo cabeçalhos de requisição, corpo de mensagem e *queries*, retornando respostas predominantemente estruturadas em formato JSON (BANIAS; ALEXANDRESCU, 2022). A adoção massiva de arquiteturas baseadas em microsserviços por grandes empresas de tecnologia tem impulsionado a necessidade de abordagens mais sofisticadas para garantia de qualidade dessas interfaces críticas (NEWMAN, 2015).

A confiabilidade desses contratos de API influencia diretamente tanto a integridade das integrações entre sistemas quanto a experiência do usuário final. Neste contexto, conforme evidenciado por Golmohammadi, Zhang e Arcuri (2023), o teste de APIs RESTful apresenta desafios particulares devido à dependência de comunicações de rede e às interações frequentes com serviços externos, como sistemas de gerenciamento de banco de dados, o que amplia significativamente a complexidade do processo de verificação. Ademais, a proliferação de serviços *web* baseados em REST tem motivado um crescimento substancial na pesquisa acadêmica sobre técnicas de teste automatizado para esse tipo de interface (KIM et al., 2022; RICHARDSON; AMUNDSEN; RUBY, 2013).

Com a adoção crescente de padrões como OpenAPI, anteriormente conhecido como Swagger, tornou-se viável descrever contratos de serviço de maneira padronizada e processável por sistemas automatizados (MARTIN-LOPEZ; SEGURA; RUIZ-CORTéS, 2019). A especificação OpenAPI permite definir formalmente *endpoints* disponíveis, métodos HTTP suportados, parâmetros de entrada diferenciados por localização, esquemas de dados via JSON Schema, respostas esperadas para cada código de status HTTP, e mecanismos de autenticação e autorização, estabelecendo assim uma base formal para automação

de processos de teste (OpenAPI Initiative, 2021). Conforme observado por Corradini et al. (2022), a mera existência de especificações OpenAPI não resolve automaticamente os desafios inerentes ao teste de APIs, uma vez que as ferramentas existentes frequentemente negligenciam informações valiosas presentes em descrições textuais dessas especificações (ED-DOUIBI; IZQUIERDO; CABOT, 2018).

Dois problemas fundamentais persistem no domínio dos testes automatizados de APIs REST. O primeiro problema refere-se à explosão combinatória de valores de parâmetros e de cenários de teste possíveis, fenômeno que torna impraticável a geração manual de casos de teste com cobertura abrangente (ARCURI, 2019). Técnicas de teste combinatorial têm sido propostas para mitigar esse problema, permitindo cobertura sistemática de interações entre parâmetros (KUHN; KACKER; LEI, 2013). O segundo problema, igualmente crítico mas frequentemente negligenciado, diz respeito à desconexão sistemática entre os requisitos de negócio e os cenários técnicos de teste (ZAMENI; WANG; MAHMOUD, 2023). Dessa forma, enquanto as especificações OpenAPI descrevem com precisão a estrutura técnica de uma API, estas frequentemente não capturam as regras de negócio subjacentes, os fluxos de trabalho esperados pelos *stakeholders* e os casos de uso prioritários do ponto de vista funcional.

Essa lacuna entre a documentação técnica e os requisitos funcionais dificulta substancialmente a criação de suítes de teste que sejam simultaneamente abrangentes em termos técnicos e alinhadas aos objetivos de negócio. A rastreabilidade entre requisitos e testes, considerada essencial pela norma IEEE 830-1998 para especificação de requisitos de software, permanece um desafio significativo em projetos de desenvolvimento de *software* (GOTEL; FINKELSTEIN, 1994). Neste contexto, conforme evidenciado por estudos empíricos, a ausência de rastreabilidade adequada pode resultar em funcionalidades não testadas, testes redundantes que desperdiçam recursos, e dificuldade substancial na análise de impacto de mudanças nos requisitos (MäDER; EGYED, 2012). A manutenção de matrizes de rastreabilidade tem sido identificada como prática fundamental em organizações com processos maduros de qualidade (SPANOUDAKIS; ZISMAN, 2005; CLELAND-HUANG; GOTEL; ZISMAN, 2012).

Nos últimos anos, a integração de modelos de linguagem de grande porte em

ferramentas de engenharia de *software* tem demonstrado potencial notável para automatizar tarefas que tradicionalmente exigiam *expertise* humana especializada, incluindo a geração de código-fonte, documentação técnica, e casos de teste (WANG et al., 2024). A arquitetura *Transformer*, que fundamenta os LLMs modernos, revolucionou o processamento de linguagem natural e abriu novas possibilidades para automação inteligente (VASWANI et al., 2017). Ademais, modelos avançados como GPT-4 demonstram capacidade de interpretar especificações técnicas complexas, compreender contexto de negócio expresso em linguagem natural, e gerar artefatos estruturados tanto em linguagem natural quanto em formatos semiestruturados como o Gherkin (SCHäFER et al., 2023). Essa capacidade emergente abre novas possibilidades para a automação inteligente de processos de teste, embora apresente desafios relacionados à qualidade dos artefatos gerados, validação de adequação aos requisitos, e custos operacionais associados (WHITE et al., 2023).

A relevância deste trabalho fundamenta-se em múltiplos aspectos identificados tanto na literatura acadêmica quanto na prática de engenharia de *software*. Conforme demonstrado pela pesquisa conduzida por Golmohammadi, Zhang e Arcuri (2023) com análise sistemática de 92 artigos científicos, o campo de teste de APIs REST permanece ativo e em evolução, com desafios significativos ainda não plenamente resolvidos por ferramentas existentes no estado da arte. Diante disso, a explosão combinatória de valores de parâmetros e a necessidade de cobertura abrangente tornam a automação não apenas desejável do ponto de vista de eficiência, mas necessária para garantir qualidade em sistemas modernos de *software* distribuído (FRASER; ARCURI, 2011).

A integração de requisitos de negócio ao processo de teste representa uma necessidade crítica frequentemente negligenciada por abordagens puramente técnicas. A literatura sobre rastreabilidade de requisitos demonstra consistentemente que a manutenção de ligações claras e bidirecionais entre requisitos, casos de teste e resultados de execução é fundamental para garantia de qualidade, especialmente em sistemas críticos onde falhas podem ter consequências severas (GOTEL; FINKELSTEIN, 1994). Por conseguinte, a ferramenta proposta neste trabalho endereça diretamente essa lacuna ao integrar Documentos de Requisitos de Negócio de forma orgânica ao fluxo de geração de testes, estabelecendo rastreabilidade desde a concepção até a execução.

A aplicação de modelos de linguagem de grande porte na geração de testes representa uma fronteira de pesquisa particularmente promissora. Estudos recentes demonstram que LLMs como GPT-4 podem gerar casos de teste de alta qualidade quando adequadamente direcionados por *prompts* bem construídos e contextualizados (SCHäFER et al., 2023; WANG et al., 2024). Neste sentido, a ferramenta proposta tem o potencial de contribuir para essa linha de pesquisa ao demonstrar aplicação prática e sistemática de LLMs em contexto específico de teste de APIs REST, explorando estratégias de *chunking* para processamento de APIs de grande porte e integração com múltiplos provedores de LLM.

A adoção da linguagem Gherkin como formato de saída garante interoperabilidade com o ecossistema estabelecido de ferramentas de *Behavior-Driven Development*, incluindo Cucumber, Behave e SpecFlow (Cucumber, 2023). A metodologia BDD tem demonstrado benefícios significativos na comunicação entre equipes técnicas e de negócio (SOLIS; WANG, 2011). Dessa forma, essa escolha facilita substancialmente a integração com processos existentes de garantia de qualidade e permite que cenários gerados sirvam simultaneamente como documentação executável e especificação de comportamento esperado do sistema, característica fundamental da abordagem BDD. A integração com *pipelines* de entrega contínua potencializa ainda mais o valor dessa automação (HUMBLE; FARLEY, 2010).

Diante desse contexto, este trabalho busca responder à seguinte questão geral de pesquisa: *Como automatizar a geração de cenários de teste para APIs REST a partir de especificações OpenAPI?*

A partir dessa questão central, derivam-se cinco questões específicas que orientam o desenvolvimento da pesquisa. A primeira questão específica (**QE1**) investiga como processar e validar especificações OpenAPI em múltiplos formatos de forma automatizada, extraindo informações estruturadas sobre *endpoints*, parâmetros e esquemas de dados. A segunda questão (**QE2**) examina como integrar os Documentos de Requisitos de Negócio ao processo de geração de testes, estabelecendo a rastreabilidade entre requisitos funcionais e cenários de validação técnica. A terceira questão (**QE3**) avalia a eficácia de modelos de linguagem de grande porte na geração de cenários de teste no formato Gherkin, considerando critérios de coerência, completude e adequação aos requisitos especificados.

A quarta questão (**QE4**) analisa como implementar a análise de cobertura cruzada entre o BRD e os *endpoints* da API, identificando lacunas e priorizando *endpoints* críticos para testes. Por fim, a quinta questão (**QE5**) investiga quais métricas são adequadas para avaliar a qualidade, a complexidade e a cobertura dos cenários de teste gerados automaticamente.

O objetivo geral deste trabalho consiste em desenvolver a ferramenta **API Parameter Coverage & Test Scenario Generator**, que automatiza a geração de cenários de teste para APIs REST combinando análise de especificações OpenAPI/Swagger, integração com Documentos de Requisitos de Negócio e geração de cenários Gherkin via LLMs. O código-fonte está disponível no repositório GitHub (GUIDINE, 2025).

Os objetivos específicos são: (1) desenvolver módulo de processamento multi-formato para especificações Swagger 2.0 e OpenAPI 3.0/3.1; (2) implementar mecanismos de integração com BRD, incluindo carregamento, geração via LLM e extração de documentos em PDF, Word e TXT; (3) analisar cobertura cruzada entre *endpoints* e requisitos do BRD; (4) integrar múltiplos provedores de LLM com estratégia de *chunking* adaptativo; (5) desenvolver sistema de *analytics* com métricas de qualidade e cobertura; e (6) implementar exportação estruturada em formato CSV.

O restante deste documento está organizado em cinco capítulos. O Capítulo 2 apresenta a fundamentação teórica, revisando conceitos essenciais sobre teste de APIs REST, especificações OpenAPI, rastreabilidade de requisitos, desenvolvimento orientado a comportamento, modelos de linguagem de grande porte e métricas de qualidade em automação de testes. O Capítulo 3 descreve detalhadamente a arquitetura do sistema proposto, tecnologias utilizadas, organização modular do código-fonte e metodologia de desenvolvimento adotada. No Capítulo 4 é apresentado um estudo de caso detalhado demonstrando a aplicação da ferramenta na API weather.gov, com análise quantitativa e qualitativa dos resultados obtidos e discussão sobre as métricas coletadas. Por fim, o Capítulo 5 discute as principais contribuições acadêmicas e práticas, limitações identificadas durante o desenvolvimento e execução, e direções promissoras para trabalhos futuros. Os apêndices reúnem os materiais complementares, incluindo trechos relevantes de código-fonte, exemplos de especificações, *prompts* para modelos de linguagem e dados suplementares.

2 Fundamentação Teórica

Este capítulo apresenta os conceitos teóricos e trabalhos relacionados que fundamentam o desenvolvimento da ferramenta proposta. A revisão da literatura aborda tópicos essenciais sobre teste de APIs REST, especificações OpenAPI, rastreabilidade de requisitos, desenvolvimento orientado a comportamento, modelos de linguagem de grande porte e métricas de qualidade em automação de testes, estabelecendo o embasamento científico necessário para compreensão das decisões de projeto e implementação adotadas ao longo do trabalho.

2.1 Arquitetura REST e APIs Web

O estilo arquitetural REST foi introduzido por Roy Fielding em sua tese de doutorado como um conjunto de princípios para projeto de sistemas distribuídos hipermídia, fundamentando-se em conceitos como cliente-servidor, *stateless*, *cacheable*, interface uniforme e sistema em camadas (FIELDING, 2000). Neste contexto, APIs que seguem rigorosamente os princípios REST, denominadas APIs RESTful, utilizam o protocolo HTTP como mecanismo primário de comunicação, empregando métodos padronizados como GET para recuperação de recursos, POST para criação, PUT para atualização completa, PATCH para atualização parcial, e DELETE para remoção, operando sobre recursos identificados por URIs (RICHARDSON; AMUNDSEN; RUBY, 2013). A escalabilidade dessa arquitetura contribuiu para sua ampla adoção em sistemas *web* atuais (NEWMAN, 2015).

Dessa forma, as APIs REST estabelecem contratos entre produtor e consumidor de serviços, definindo a estrutura de requisições esperadas, formato de respostas produzidas e códigos de status HTTP que sinalizam o resultado de cada operação. Nesse contexto, os testes de API têm como objetivo verificar se essas interfaces se comportam conforme o especificado, avaliando aspectos como a correção das respostas, o tratamento adequado de entradas inválidas e a consistência das condições de erro (BANIAS; ALEXANDRESCU, 2022). A validação sistemática desses contratos é fundamental para garantir a integridade em arquiteturas de microsserviços (SEGURA et al., 2018).

Revisões sistemáticas da literatura evidenciam que, embora a automação de testes de API tenha evoluído significativamente ao longo dos últimos anos, ainda há notável falta de consenso sobre métricas de cobertura específicas para APIs, situação que contrasta com a relativa maturidade das métricas de cobertura de código tradicional estabelecidas para aplicações monolíticas (GOLMOHAMMADI; ZHANG; ARCURI, 2023). Por outro lado, conforme apontado por Kim et al. (2022) em análise empírica de ferramentas existentes, as soluções atualmente disponíveis apresentam limitações significativas tanto em termos de cobertura de código alcançada durante execução de testes quanto em capacidade de detectar falhas sutis em APIs reais, indicando espaço substancial para contribuições nessa área de pesquisa. Técnicas de teste metamórfico têm sido propostas como alternativa para superar limitações do oráculo de teste em APIs (SEGURA et al., 2018).

2.2 Especificação OpenAPI

A especificação OpenAPI, anteriormente conhecida como Swagger antes da transferência para a OpenAPI Initiative, tornou-se o padrão *de facto* para documentação formal de APIs REST em ecossistemas de desenvolvimento modernos (OpenAPI Initiative, 2021). A especificação permite descrever de forma completa e processável por sistemas automatizados todos os aspectos relevantes de uma API REST, incluindo *endpoints* disponíveis com suas respectivas URIs, métodos HTTP suportados para cada *endpoint*, parâmetros de entrada diferenciados por localização como *path*, *query*, *header* e *body*, esquemas de dados estruturados via JSON Schema, respostas esperadas para cada código de status HTTP possível, e mecanismos de autenticação e autorização suportados (MARTIN-LOPEZ; SEGURA; RUIZ-CORTéS, 2019). Na Figura 2.1 é ilustrado um exemplo típico de especificação OpenAPI visualizada no Swagger Editor, demonstrando tanto o código YAML estruturado à esquerda quanto a documentação interativa gerada automaticamente à direita a partir da especificação formal.

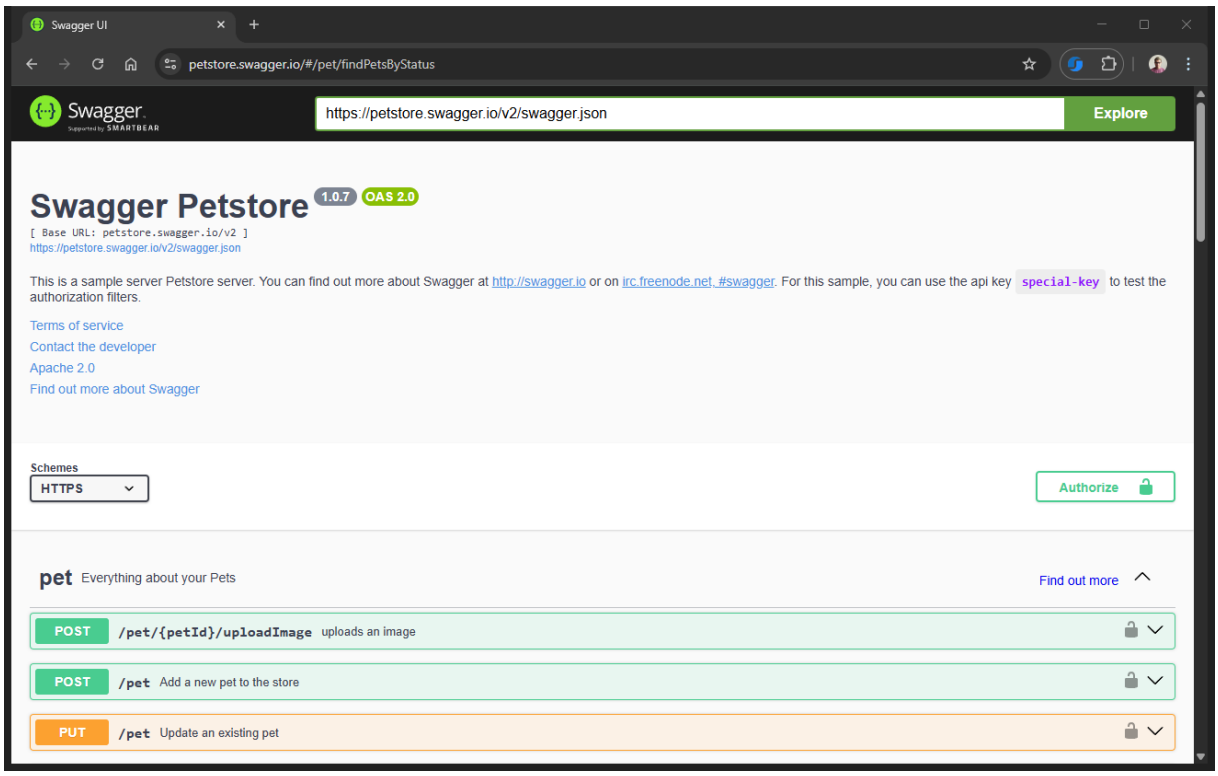

Figura 2.1: Exemplo da UI OpenAPI visualizada no Swagger Editor

Conforme observado na Figura 2.2, uma especificação OpenAPI típica é estruturada hierarquicamente, iniciando com metadados da API que incluem título, versão semântica e descrição textual, seguidos pela definição de servidores base disponíveis, caminhos de *endpoints* organizados por recurso, operações disponíveis para cada caminho, parâmetros detalhados de cada operação, *schemas* de *request* e *response bodies* estruturados, e definições de componentes reutilizáveis que promovem consistência e manutenibilidade (ED-DOUIBI; IZQUIERDO; CABOT, 2018). Neste contexto, essa estrutura hierárquica e bem definida facilita não apenas a documentação manual, mas especialmente o processamento automatizado por ferramentas de análise e geração de testes.

Dessa forma, a formalização proporcionada pelo padrão OpenAPI cria oportunidades significativas para a automação de processos de teste. Nesse contexto, ferramentas especializadas podem interpretar especificações OpenAPI e derivar testes automaticamente por meio de diferentes estratégias, gerando valores de parâmetros que respeitam os *constraints* definidos, construindo requisições HTTP bem formadas e validando respostas com base em esquemas formalmente especificados (CORRADINI et al., 2022).

Conforme catalogado em uma revisão sistemática recente, diversas ferramentas de teste automatizado foram desenvolvidas especificamente para explorar essas capacidades, incluindo RESTler, que utiliza técnicas de *fuzzing* guiado por gramática (ATLIDAKIS; GODEFROID; POLISHCHUK, 2019), EvoMaster, que aplica algoritmos evolutivos para maximização de cobertura (ARCURI, 2018), Schemathesis, que realiza testes baseados em propriedades a partir de especificações, RESTest, que gera casos de teste a partir da análise de dependências entre operações (MARTIN-LOPEZ; SEGURA; RUIZ-CORTéS, 2020), e RestTestGen, que emprega técnicas combinadas de análise estática e dinâmica (VIGLIANISI; DALLAGO; CECCATO, 2020).

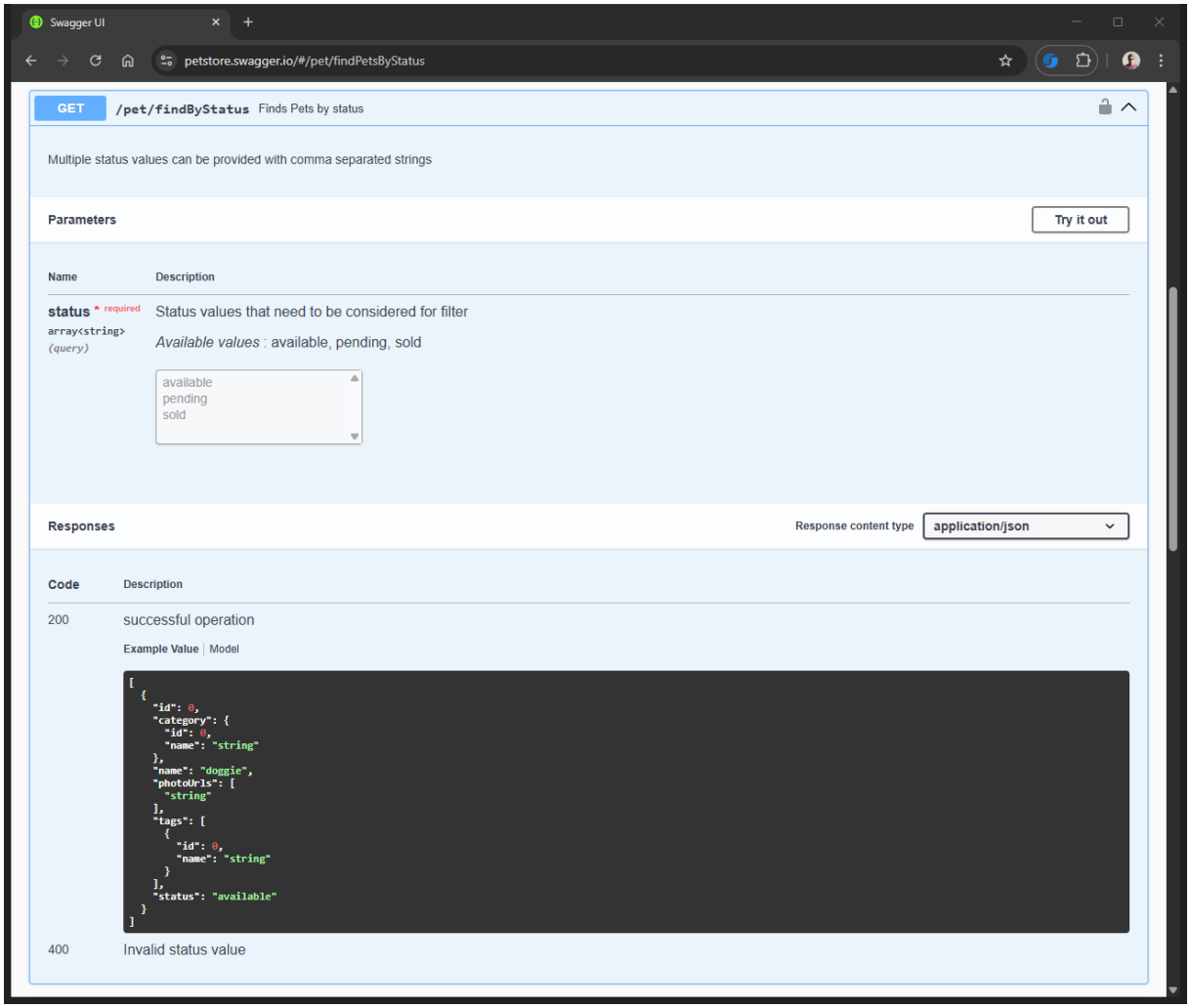

Figura 2.2: Exemplo de especificação OpenAPI visualizada no Swagger Editor

Por outro lado, conforme observação crítica de Corradini et al. (2022), a maior parte dessas ferramentas foca exclusivamente em informações estruturadas presentes nas

especificações OpenAPI, frequentemente negligenciando descrições textuais em linguagem natural que acompanham *endpoints*, parâmetros e *schemas*. Diante disso, trabalhos recentes, como o apresentado por Kim et al. (2023), demonstram que a extração e processamento de informações contidas em descrições textuais através de técnicas de processamento de linguagem natural pode melhorar significativamente a qualidade dos testes gerados, motivando abordagens híbridas que integrem análise estrutural com processamento de linguagem natural ao fluxo de geração de testes.

2.3 Critérios de Cobertura para APIs REST

Martin-Lopez, Segura e Ruiz-Cortés (2019) propõem um conjunto abrangente de critérios de cobertura específicos para APIs REST que transcendem a simples invocação de cada *endpoint* disponível. Neste contexto, os autores sugerem métricas organizadas em múltiplos níveis de granularidade, incluindo cobertura de parâmetros que garante que todos os parâmetros definidos sejam exercitados ao menos uma vez, cobertura de valores que assegura que diferentes valores representativos de cada parâmetro sejam testados incluindo casos limite, cobertura de operações que verifica diferentes métodos HTTP aplicados sobre o mesmo recurso, e cobertura de dependências entre operações que valida sequências de chamadas interdependentes respeitando ordem temporal e lógica. Esses critérios são análogos aos da técnica de teste funcional tradicional, como o Particionamento em Classes de Equivalência e a Análise do Valor Limite para cada parâmetro (AMMANN; OFFUTT, 2016).

Ademais, além do trabalho de Martin-Lopez, Arcuri (2019) propõem critérios baseados na cobertura de código do servidor, argumentando que métricas de caixa-branca que consideram a implementação interna podem complementar de forma substantiva abordagens de caixa-preta baseadas apenas em especificações externas. Essa abordagem relaciona-se com a técnica de teste estrutural, que inclui critérios baseados em fluxo de controle e fluxo de dados (AMMANN; OFFUTT, 2016). Dessa forma, a ferramenta EvoMaster, desenvolvida pelos autores e disponibilizada como *software* de código aberto, utiliza algoritmos genéticos multiobjetivos para maximizar simultaneamente a cobertura de código da implementação do servidor e a cobertura da especificação da API, por meio

da geração evolutiva de casos de teste que exploram o espaço de busca de forma inteligente (FRASER; ARCURI, 2011).

A literatura destaca a importância de uma cobertura abrangente de códigos de resposta HTTP, incluindo respostas de sucesso (2xx), erros do cliente (4xx) e erros do servidor (5xx), bem como a cobertura de esquemas de dados (GOLMOHAMMADI; ZHANG; ARCURI, 2023). Essa abordagem garante a validação sistemática de objetos JSON complexos por meio de combinações de valores válidos e inválidos. Técnicas de teste combinatorial permitem reduzir o espaço de teste mantendo cobertura adequada de interações entre parâmetros (KUHN; KACKER; LEI, 2013). Testes metamórficos complementam essa abordagem ao permitir validação sem oráculos explícitos (SEGURA et al., 2018). Esses critérios influenciam diretamente as decisões arquiteturais e a implementação dos algoritmos da ferramenta proposta neste trabalho.

2.4 Rastreabilidade de Requisitos

A rastreabilidade de requisitos é definida formalmente pela IEEE como o grau em que um relacionamento bidirecional pode ser estabelecido entre dois ou mais produtos do processo de desenvolvimento de *software*, especialmente produtos com relacionamento predecessor-sucessor ou primário-subordinado entre si (GOTEL; FINKELSTEIN, 1994). Neste contexto, no âmbito específico de teste de *software*, a rastreabilidade permite conectar de forma estruturada e verificável requisitos de negócio originais a casos de teste derivados e resultados de execução obtidos, garantindo que validações técnicas realizadas estejam consistentemente alinhadas com objetivos funcionais estabelecidos pelos *stakeholders* (SPANOUDAKIS; ZISMAN, 2005).

Dessa forma, enquanto especificações OpenAPI descrevem com precisão técnica a estrutura formal de uma API incluindo tipos de dados, formatos de mensagem e protocolos de comunicação, Documentos de Requisitos de Negócio capturam as necessidades funcionais de mais alto nível, regras de negócio que governam o comportamento esperado do sistema, e casos de uso prioritários do ponto de vista do cliente ou *stakeholder* que financia o desenvolvimento (ABDELFATTAH et al., 2024). Por conseguinte, a integração sistemática entre essas duas fontes complementares de informação permite filtrar de forma

fundamentada quais *endpoints* são funcionalmente prioritários para teste com base em requisitos explícitos, evitando desperdício de esforço e recursos em *endpoints* tecnicamente presentes mas funcionalmente não críticos ou fora de escopo atual (CLELAND-HUANG; GOTEL; ZISMAN, 2012). A Figura 2.3 ilustra conceitualmente essa relação hierárquica entre os três níveis de rastreabilidade: requisitos de negócio no topo, *endpoints* da API no meio, e cenários de teste na base, com setas bidirecionais indicando as ligações de rastreamento entre cada camada.

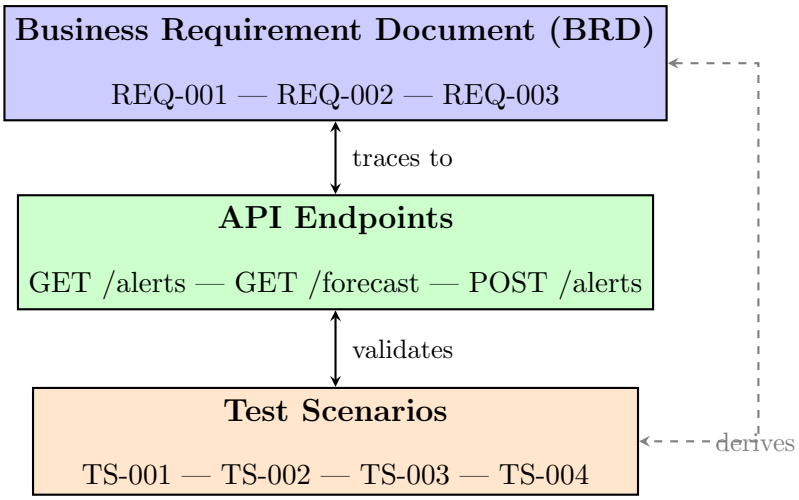

Figura 2.3: Diagrama conceitual de rastreabilidade entre BRD e cenários de teste

Conforme demonstrado na Figura 2.3, as conexões bidirecionais entre BRD, *endpoints* e cenários de teste estabelecem uma matriz de rastreabilidade que permite tanto navegação descendente (de requisitos para testes) quanto ascendente (de testes para requisitos). Ademais, a Matriz de Rastreabilidade de Requisitos, conhecida pela sigla RTM derivada do inglês *Requirements Traceability Matrix*, constitui o artefato tradicionalmente utilizado para documentar de forma tabular e estruturada os relacionamentos entre requisitos, casos de teste derivados e outros artefatos relevantes do projeto (MäDER; EGYED, 2012). Conforme observado por pesquisas empíricas conduzidas ao longo da última década, a manutenção de rastreabilidade adequada e atualizada pode acelerar substancialmente atividades de desenvolvimento como análise de impacto de mudanças, reduzir a incidência de defeitos através de melhor cobertura de requisitos, e facilitar comunicação entre equipes técnicas e de negócio (SPANOUDAKIS; ZISMAN, 2005). Por outro lado, o processo de criação e manutenção manual de matrizes de rastreabilidade é reconhecidamente traba-

lhoso, propenso a erros de inconsistência, e frequentemente negligenciado sob pressão de prazos apertados, motivando fortemente abordagens automatizadas (CLELAND-HUANG; GOTEL; ZISMAN, 2012).

2.5 Desenvolvimento Orientado a Comportamento

O Desenvolvimento Orientado a Comportamento, conhecido pela sigla BDD derivada do inglês *Behavior-Driven Development*, é uma metodologia que estende e refina os princípios do Desenvolvimento Orientado a Testes tradicional, focando especificamente na especificação do comportamento esperado do sistema através de exemplos concretos escritos em linguagem de domínio quase natural (NORTH, 2006). Neste sentido, BDD enfatiza a colaboração estruturada entre desenvolvedores que implementam funcionalidades, testadores que verificam conformidade, e *stakeholders* de negócio que definem requisitos, utilizando uma linguagem ubíqua compartilhada por todos os membros da equipe que elimina ambiguidades e mal-entendidos (SOLIS; WANG, 2011).

Dessa forma, a linguagem Gherkin constitui a sintaxe mais amplamente adotada para expressar cenários BDD de forma estruturada e processável, seguindo a estrutura canônica *Given/When/Then* que em português corresponde a Dado/Quando/Então (Cucumber, 2023). Cada cenário Gherkin descreve um teste comportamental através de três elementos fundamentais: uma precondição que estabelece o estado inicial do sistema, uma ação que representa a operação sendo testada, e um resultado esperado que define o comportamento correto do sistema. Ademais, essa estrutura é interpretável tanto por humanos que podem revisar e validar a especificação quanto por ferramentas automatizadas de execução de teste. O Cucumber é o *framework* BDD mais popular, disponível para múltiplas linguagens de programação incluindo Java, Ruby, JavaScript e Python (através da implementação Behave). O Cucumber interpreta arquivos `.feature` escritos em Gherkin, vincula cada *step* a implementações de código chamadas *step definitions*, e executa os testes gerando relatórios de sucesso ou falha (ZAMENI; WANG; MAHMOUD, 2023). Outras ferramentas compatíveis incluem SpecFlow para a plataforma .NET. A linguagem também suporta construções avançadas como *Scenario Outline* para parametrização de cenários com múltiplos conjuntos de dados, *Background* para definição de precondições

compartilhadas entre cenários de uma mesma *feature*, e *tags* para categorização e filtragem seletiva durante execução.

Para ilustrar, o Código 2.1 demonstra a estrutura completa de um arquivo Gherkin aplicado ao contexto de teste de API REST, incluindo a declaração da *Feature* com descrição no formato de *user story* que contextualiza o objetivo funcional, *tags* organizacionais para categorização e filtragem de cenários durante execução, e dois cenários representativos: um validando o fluxo positivo de criação de recurso com dados válidos, e outro verificando o tratamento adequado de erros quando campos obrigatórios estão ausentes ou inválidos.

```
Feature: Pet Store API - Pet Management
  As an API consumer
  I want to manage pets in the store
  So that I can maintain the pet inventory

  @api @pet @positive @smoke
  Scenario: Successfully create a new pet
    Given the API endpoint "/pet" is available
    And I have a valid authentication token
    And the request body contains valid pet data:
      | name   | status    | category |
      | Buddy  | available | dog      |
    When I send a POST request to "/pet"
    Then the response status code should be 200
    And the response body should contain the pet ID
    And the pet name should be "Buddy"
    And the pet status should be "available"

  @api @pet @negative @validation
  Scenario: Fail to create pet with missing required fields
    Given the API endpoint "/pet" is available
    And the request body is missing the "name" field
    When I send a POST request to "/pet"
    Then the response status code should be 400
    And the response should contain an error message
    And the error message should indicate "name is required"
```

Listing 2.1: Estrutura completa de cenário Gherkin para teste de API REST

Por conseguinte, essa característica de "especificação executável" proporcionada pela abordagem BDD reduz sistematicamente a divergência entre o que está documentado em artefatos de projeto e o que está efetivamente implementado e testado, além de facilitar substancialmente a comunicação entre equipes técnicas e não técnicas através de uma linguagem comum, pesquisas já exploraram ativamente a geração automática de cenários BDD a partir de requisitos expressos em linguagem natural ou especificações de alto nível (NORTH, 2006). Diante disso, Zameni, Wang e Mahmoud (2023) investigam especificamente o uso de LLMs para gerar cenários Gherkin a partir de requisitos textuais não estruturados, demonstrando empiricamente que modelos como GPT-4 conseguem produzir cenários estruturalmente coerentes e semanticamente bem formados quando guiados por *prompts* adequadamente construídos e contextualizados. Essa linha de pesquisa emergente inspira diretamente a abordagem técnica adotada neste trabalho para geração automatizada de cenários.

2.6 Modelos de Linguagem de Grande Porte em Engenharia de Software

Modelos de Linguagem de Grande Porte, conhecidos pela sigla LLM derivada do inglês *Large Language Models*, representam um avanço significativo em capacidade de compreensão de linguagem natural e geração de texto coerente. Baseados na arquitetura *Transformer* introduzida originalmente para tarefas de tradução automática (VASWANI et al., 2017), esses modelos são treinados em vastas quantidades de dados textuais cobrindo múltiplos domínios do conhecimento humano, demonstrando habilidade emergente em tarefas diversas como tradução entre idiomas, sumarização de textos longos, geração de código-fonte, e resposta a perguntas complexas (BROWN et al., 2020).

Dessa forma, LLMs têm sido aplicados com sucesso crescente em múltiplas tarefas do ciclo de desenvolvimento, incluindo geração de código a partir de descrições em linguagem natural que reduz esforço de implementação, geração automatizada de testes unitários

que aumenta cobertura, documentação automática de código existente que melhora manutenibilidade, detecção de *bugs* através de análise de padrões suspeitos, e refatoração de código para melhorar qualidade estrutural (WANG et al., 2024). Trabalhos recentes demonstram que LLMs podem interpretar especificações técnicas complexas expressas em formatos estruturados e gerar casos de teste funcionalmente relevantes. Para ilustrar, Kim et al. (2024) propõem especificamente o RESTGPT, uma abordagem inovadora que utiliza LLMs para extrair regras implícitas de descrições em linguagem natural presentes em especificações OpenAPI e gerar valores de parâmetros contextualmente mais adequados do que técnicas baseadas apenas em análise sintática.

Ademais, Schäfer et al. (2023) conduziram estudo empírico rigoroso sobre geração de testes unitários com LLMs, demonstrando que modelos de estado da arte como GPT-4 podem gerar testes com alta taxa de compilação sem erros sintáticos e cobertura de código comparável a ferramentas tradicionais de geração automatizada de testes. Por outro lado, os autores observam criticamente que a qualidade dos testes gerados por LLMs depende fortemente da qualidade dos *prompts* utilizados como entrada, enfatizando a importância fundamental de engenharia de *prompts* adequada que forneça contexto suficiente, exemplos representativos, e instruções claras sobre formato de saída esperado (WHITE et al., 2023). Estudos complementares indicam que técnicas como *few-shot prompting*, onde exemplos de entrada e saída desejada são fornecidos ao modelo, e *chain-of-thought prompting*, que solicita raciocínio passo a passo explícito antes da resposta final, podem melhorar significativamente a qualidade dos artefatos gerados (WEI et al., 2022).

Diante disso, um desafio técnico significativo na aplicação prática de LLMs consiste no gerenciamento de contexto, dado que modelos possuem limites físicos de *tokens* que podem ser processados em uma única invocação, e especificações de APIs de grande porte frequentemente excedem esses limites (ZAMENI; WANG; MAHMOUD, 2023). Estratégias de *chunking* que segmentam a entrada em porções menores processáveis, e técnicas de sumarização que condensam informações mantendo elementos essenciais para processar grandes volumes de dados de forma eficiente, dividindo a entrada em múltiplas chamadas ao modelo e consolidando resultados de forma coerente ao final do processamento (LEWIS et al., 2020). A implementação de *pipelines* de processamento iterativo, onde cada *chunk*

é enriquecido com contexto resumido dos *chunks* anteriores, permite manter coerência semântica entre as partes processadas separadamente.

2.7 Métricas de Qualidade em Automação de Testes

A avaliação objetiva da qualidade de cenários de teste gerados automaticamente requer métricas mensuráveis e reproduzíveis. Dessa forma, métricas tradicionais de teste de *software* incluem cobertura de código medida através de critérios como linhas executadas, *branches* tomados, e condições exercitadas, cobertura de requisitos que verifica se todos requisitos especificados possuem testes correspondentes, taxa de detecção de defeitos que mede eficácia dos testes em encontrar falhas reais, e taxa de falsos positivos que quantifica testes que falham incorretamente (AMMANN; OFFUTT, 2016). Neste contexto, no domínio específico de APIs REST, métricas adicionais são particularmente relevantes, incluindo cobertura de *endpoints* que verifica se todas as operações expostas são testadas, cobertura de parâmetros que assegura exercício de todos parâmetros definidos, diversidade de valores de parâmetros que avalia amplitude do espaço de entrada testado, e cobertura de códigos de resposta HTTP que valida tratamento de diferentes situações de sucesso e erro (MARTIN-LOPEZ; SEGURA; RUIZ-CORTéS, 2019).

Ademais, para cenários gerados especificamente por LLMs, métricas de qualidade textual tornam-se igualmente relevantes, incluindo coerência que avalia se a estrutura lógica do cenário faz sentido e *steps* estão ordenados apropriadamente, completude que verifica presença de todos os passos necessários para executar o teste incluindo *setup* e *teardown*, correção que valida adequação semântica às especificações da API, e legibilidade que mede clareza da linguagem utilizada para facilitar compreensão humana (YUAN et al., 2024). Para ilustrar, Yuan et al. (2024) propõem especificamente um *framework* de avaliação multi-dimensional que combina métricas automáticas computáveis para análise em larga escala com avaliação humana qualitativa para aferir aspectos subjetivos da qualidade de casos de teste gerados por LLMs.

Por outro lado, além de métricas de qualidade dos artefatos produzidos, aspectos operacionais do processo de automação devem ser monitorados sistematicamente para viabilizar uso prático da ferramenta, incluindo tempo de execução total e por etapa que

impacta viabilidade em *pipelines* CI/CD, consumo de recursos computacionais como *tokens* de LLM que tem implicações de custo, complexidade de entrada medida através de características do *schema* processado, e custo financeiro direto de utilização de APIs comerciais de LLM (ABDELFATTAH et al., 2024). A integração com *pipelines* de entrega contínua requer atenção especial a esses aspectos operacionais (HUMBLE; FARLEY, 2010). Neste sentido, *dashboards* analíticos que agregam e visualizam métricas ao longo de múltiplas execuções permitem análise de tendências temporais e tomada de decisão informada sobre estratégias de teste e configurações de ferramenta.

A definição e adoção de um conjunto de métricas técnicas, textuais e operacionais permitem comparar objetivamente cenários de teste gerados automaticamente com aqueles produzidos manualmente, e avaliar a escalabilidade e sustentabilidade econômica do uso de LLMs como componentes centrais em estratégias de automação de testes de APIs.

2.8 Síntese do Capítulo

A literatura revisada neste capítulo sugere cinco diretrizes convergentes que fundamentam a concepção e o desenvolvimento deste trabalho. A primeira diretriz enfatiza a necessidade de cobertura explícita e mensurável de APIs REST baseada em especificações formais que permitam automação efetiva (MARTIN-LOPEZ; SEGURA; RUIZ-CORTéS, 2019). A segunda diretriz ressalta a importância de integrar requisitos de negócio com validações técnicas para garantir a relevância funcional dos testes gerados, estabelecendo rastreabilidade adequada e bidirecional (GOTEL; FINKELSTEIN, 1994). A terceira diretriz reconhece o valor de representações legíveis e executáveis, como Gherkin/BDD, para aproximar especificação e teste, facilitando a comunicação entre *stakeholders* técnicos e não técnicos (ZAMENI; WANG; MAHMOUD, 2023). Ademais, a quarta diretriz explora a aplicação de LLMs para a geração inteligente de cenários de teste a partir de especificações complexas, aproveitando capacidades emergentes desses modelos para interpretação de contexto e geração de artefatos estruturados (WANG et al., 2024).

Por fim, a quinta diretriz estabelece a necessidade de métricas abrangentes que capturem múltiplas dimensões de qualidade e assegurem rastreabilidade em processos de automação de testes (ABDELFATTAH et al., 2024; HUMBLE; FARLEY, 2010).

3 Materiais e Métodos

Neste capítulo é descrita a metodologia adotada para o desenvolvimento da ferramenta proposta. A Seção 3.1 apresenta a visão geral do sistema e seus objetivos. A Seção 3.2 descreve a arquitetura em camadas. A Seção **??** detalha o fluxo de funcionamento em sete etapas, conforme ilustrado na Figura 3.4. A Seção **??** lista as tecnologias utilizadas. A Seção **??** apresenta a estrutura do projeto. Por fim, a Seção **??** descreve os componentes principais. A exposição metodológica permite a compreensão completa das decisões de projeto e facilita a reprodutibilidade do trabalho por pesquisadores e desenvolvedores interessados em estender ou adaptar a solução apresentada.

3.1 Visão Geral do Sistema

O sistema desenvolvido, denominado **API Parameter Coverage & Test Scenario Generator**, tem como objetivo principal automatizar o ciclo completo de geração de cenários de teste para APIs REST a partir de especificações OpenAPI/Swagger (GUIDINE, 2025). Neste contexto, a solução integra de forma orgânica requisitos de negócio estruturados através de Documentos de Requisitos de Negócio e utiliza inteligência artificial via modelos de linguagem de grande porte para síntese de cenários comportamentais no formato Gherkin, garantindo simultaneamente cobertura técnica abrangente e alinhamento com prioridades funcionais estabelecidas pelos *stakeholders* do projeto. A abordagem proposta diferencia-se das soluções existentes ao combinar análise estática de especificações com geração dinâmica assistida por inteligência artificial.

Dessa forma, a ferramenta foi concebida para atender simultaneamente a múltiplos objetivos identificados na revisão da literatura apresentada no capítulo anterior. O primeiro objetivo consiste em reduzir substancialmente o esforço manual envolvido na criação de suítes de teste abrangentes, automatizando etapas que tradicionalmente consomem tempo significativo dos analistas de qualidade. O segundo objetivo visa garantir alinhamento consistente entre os requisitos de negócio estabelecidos pelos *stakeholders* e as validações

técnicas implementadas nos testes automatizados, eliminando divergências sistemáticas que frequentemente ocorrem em processos manuais. O terceiro objetivo busca maximizar a cobertura de *endpoints* e de parâmetros funcionalmente relevantes por meio de análise automatizada de especificações e requisitos, aplicando critérios objetivos de priorização. O quarto objetivo foca em produzir artefatos rastreáveis e bem documentados que sejam diretamente integráveis com *pipelines* de CI/CD modernos, conforme práticas estabelecidas de entrega contínua (HUMBLE; FARLEY, 2010). Por fim, o quinto objetivo consiste em fornecer métricas objetivas e acionáveis de qualidade, complexidade e cobertura que suportem a tomada de decisão informada sobre estratégias de teste.

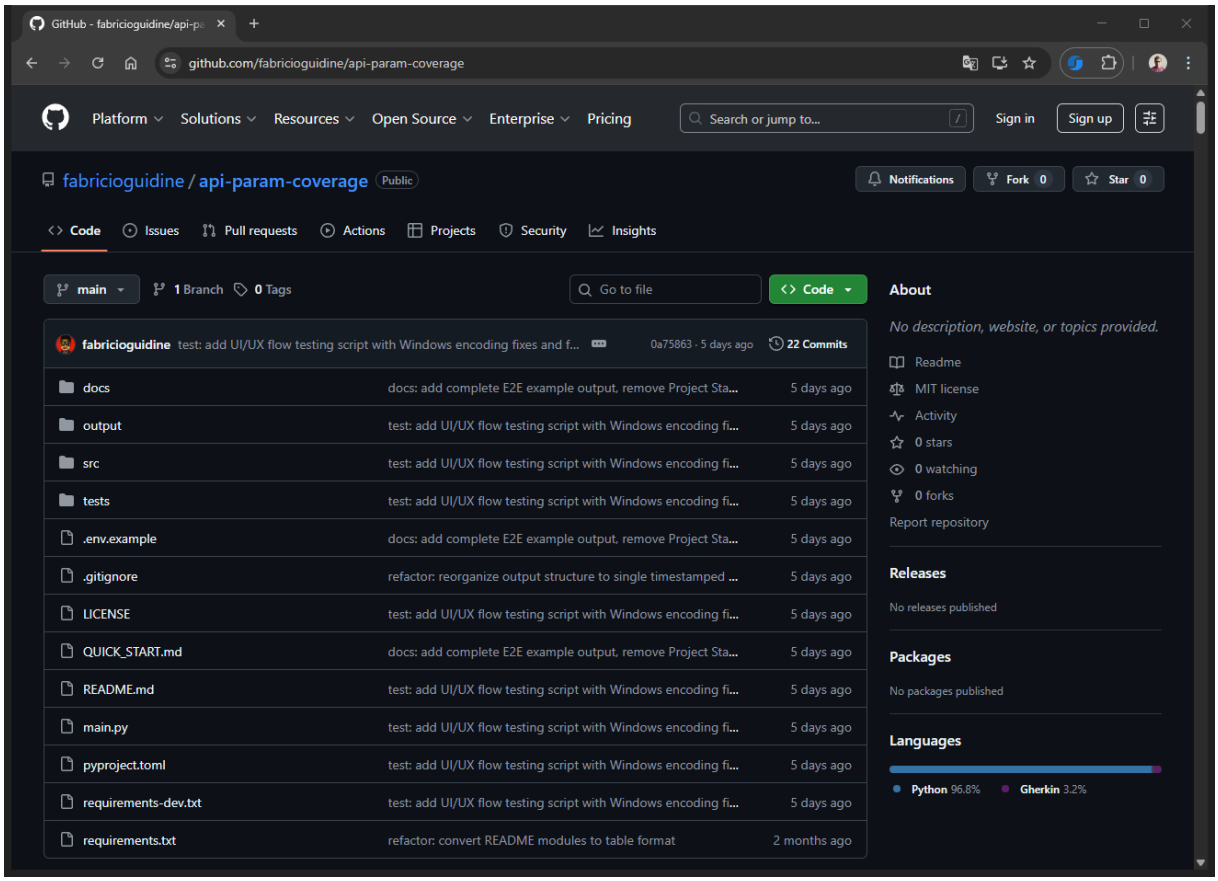

Figura 3.1: Página principal do repositório GitHub do projeto

Ademais, o repositório do projeto está hospedado publicamente no GitHub sob licença MIT, conforme ilustrado na Figura 3.1, que apresenta a página principal do repositório, contendo todo o código-fonte organizado modularmente, documentação técnica abrangente, incluindo guias de instalação e uso, exemplos práticos de execução com

APIs reais e suíte completa de testes automatizados cobrindo os módulos principais. A escolha da licença MIT promove a adoção ampla e facilita contribuições da comunidade de desenvolvedores. Essa decisão de licenciamento foi tomada considerando o equilíbrio entre a proteção dos direitos autorais e a maximização do impacto científico e prático da ferramenta.

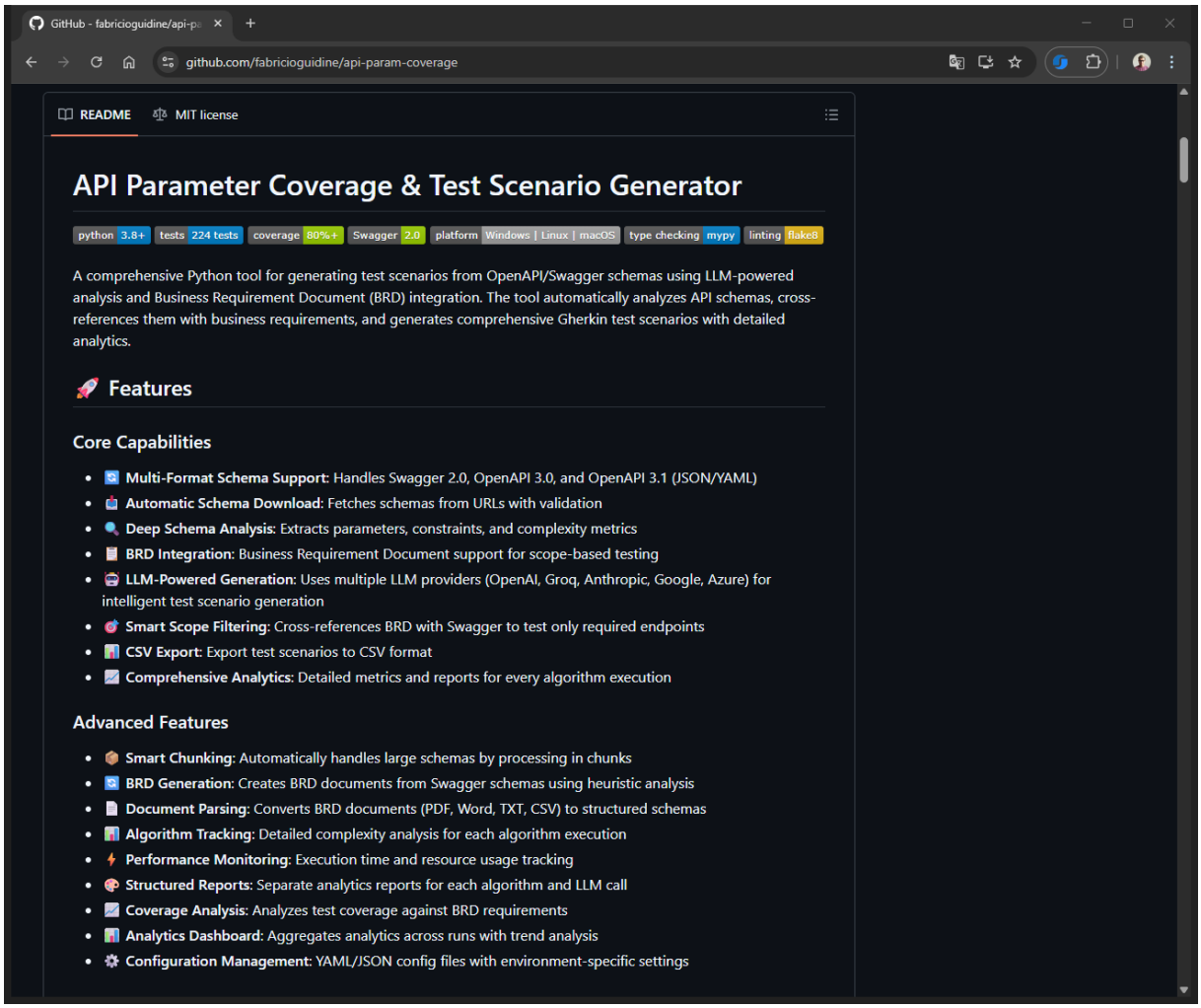

Figura 3.2: Continuação da página principal do repositório GitHub do projeto

Conforme observado na Figura 3.1, a estrutura do repositório segue as convenções estabelecidas para projetos Python de código aberto, facilitando a navegação e as contribuições da comunidade. Por conseguinte, a disponibilização pública do código-fonte promove transparência metodológica, permite validação independente dos resultados e facilita extensões futuras por parte da comunidade de pesquisa e desenvolvimento.

3.2 Arquitetura do Sistema

A arquitetura do sistema é composta por seis camadas interconectadas que colaboram para executar o *pipeline* completo de geração de cenários de teste. O *design* arquitetural segue o padrão arquitetural Dutos e Filtros (*Pipes and Filters*) (FOWLER, 2002), no qual cada estágio recebe a saída estruturada do estágio anterior como entrada e produz uma nova saída estruturada para o estágio subsequente. Essa abordagem favorece a separação de responsabilidades, a facilidade de manutenção e a evolução incremental da ferramenta. A escolha desse padrão arquitetural foi motivada pela natureza sequencial do processamento de especificações e pela necessidade de pontos de extensão bem definidos. A Figura 3.3 ilustra os principais componentes arquiteturais e o fluxo de dados entre eles.

A Camada de Entrada recebe especificações OpenAPI via URL remota ou arquivo local. A Camada de Processamento contém os módulos *Swagger*, responsável pela obtenção e validação da especificação através dos submódulos *Fetcher* e *Validator*, e *Engine*, que realiza o processamento através dos submódulos *Processor*, *Analyzer* e Gerador de CSV. A Camada de Integração LLM gerencia a comunicação com provedores de modelos de linguagem (OpenAI, Groq, Claude e Gemini) através do *LLM Prompter* e seus submódulos de segmentação, construção de *prompts* e gerenciamento de provedores. A Camada de Requisitos (BRD) integra o Gerador de BRD para geração automática de requisitos e o Gerenciador de BRD para carregamento, análise sintática e referência cruzada de documentos de requisitos.

A Camada de Análise consolida métricas de execução através do módulo de análise, com submódulos de métricas, rastreamento e painel de controle, além do Analisador de Cobertura para análise de cobertura de testes. Por fim, a Camada de Saída gerencia a exportação dos artefatos através do Gerenciador de Saída, produzindo CSV de cenários, relatórios analíticos e relatórios de validação. Os artefatos gerados seguem formatos padronizados que facilitam a integração com outras ferramentas do ecossistema de testes.

As conexões bidirecionais entre a Camada de Processamento e as camadas de LLM e BRD permitem o fluxo iterativo de dados, essencial para a geração inteligente de cenários comportamentais alinhados aos requisitos de negócio.

A Figura 3.3 ilustra os principais componentes arquiteturais e o fluxo de dados

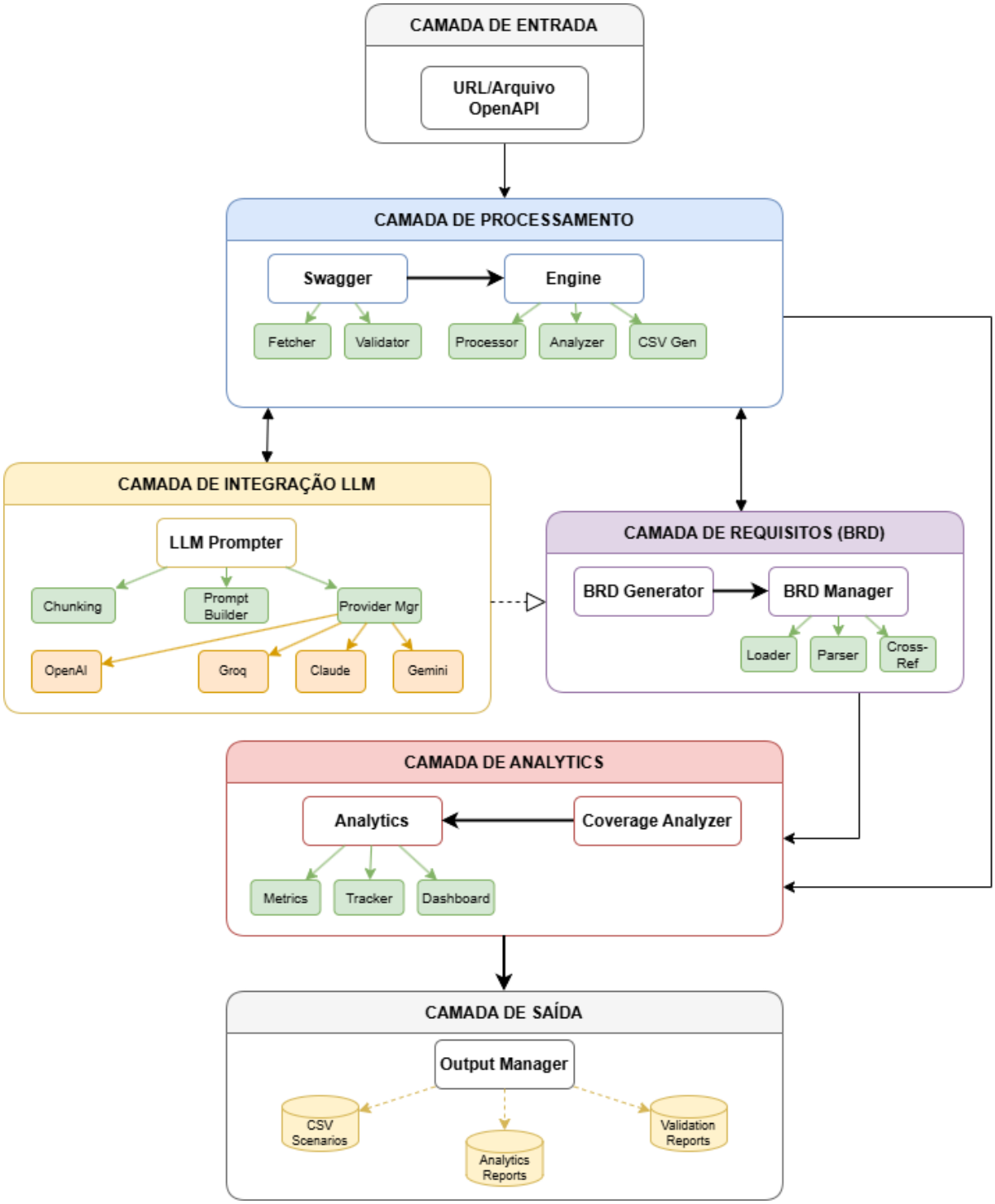

Figura 3.3: Arquitetura do sistema organizada em camadas

entre eles por meio de um diagrama de blocos, destacando três camadas conceituais principais. A camada de entrada engloba o módulo Swagger, composto pelos subcomponentes *Fetcher*, responsável pelo *download* de *schemas*, e *Validator*, responsável pela verificação de conformidade estrutural conforme a especificação OpenAPI 3.0. A camada de processamento contém o *Engine* principal, com os subcomponentes *Processor* para extração de estruturas internas, *Analyzer* para cálculo de métricas e LLM para integração com modelos de linguagem, além do módulo BRD paralelo contendo *Loader* para carregamento de requisitos e referência cruzada para análise de cobertura. Por fim, a camada de saída inclui o *CSV Generator* para exportação estruturada e o módulo *Analytics* para consolidação das métricas coletadas ao longo de todo o *pipeline*.

3.3 Fluxo de Funcionamento

O sistema opera por meio de sete etapas sequenciais bem definidas, cada uma responsável por realizar uma transformação específica sobre os dados de entrada até a produção dos artefatos finais estruturados. Essa estruturação em etapas discretas permite não apenas a execução ordenada do processamento, mas também a possibilidade de retomada parcial em caso de falhas, evitando reprocessamento desnecessário de etapas já concluídas. O mecanismo de *checkpointing* implementado permite recuperação eficiente em cenários de interrupção.

O processamento é orquestrado pelo módulo principal implementado em `main.py`, que coordena a execução ordenada de cada etapa do *pipeline*, gerencia a passagem de dados entre módulos por meio de estruturas padronizadas, implementa tratamento de erros e recuperação quando possível e mantém *logging* detalhado de todas as operações para rastreabilidade. O orquestrador também é responsável por validar pré-condições antes de iniciar cada etapa, garantindo que os dados de entrada atendam aos requisitos mínimos esperados pelo módulo subsequente. Essa validação prévia reduz significativamente a ocorrência de erros em estágios avançados do processamento.

A Figura 3.4 apresenta o fluxograma completo do processo de geração de cenários, incluindo o ponto de decisão relacionado à disponibilidade de BRD, no qual o usuário pode optar por carregar um documento previamente estruturado, gerar automaticamente

via LLM ou realizar o *parsing* de documentos não estruturados. Essa flexibilidade permite que a ferramenta seja utilizada tanto em cenários onde existe documentação formal de requisitos quanto em contextos mais ágeis, nos quais os requisitos podem ser inferidos diretamente a partir da especificação da API.

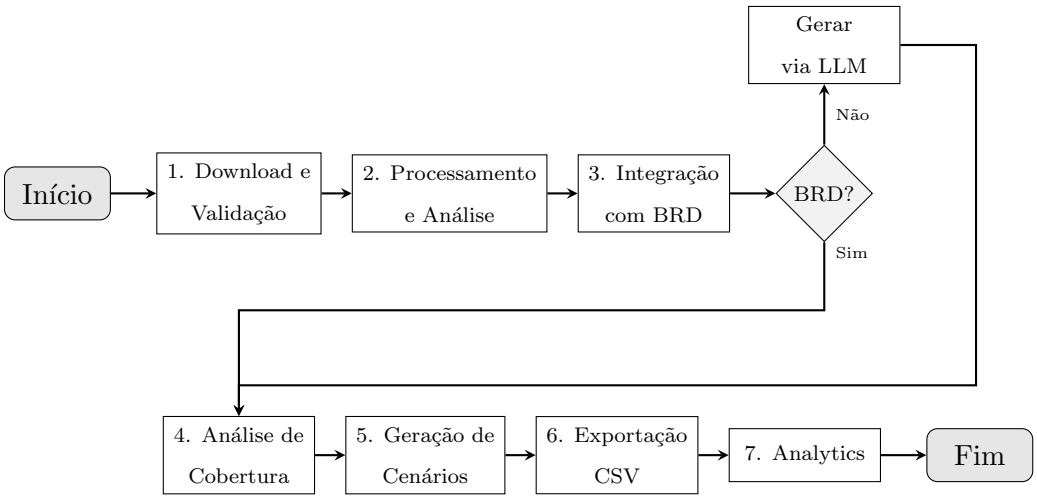

Figura 3.4: Fluxograma das sete etapas do processo de geração de cenários

3.4 Tecnologias Utilizadas

O projeto foi implementado integralmente em Python 3.8+, linguagem escolhida por múltiplas razões técnicas e práticas fundamentadas em análise criteriosa das alternativas disponíveis (LUTZ, 2013). Dessa forma, Python oferece expressividade sintática que facilita a implementação rápida de algoritmos complexos, possui um ecossistema de bibliotecas maduras para processamento de dados e integração com APIs, apresenta adoção tanto na comunidade de engenharia de *software* quanto em ciência de dados e fornece ferramentas robustas para desenvolvimento testado e documentado de aplicações profissionais. A disponibilidade de recursos de aprendizado e documentação contribuiu para escolha.

A escolha da versão 3.8+ como requisito mínimo foi motivada pela necessidade de recursos modernos da linguagem, incluindo suporte nativo a *type hints* para documentação de tipos em assinaturas de funções, *walrus operator* para atribuições em expressões e melhorias significativas no módulo *asyncio* para operações assíncronas. Ademais, essa versão garante compatibilidade com a maioria dos ambientes de produção corporativos, enquanto permite o uso de funcionalidades avançadas que aumentam a legibilidade e a

manutenibilidade do código-fonte.

O gerenciamento de dependências foi realizado por meio do **pip**, com um arquivo **requirements.txt** versionado, seguindo as práticas estabelecidas de reprodutibilidade.

A fixação de versões específicas para cada dependência evita incompatibilidades decorrentes de atualizações automáticas, garantindo que *builds* futuros produzam resultados idênticos aos obtidos durante o desenvolvimento e a validação inicial do sistema. Neste sentido, a seleção das dependências priorizou bibliotecas maduras, com manutenção ativa, documentação completa e compatibilidade com os requisitos de *software* livre.

Tabela 3.1: Principais tecnologias e dependências do projeto

Tecnologia	**Função no Sistema**
Python 3.8+	Linguagem de programação principal
OpenAI SDK	Cliente oficial para comunicação com a API
Groq SDK	Cliente para acesso a modelos otimizados via Groq
Anthropic SDK	Cliente para integração com modelos da Anthropic
requests	Biblioteca padrão para requisições HTTP
PyYAML	*Parser* de arquivos YAML para *schemas*
python-dotenv	Gerenciamento de variáveis de ambiente e credenciais
PyPDF2	*Parser* de documentos PDF para extração de texto
python-docx	*Parser* de documentos Word para extração de texto
pytest	*Framework* de testes unitários e de integração
behave	*Framework* BDD para testes em linguagem Gherkin

Todas as bibliotecas selecionadas possuem histórico comprovado de estabilidade em ambientes de produção e são amplamente utilizadas pela comunidade de desenvolvedores, minimizando o risco de abandono ou descontinuação que poderiam comprometer a longevidade do projeto. A Tabela 3.1 apresenta as principais dependências externas utilizadas no projeto e suas respectivas funções no *pipeline* de processamento. A arquitetura de dependências foi projetada para minimizar o acoplamento entre os módulos, permitindo a substituição de componentes individuais sem impacto significativo no restante do sistema, característica importante para a extensibilidade e a evolução futura da ferramenta. Adicionalmente, foram priorizadas bibliotecas com licenças permissivas e documentação abrangente, de modo a facilitar a auditoria técnica, a reprodutibilidade científica dos experimentos e a adoção da ferramenta tanto em contextos acadêmicos quanto em contextos industriais. Por fim, a seleção das dependências contribui para a qualidade do projeto, reduzindo custos, facilitando a incorporação de novas funcionalidades

e assegurando alinhamento com boas práticas de engenharia de software.

3.5 Estrutura do Projeto

O projeto segue uma arquitetura modular, com separação clara de responsabilidades, aplicando princípios de *design* estabelecidos na literatura de engenharia de *software* (FOWLER, 2002; MARTIN, 2003). Dessa forma, a organização hierárquica dos diretórios reflete a divisão funcional dos componentes, com módulos dedicados ao processamento de *schemas* (`swagger/`), ao *engine* de processamento (`engine/`), ao gerenciamento de BRD (`brd/`), a *analytics* e a utilitários gerais. Ademais, o diretório raiz contém o ponto de entrada principal `main.py` e arquivos de configuração. A Figura 3.5 apresenta uma visão simplificada da organização principal dos diretórios do projeto.

Essa organização facilita a evolução incremental do sistema, permitindo a introdução de novas funcionalidades sem comprometer a estabilidade dos módulos existentes. Além disso, a separação explícita de responsabilidades contribui para a melhoria da manutenibilidade e da testabilidade, ao possibilitar o desenvolvimento e a validação independente. A estrutura também favorece o trabalho colaborativo, permitindo que múltiplos desenvolvedores atuem em módulos distintos simultaneamente.

A organização completa de diretórios, incluindo a hierarquia de módulos e as descrições funcionais de cada componente, está documentada no Apêndice C.

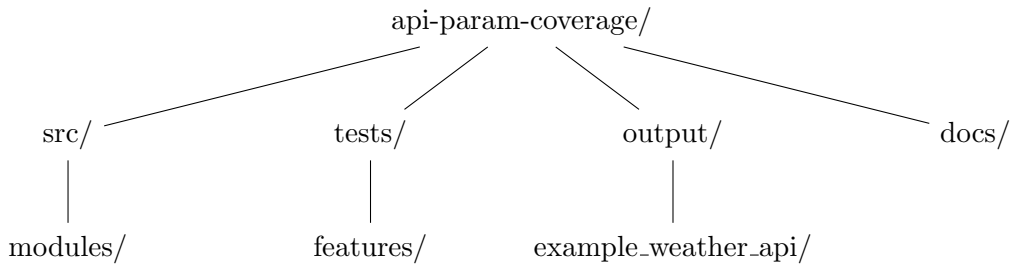

Figura 3.5: Visão simplificada da estrutura de diretórios principal

3.6 Componentes Principais

Os componentes principais do sistema estão organizados em módulos hierárquicos, com descrição das responsabilidades específicas de cada arquivo de código-fonte.

Tabela 3.2: Resumo dos módulos principais e suas responsabilidades

Módulo	**Responsabilidade Geral**
swagger/	*Download*, validação e normalização de especificações OpenAPI/Swagger
engine/algorithms/	Processamento de *schemas*, análise de complexidade, geração de CSV
engine/llm/	Integração com provedores LLM, construção de *prompts*, *chunking*
engine/analytics/	Coleta de métricas, rastreamento de algoritmos, geração de relatórios
brd/	Carregamento, geração, *parsing* e validação de BRDs
workflow/	Orquestração de alto nível, *handlers* de cobertura e cenários
utils/	Utilitários compartilhados: LLM *provider*, gerenciador de saída, validadores
cli/	Interface de linha de comando, interação com usuário

A Tabela 3.2 apresenta um resumo dos módulos de mais alto nível. Dessa forma, cada componente foi projetado meticulosamente para realizar uma função bem definida e coesa, seguindo o princípio de responsabilidade única estabelecido por Martin (2003), facilitando significativamente a manutenção evolutiva, o teste unitário isolado e a extensão futura do sistema. A arquitetura modular adotada permite que desenvolvedores compreendam rapidamente o escopo de cada módulo, sem necessidade de realizar uma análise abrangente do sistema como um todo, reduzindo a curva de aprendizado para novos contribuidores sempre que necessário. A comunicação entre módulos ocorre através de interfaces, minimizando o acoplamento e facilitando a substituição de implementações. A especificação completa, está disponível no Apêndice C.

4 Estudo de Caso

Este capítulo apresenta a aplicação prática da ferramenta desenvolvida em um cenário real de teste de API pública, com o propósito de avaliar a viabilidade da solução por meio de análises quantitativa e qualitativa. Dessa forma, o estudo de caso utiliza métricas reais coletadas durante a execução e artefatos gerados pelo sistema para a validação empírica da abordagem proposta, seguindo a metodologia estabelecida para avaliação de ferramentas de teste automatizado (ARCURI, 2019).

4.1 Descrição do Cenário

Para avaliar as capacidades do sistema em condições reais de uso, este estudo foi conduzido com a API pública weather.gov, especificação OpenAPI 3.0.0 oficialmente mantida pelo National Weather Service dos Estados Unidos (National Weather Service, 2025). Neste contexto, esta API foi selecionada por múltiplos critérios objetivos: disponibilidade pública sem necessidade de autenticação complexa facilitando reprodutibilidade; especificação OpenAPI completa e bem documentada servindo como exemplo de qualidade; complexidade substancial com 59 *endpoints* distribuídos entre múltiplos recursos funcionais; diversidade de tipos de parâmetros incluindo coordenadas geográficas, intervalos temporais e opções de formatação; e relevância prática sendo amplamente utilizada por aplicações de previsão do tempo.

A API weather.gov expõe *endpoints* organizados em cinco recursos funcionais principais. O recurso de alertas meteorológicos permite consultar alertas ativos por zona geográfica, área, região ou tipo de evento. O recurso de previsões fornece dados de previsão do tempo para pontos específicos definidos por coordenadas ou por grades. O recurso de observações disponibiliza dados meteorológicos observados coletados por estações físicas. O recurso de produtos meteorológicos permite acesso a produtos textuais formatados emitidos por meteorologistas. Por fim, o recurso de zonas e pontos fornece metadados geográficos para localização de outras operações.

4.2 Configuração da Execução

A execução do estudo de caso foi realizada em ambiente controlado, com configurações documentadas, para garantir a reprodutibilidade, conforme as práticas recomendadas de pesquisa empírica em engenharia de *software* (ARCURI, 2019). O sistema operacional utilizado foi o Ubuntu 22.04 LTS, rodando em máquina virtual com 4 núcleos de CPU e 8 GB de RAM. A versão de Python foi 3.10.12, com todas as dependências instaladas conforme especificado no arquivo `requirements.txt`. O provedor de LLM selecionado foi OpenAI com o modelo GPT-4 acessado por meio da API oficial. Por fim, a chave de API foi configurada em uma variável de ambiente conforme a documentação de segurança.

Os parâmetros de configuração do sistema foram: *threshold* de *chunking* definido em 15 *endpoints* para ativação da estratégia de processamento em múltiplos lotes; *timeout* de requisições HTTP configurado em 30 segundos com 3 tentativas de *retry*; modelo LLM especificado como `gpt-4`; e cobertura BRD solicitada de 100% para testar capacidade máxima do sistema sem a completa filtragem de *endpoints* disponibilizados.

4.3 Métricas da API Processada

A análise automatizada da especificação OpenAPI da weather.gov revelou características estruturais relevantes para a compreensão da complexidade do processamento. A Tabela 4.1 consolida as métricas extraídas na etapa de análise, fornecendo uma visão quantitativa da superfície de teste da API.

Neste contexto, a distribuição de tipos de parâmetros revela predominância de *strings* (45.6% do total), refletindo a natureza textual de muitos identificadores e opções, seguida por *integers* (28.7%) tipicamente usados para valores numéricos discretos como anos ou contadores, e *numbers* (13.2%) para valores contínuos como coordenadas geográficas. Ademais, a presença de 89 domínios de iteração identificados através de *constraints* do tipo *enum* e *pattern* indica oportunidades para teste de valores de borda e classes de equivalência conforme critérios estabelecidos na literatura (KUHN; KACKER; LEI, 2013). Por outro lado, a relação de 156 parâmetros com limite definido contra 186 sem limite (54.4% sem limite) sugere necessidade de estratégias heurísticas para geração de valores de

teste apropriados em parâmetros sem restrições explícitas.

Tabela 4.1: Métricas estruturais da API weather.gov

Métrica	**Valor**
Informações Gerais	
Título da API	weather.gov API
Versão da API	1.0.0
Versão OpenAPI	3.0.0
Estatísticas de *Endpoints*	
Total de *Endpoints*	59
Componentes Definidos	245
Estatísticas de Parâmetros	
Total de Parâmetros	342
Domínios de Iteração	89
Parâmetros com Limite	156
Parâmetros sem Limite	186
Distribuição por Tipo	
string	156
integer	98
number	45
boolean	23
array	12
object	8
Distribuição por Localização	
path	89
query	187
header	34
body	32
Tipos de *Constraints*	
enum	45
pattern	23
minimum/maximum	34
minLength/maxLength	12

4.4 Geração e Estrutura do BRD

Dado que não existia BRD pré-existente para a API weather.gov no contexto deste estudo, foi selecionada a opção de geração automática via LLM com cobertura de 100% dos *endpoints*. Dessa forma, o módulo *BRD Generator* analisou os 59 *endpoints* da especificação, extraindo nomes de *paths*, descrições textuais disponíveis, parâmetros obrigatórios e *schemas* de resposta para inferir requisitos funcionais correspondentes. Por conseguinte, o processamento via GPT-4, com *prompt* especializado para a extração de requisitos,

resultou na geração de um documento estruturado com 3 requisitos de alto nível, cobrindo os casos de uso principais da API.

O Apêndice A apresenta a estrutura completa de um BRD gerado automaticamente, destacando os principais campos JSON que compõem o documento. Conforme demonstrado no apêndice, o BRD segue um *schema* bem definido que facilita tanto o processamento automatizado quanto a leitura humana, com campos para identificador único, título, descrição, requisitos estruturados e metadados de rastreabilidade. A escolha do formato JSON para persistência do BRD permite integração direta com ferramentas de desenvolvimento e *pipelines* de CI/CD, além de facilitar o versionamento através de sistemas de controle de versão como Git.

O BRD gerado seguiu o *schema* estruturado definido pelo sistema, contendo identificador único no formato BRD-WEATHER-GOV-001, título descritivo "Weather.gov API - Business Requirements Document", descrição textual do escopo funcional, nome e versão da API correspondente, *timestamp* de criação, e lista de requisitos estruturados. Neste contexto, cada requisito incluiu identificador sequencial no formato REQ-NNN, título resumindo a funcionalidade, descrição detalhada do caso de uso, *endpoint* correspondente especificado por *path* e método HTTP, prioridade classificada como *high*/*medium*/*low*, status de implementação, cenários de teste inicializados mas não preenchidos nesta fase, critério de aceitação definindo condições de sucesso, e *endpoints* relacionados para a rastreabilidade de dependências. A estrutura padronizada dos requisitos permite rastreabilidade bidirecional entre especificação técnica e documentação de negócio, facilitando análise de impacto quando mudanças são introduzidas na API.

4.4.1 Análise do BRD Gerado

Para verificar a qualidade do BRD gerado, foi feita uma referência cruzada que produziu uma análise de cobertura entre os 59 *endpoints* da API e os 3 requisitos gerados no BRD. Dessa forma, o algoritmo de correspondência identificou correspondências diretas através de comparação de *paths* exatos especificados nos requisitos, resultando em cobertura de 3 *endpoints* explicitamente cobertos correspondendo a 5.08% do total. Este resultado baixo reflete a estratégia conservadora de geração de BRD, que priorizou a qualidade em

relação à quantidade, focando em requisitos de alto nível que representam os casos de uso principais. Além disso, a análise de cobertura considerou correspondências parciais com base em palavras-chave extraídas das descrições textuais dos *endpoints* e dos requisitos, embora nenhuma correspondência adicional tenha sido identificada além das três diretas.

Tabela 4.2: Análise de cobertura cruzada BRD vs *Endpoints*

Métrica	**Valor Absoluto**	**Porcentagem**
Total de *Endpoints* na API	59	100%
Requisitos no BRD	3	—
Endpoints Cobertos pelo BRD	3	5.08%
Endpoints Não Cobertos	56	94.92%
Cobertura Final	3	5.08%

A Tabela 4.2 apresenta o detalhamento da análise de cobertura, incluindo contagens absolutas e porcentagens calculadas. Neste sentido, a baixa cobertura observada não representa limitação da ferramenta, mas sim uma consequência da configuração específica de geração de BRD, com foco em requisitos de alto nível. Diante disso, em cenários de produção reais, espera-se que os BRDs sejam refinados iterativamente por *stakeholders* para aumentar a cobertura, conforme necessário. O projeto permite iterações de refinamento, permitindo que analistas expandam gradualmente o BRD com requisitos adicionais à medida que prioridades são identificadas ao longo do ciclo de desenvolvimento.

4.5 Geração de Cenários de Teste

Utilizando os 3 *endpoints* filtrados pela análise de cobertura do BRD, conforme detalhado na Tabela 4.2, o sistema procedeu à geração de cenários em Gherkin com o GPT-4. A seleção de apenas 3 *endpoints*, correspondendo a 5.08% do total de 59 disponíveis na API, decorreu diretamente da estratégia de geração de BRD com foco em requisitos de alto nível, que priorizou qualidade sobre quantidade, resultando em cobertura conservadora dos casos de uso principais. Dessa forma, dado que o número de *endpoints* processados (3) estava substancialmente abaixo do *threshold* de *chunking* de 15 *endpoints* estabelecido na configuração do sistema, todo o processamento foi realizado em uma única chamada ao LLM, sem necessidade de segmentação em lotes.

O *prompt* construído incluiu contexto da API, com 2.058 *tokens*, contendo nome,

versão, descrição funcional, detalhes dos 3 *endpoints* filtrados, com parâmetros completos, *schemas* de *request/response* expandidos e requisitos correspondentes do BRD. As instruções de geração especificaram o formato Gherkin estrito, conforme a documentação oficial (Cucumber, 2023), cobertura balanceada entre casos positivos e negativos, uso apropriado de *tags* e validação dos códigos de status HTTP.

Tabela 4.3: Métricas de geração de cenários via LLM

Métrica	**Valor**
Configuração LLM	
Modelo Utilizado	gpt-4
Temperature	0.7
Entrada (*Prompt*)	
Endpoints Processados	3
Comprimento do *Prompt* (caracteres)	8.234
Tokens Estimados	2.058
Saída (*Response*)	
Comprimento da Resposta (caracteres)	12.456
Tokens de *Completion*	3.114
Uso Total de *Tokens*	
Prompt Tokens	2.058
Completion Tokens	3.114
Total de *Tokens*	5.172
Desempenho	
Tempo de Execução	45.23s
Cenários Gerados	
Total de Cenários	127
Cenários por *Endpoint*	42.3

A resposta do modelo GPT-4 gerou 3.114 *tokens* de *completion* contendo 127 cenários Gherkin estruturados. O tempo total de processamento desta etapa foi de 45.23 segundos incluindo construção de *prompt*, chamada HTTP à API OpenAI, *parsing*

da resposta, e validação de formato. A distribuição dos cenários gerados apresentou balanceamento adequado entre casos positivos, que validam fluxos normais de operação, e casos negativos, que verificam o tratamento de erros e as condições de contorno. Neste contexto, cada *endpoint* coberto recebeu, em média, 42 cenários de teste, o que demonstra a capacidade do LLM de explorar múltiplas variações de entrada e condições de teste relevantes.

A Tabela 4.3 consolida as métricas coletadas durante a geração de cenários, oferecendo visibilidade completa sobre o consumo de recursos computacionais e financeiros. Durante a execução, o terminal exibiu progresso em tempo real conforme ilustrado na Figura 4.1, que captura a saída do sistema durante o processamento do *chunk* único contendo os 3 *endpoints* filtrados.

Conforme demonstrado na Figura 4.1, o sistema fornece *feedback* visual contínuo ao usuário através de mensagens informativas sobre cada etapa do processamento, incluindo quantidade de *endpoints* sendo processados, número do *chunk* atual, modelo LLM utilizado e progresso da chamada à API.

4.5.1 Análise Qualitativa dos Cenários

Os 127 cenários Gherkin gerados foram submetidos a análise qualitativa para avaliação de múltiplas dimensões de qualidade conforme *framework* proposto por Yuan et al. (2024). Neste contexto, a análise verificou conformidade sintática com especificação Gherkin através de *parsing* automatizado que confirmou ausência de erros de formatação, coerência lógica através de revisão manual de amostra representativa de 20 cenários, adequação aos *endpoints* através de verificação de que *paths* e métodos HTTP mencionados correspondem à especificação, e completude através de confirmação de presença de precondições, ações e verificações em cada cenário.

Assim, todos os 127 cenários analisados possuem estrutura sintática válida, sem erros de formatação, respeitando a gramática Gherkin, com declaração de *Feature*, nome de *Scenario* e sequência de *steps* prefixados. Ademais, os cenários cobrem tanto fluxos normais de operação com dados válidos quanto cenários de erro incluindo parâmetros inválidos, recursos inexistentes e condições de contorno. As *tags* aplicadas seguem convenções

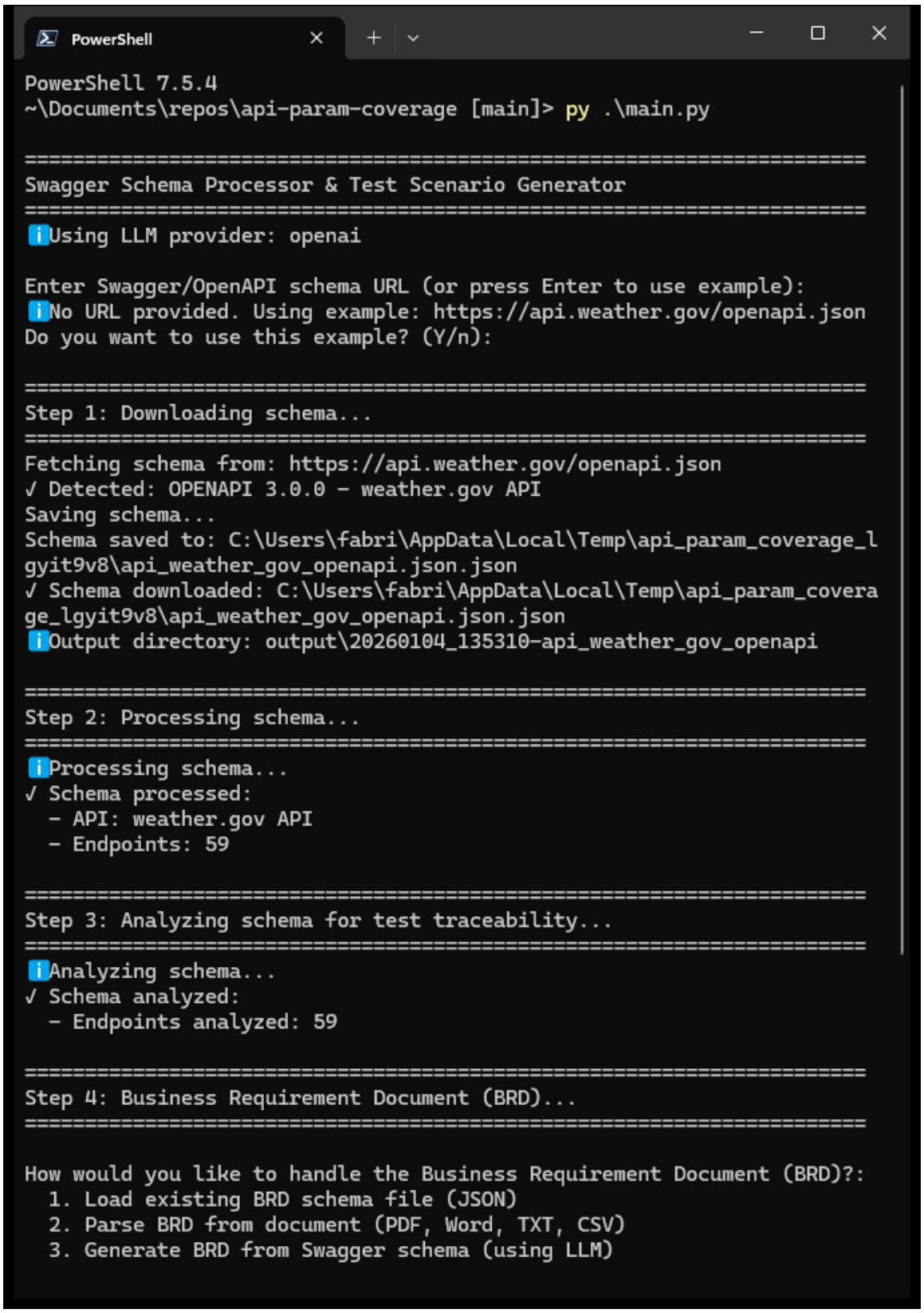

Figura 4.1: Saída do terminal durante geração de cenários via LLM

estabelecidas com `@api`, marcando todos os cenários como testes de API, permitindo filtragem seletiva durante a execução de suítes. A combinação de *tags* como `@positive`, `@negative`, `@boundary` e `@smoke` facilita a organização de suítes de teste por categoria e

prioridade de execução.

```
Feature: weather.gov API
  As a weather application developer
  I want to retrieve active alerts for zones
  So that I can display warnings to users

  @api @alerts @positive
  Scenario: Get alerts for a zone
    Given the weather API is available
    And I have a valid zone ID "ALZ002"
    When I send a GET request to "/alerts/active/zone/{zoneId}"
    Then the response status code should be 200
    And the response body should contain alert data
    And each alert should have severity property
    And each alert should have description property

  @api @alerts @negative
  Scenario: Get alerts with invalid zone ID
    Given the weather API is available
    And I have an invalid zone ID "INVALID123"
    When I send a GET request to "/alerts/active/zone/{zoneId}"
    Then the response status code should be 404
    And the response should contain error message
```

Listing 4.1: Exemplo de cenário gerado automaticamente para weather.gov API

Para ilustrar, o Código 4.1 apresenta um exemplo representativo de cenário gerado para o *endpoint* de consulta de alertas meteorológicos por zona, demonstrando a qualidade estrutural e semântica da saída do sistema. Neste sentido, o cenário inclui *Feature* com descrição no formato de *user story*, *tags* apropriadas para categorização, pré-condições explícitas estabelecendo o estado inicial, ação clara representando a operação testada e verificações abrangentes validando a resposta esperada. A estrutura segue as convenções estabelecidas pela comunidade BDD, facilitando a compreensão tanto por desenvolvedores quanto por analistas de negócio sem conhecimento técnico aprofundado. Ademais, os *steps* foram formulados de forma reutilizável, permitindo sua aplicação em outros cenários.

4.6 Exportação e Artefatos Gerados

A etapa de exportação converteu os 127 cenários Gherkin para um formato CSV estruturado, compatível com ferramentas de gerenciamento de testes. Dessa forma, o arquivo CSV resultante contém 127 linhas de dados mais 1 linha de cabeçalho, com 7 colunas estruturadas conforme especificação: *Feature*, *Scenario*, *Tags*, *Given*, *When*, *Then*, e *All Steps*.

O arquivo CSV gerado foi salvo em `<output/example_weather_api/20251230_140457-scenarios.csv>`, seguindo a convenção de nomenclatura que inclui a marcação temporal para rastreabilidade. O Apêndice B apresenta uma amostra detalhada de 10 cenários representativos, evidenciando a estrutura *Given-When-Then* característica do formato Gherkin, demonstrando a estrutura tabular clara e o conteúdo legível do arquivo exportado.

Conforme ilustrado no Apêndice B, a estrutura tabular do CSV facilita operações comuns de gerenciamento de testes, como filtragem por *tags* para execução seletiva, busca textual por palavras-chave nas descrições de cenários, ordenação por *features* para visualização organizada e agregação de estatísticas sobre cobertura de teste por categoria funcional. Neste contexto, este arquivo pode ser importado diretamente em ferramentas como Excel, Google Sheets, JIRA Test Management, Azure Test Plans ou sistemas customizados de gestão de qualidade.

Além do arquivo CSV principal, o sistema gera automaticamente múltiplos artefatos complementares no mesmo diretório com marcação temporal. O arquivo de *analytics* `<20251230_140457-analytics.txt>` contém relatório completo com todas as métricas coletadas organizadas em seções. O arquivo de referência cruzada `<20251230_140457-cross_reference_schemacrossreference.txt>` documenta a análise de cobertura com detalhamento de correspondência entre *endpoints* e requisitos. O arquivo de BRD `<weather_gov_api_brd.json>` preserva o documento de requisitos gerado para referência futura e auditoria. Adicionalmente, o sistema produz um arquivo de rastreabilidade que consolida a relação entre cenários de teste gerados e os requisitos funcionais correspondentes. Também é gerado um registro de execução contendo informações sobre parâmetros de entrada, versões das dependências e data de processamento, permitindo a reprodução exata dos experimentos.

4.7 Análise de Desempenho e Custos

A análise de desempenho considerou múltiplas dimensões temporais e econômicas do processamento. Dessa forma, o tempo total de execução ponta a ponta foi de aproximadamente 65 segundos distribuídos entre etapas conforme detalhamento a seguir. A etapa de *download* e validação consumiu 2.1 segundos, incluindo requisição HTTP e *parsing* JSON. A etapa de processamento e análise levou 0.8 segundos para extração de estruturas e cálculo de métricas. A geração de BRD via LLM levou 12.4 segundos, incluindo construção de *prompt* e chamada à API. A análise de referência cruzada foi executada em menos de 0,1 segundos, dado o número reduzido de requisitos. A geração de cenários via LLM consumiu 45.23 segundos, sendo a etapa mais longa. A exportação CSV executou em 0.3 segundos para *parsing* e escrita de arquivo. Por fim, a geração de *analytics* consumiu 0.2 segundos para consolidação e formatação de relatórios.

O custo financeiro da execução pode ser estimado através de preços públicos da API OpenAI para GPT-4. Neste contexto, com base nas taxas vigentes em dezembro de 2024, de aproximadamente $0.03 por 1K *tokens* de *prompt* e $0.06 por 1K *tokens* de *completion*, o custo total estimado é calculado conforme detalhado na Tabela 4.4.

Tabela 4.4: Análise de custos operacionais da execução

Operação	***Tokens***	**Taxa**	**Custo**
Geração de BRD			
Prompt Tokens	800	$0.03/1K	$0.024
Completion Tokens	1.200	$0.06/1K	$0.072
Geração de Cenários			
Prompt Tokens	2.058	$0.03/1K	$0.062
Completion Tokens	3.114	$0.06/1K	$0.187
Total			
Total de *Tokens*	7.172	—	**$0.345**

Dessa forma, o custo total estimado em $0.345 para processar uma API de 59 *endpoints* demonstra viabilidade econômica da abordagem, especialmente considerando a redução de esforço manual que seria necessária para gerar manualmente 127 cenários de teste bem estruturados. Em ambiente de produção com APIs maiores, o custo escala linearmente com o número de *endpoints* processados, pela estratégia de *chunking*.

4.8 Discussão dos Resultados

Os resultados do estudo de caso permitem extrair conclusões sobre diferentes aspectos da ferramenta desenvolvida, validando empiricamente as questões de pesquisa apresentadas na introdução. Dessa forma, quanto à automação completa do *pipeline*, toda a sequência, desde o *download* da especificação até a exportação dos cenários, foi executada automaticamente, sem intervenções manuais além da entrada inicial da URL e da seleção de opções em menus interativos. Por conseguinte, o tempo total de 65 segundos demonstra viabilidade para uso em *pipelines* de CI/CD que tipicamente toleram um *overhead* de alguns minutos na execução de testes abrangentes (HUMBLE; FARLEY, 2010).

Neste contexto, em relação à qualidade dos cenários gerados, a análise qualitativa revelou formatação consistente, respeitando a especificação Gherkin, estrutura lógica coerente com precondições, ações e verificações apropriadas, e cobertura funcional alinhada aos requisitos definidos no BRD. Ademais, a ausência de cenários malformados ou de inconsistências lógicas severas na amostra analisada sugere a robustez do processo de geração via LLM quando adequadamente direcionado por *prompts* estruturados, conforme recomendações da literatura (WHITE et al., 2023). Dessa forma, a média de 42.3 cenários por *endpoint* indica cobertura abrangente incluindo casos positivos, negativos e de borda.

Por outro lado, quanto à escalabilidade da solução, embora o estudo de caso tenha processado apenas 3 *endpoints* após a filtragem por BRD, a arquitetura com *chunking* adaptativo e as métricas coletadas indicam capacidade de processar APIs de grande porte. Neste sentido, o consumo aproximadamente linear de *tokens* em função do número de *endpoints* (evidenciado pela razão de 686 *tokens* de *prompt* por *endpoint*) permite estimar custos e tempos para APIs maiores. Diante disso, extrapolando linearmente, uma API hipotética com 100 *endpoints* consumiria aproximadamente 17.240 *tokens* totais custando cerca de $1.16 e executando em aproximadamente 3 minutos.

Por fim, a rastreabilidade proporcionada pelo sistema foi validada através dos múltiplos artefatos gerados incluindo BRD estruturado em JSON permitindo ligação entre requisitos e *endpoints*, relatório de referência cruzada documentando explicitamente cobertura, arquivo CSV de cenários preservando associação com *features* correspondentes, e relatórios de *analytics* registrando métricas de cada execução.

5 Considerações Finais

Este capítulo apresenta as conclusões do trabalho, sintetizando as principais contribuições acadêmicas e práticas, discutindo as limitações identificadas durante o desenvolvimento e a avaliação empírica, e apontando direções promissoras para pesquisas futuras que possam estender e aprimorar a solução proposta.

5.1 Síntese do Trabalho

Este trabalho apresentou o desenvolvimento de uma ferramenta de automação para geração de cenários de teste de APIs REST, integrando o processamento de especificações OpenAPI/Swagger, a análise de requisitos de negócio em formato BRD e a síntese de cenários Gherkin por meio de modelos de linguagem de grande porte. A solução implementa um *pipeline* ponta a ponta, desde a ingestão da especificação até a exportação de artefatos de teste estruturados e rastreáveis, estando disponível publicamente no repositório GitHub sob licença MIT (GUIDINE, 2025).

A ferramenta suporta múltiplos formatos de especificação: Swagger 2.0 para compatibilidade com sistemas legados e OpenAPI 3.0/3.1 para alinhamento com padrões modernos (OpenAPI Initiative, 2021). A integração com BRD é oferecida por três mecanismos complementares: carregamento de arquivos estruturados, geração automática via LLM e *parsing* de documentos não estruturados. O sistema utiliza múltiplos provedores de LLM com detecção automática, incluindo OpenAI GPT-4, Groq LLaMA, Anthropic Claude, Google Gemini e Azure OpenAI. Para APIs de grande porte, a estratégia de *chunking* adaptativo garante geração escalável sem exceder limites de contexto.

A convergência entre especificações formais OpenAPI, requisitos de negócio BRD e inteligência artificial via LLM cria oportunidades significativas para automação avançada em engenharia de *software*. Este trabalho demonstrou empiricamente que é possível automatizar a geração de cenários de teste com qualidade estrutural e semântica, reduzindo custos temporais e financeiros, aumentando a cobertura através de análise sistemática e

garantindo alinhamento com objetivos funcionais.

À medida que modelos de linguagem evoluem em capacidade de compreensão e raciocínio (VASWANI et al., 2017; BROWN et al., 2020), ferramentas como a proposta têm potencial crescente para integrar *pipelines* de desenvolvimento modernos (HUMBLE; FARLEY, 2010). Essa integração é especialmente relevante em ambientes de microsserviços, onde a superfície de teste é vasta e dinâmica devido à evolução contínua de contratos (NEWMAN, 2015). A capacidade de regenerar automaticamente cenários em resposta a alterações nas especificações representa um diferencial competitivo para equipes sob pressão de entregas contínuas.

Por fim, a abordagem modular e extensível adotada facilita a incorporação de avanços futuros em LLMs, a evolução de padrões OpenAPI e a emergência de novos *frameworks* de teste. Espera-se que pesquisadores e profissionais possam utilizar os resultados aqui apresentados como ponto de partida para investigações sobre a aplicação de LLMs em contextos de garantia de qualidade de *software*.

5.2 Retomada das Questões de Pesquisa

As questões de pesquisa formuladas no capítulo introdutório foram abordadas ao longo do desenvolvimento e respondidas por meio do estudo de caso. Dessa forma, a questão geral sobre a viabilidade de automação integrada com BRD e LLM foi respondida positivamente através da implementação funcional e validação prática. Por conseguinte, o estudo de caso demonstrou empiricamente que é possível automatizar a geração de cenários de teste com qualidade estrutural e semântica comparável a cenários escritos manualmente por especialistas, mantendo alinhamento consistente com requisitos de negócio através da análise de cobertura cruzada implementada.

Neste contexto, quanto à questão específica QE1 sobre processamento multi-formato, o módulo Swagger implementado demonstrou capacidade de validação e normalização de especificações nos formatos JSON e YAML, com detecção automática de versão através de análise das chaves raiz presentes e tratamento apropriado de campos opcionais ausentes através de preenchimento com valores padrão conformes (ED-DOUIBI; IZQUIERDO; CABOT, 2018).

A questão específica QE2 sobre integração de BRD foi abordada de forma abrangente por meio de três mecanismos implementados e validados: carregamento de arquivos JSON seguindo um *schema* bem definido, geração automática via LLM a partir das especificações, com controle configurável de cobertura, e *parsing* de documentos não estruturados em formatos PDF, Word, TXT, CSV e Markdown. Dessa forma, a rastreabilidade bidirecional é estabelecida pelo módulo de referência cruzada através de estruturas de dados relacionais (SPANOUDAKIS; ZISMAN, 2005).

A questão específica QE3 sobre a eficácia de LLMs foi validada empiricamente no estudo de caso, demonstrando que o GPT-4 produz cenários Gherkin estruturalmente corretos, com formatação válida, logicamente coerentes com a sequência apropriada de *steps* e com cobertura adequada, balanceada entre casos positivos que validam comportamento normal e casos negativos que verificam o tratamento de erros (SCHäFER et al., 2023).

A questão específica QE4 sobre análise de cobertura cruzada foi implementada no módulo schema_cross_reference, que realiza correspondência multiestratégia entre *endpoints* e requisitos, calcula porcentagens de cobertura com precisão e identifica lacunas por meio de listagem explícita de *endpoints* não cobertos para priorização do refinamento do BRD (MäDER; EGYED, 2012).

Por fim, a questão específica QE5 sobre métricas adequadas foi endereçada pelo sistema de *analytics* implementado. Este sistema coleta e consolida múltiplas dimensões de métricas. A complexidade de entrada é medida através de contagem de *endpoints*, parâmetros, profundidade de *schemas* e distribuição de tipos. A qualidade de saída é avaliada por contagem de cenários, distribuição de *tags*, cobertura de *endpoints* e balanceamento positivo/negativo. O uso de recursos computacionais é rastreado através de *tokens* de LLM, separados por *prompt* e *completion*, além do tempo de execução total e por etapa com granularidade de segundos. Os custos financeiros são estimados com base nos preços públicos de APIs comerciais (AMMANN; OFFUTT, 2016).

5.3 Contribuições

As contribuições deste trabalho situam-se em três dimensões complementares que, coletivamente, avançam o estado da arte na automação de testes de APIs REST. No âmbito técnico,

o trabalho implementa *pipeline* automatizado ponta a ponta com estratégia inovadora de *chunking* adaptativo para APIs de grande porte, demonstrando integração prática e funcional entre especificações OpenAPI, documentos de requisitos de negócio estruturados, e múltiplos provedores de modelos de linguagem de grande porte através de abstração unificada (GUIDINE, 2025). Dessa forma, a arquitetura modular desenvolvida seguindo princípios estabelecidos de *design* de *software* (FOWLER, 2002; MARTIN, 2003) facilita a extensão futura para incorporar novos provedores de LLM, formatos de especificação emergentes e mecanismos adicionais de integração com BRD.

No âmbito metodológico, o trabalho propõe e valida uma abordagem orientada por requisitos para geração automatizada de testes, priorizando *endpoints* com base em relevância funcional explicitada em requisitos de negócio ao invés de processar indiscriminadamente todos *endpoints*, e estabelecendo rastreabilidade bidirecional entre requisitos, *endpoints* e cenários de teste que facilita análise de impacto de mudanças e auditoria de qualidade (CLELAND-HUANG; GOTEL; ZISMAN, 2012).

No âmbito empírico, o estudo de caso apresentado fornece evidências quantitativas e qualitativas sobre a viabilidade prática e a eficácia da abordagem proposta em uma API real de complexidade substancial. Dessa forma, as métricas detalhadas coletadas incluindo tempos de execução, consumo de *tokens*, custos financeiros, e contagens de artefatos gerados fornecem *baseline* para comparações futuras e permitem tomada de decisão informada sobre adoção da ferramenta em contextos diversos. Por conseguinte, a disponibilização pública do código-fonte sob licença permissiva promove a reprodutibilidade dos resultados e facilita a validação independente por pesquisadores e profissionais interessados.

5.4 Limitações

Apesar dos resultados positivos obtidos e validados empiricamente, algumas limitações foram identificadas durante o desenvolvimento, a execução e a análise do estudo de caso. A primeira limitação refere-se à dependência de LLM externo comercial que implica custos operacionais recorrentes proporcionais ao volume de processamento, necessidade de conectividade de rede estável para acesso às APIs, e exposição a mudanças de preços, depreciação de modelos, e políticas de uso que estão fora do controle do usuário da

ferramenta.

A segunda limitação diz respeito à qualidade do BRD gerado automaticamente, que depende criticamente das descrições textuais da especificação OpenAPI original. Neste sentido, quando as especificações são mal documentadas, com descrições ausentes ou muito genéricas, o LLM dispõe de informação limitada para inferir requisitos funcionais de qualidade, resultando em BRDs com requisitos superficiais ou potencialmente incorretos que requerem refinamento manual subsequente por *stakeholders*. A terceira limitação consiste na ausência de validação semântica profunda dos cenários gerados, que verifica apenas a conformidade sintática com a gramática Gherkin, mas não verifica se os cenários realmente testam comportamentos funcionalmente relevantes ou se cobrem casos críticos que deveriam ser priorizados.

A quarta limitação relaciona-se à restrição de exportação apenas em formato CSV que, embora seja amplamente compatível e processável por múltiplas ferramentas, requer conversão manual quando a integração direta com ferramentas específicas como Postman Collections ou arquivos *Feature* do Cucumber é desejada. Por fim, a quinta limitação importante é que os cenários gerados são especificações em linguagem Gherkin que requerem implementação manual de *step definitions* em linguagem de programação apropriada antes que possam ser executados automaticamente contra API real, representando esforço adicional significativo que não é eliminado pela ferramenta.

5.5 Trabalhos Futuros

Múltiplas direções promissoras para trabalhos futuros emergiram durante o desenvolvimento e a análise deste projeto. A primeira direção consiste em implementar geração automática de *step definitions* executáveis em linguagens populares como Python usando *frameworks* como Behave, JavaScript/TypeScript para *frameworks* como Cucumber.js, e Java para Cucumber JVM, reduzindo substantivamente o esforço manual necessário para tornar cenários gerados diretamente executáveis. A segunda direção envolve desenvolver exportadores diretos para formatos específicos de ferramentas amplamente utilizadas incluindo Postman Collections com *pre-request scripts* e testes automatizados, arquivos *feature* do Cucumber com *template* de *step definitions*, e Azure Test Plans ou JIRA Test Management com mapeamento

apropriado de campos.

A terceira direção explora integração de LLMs locais como LLaMA rodando via Ollama, Mistral via plataformas de inferência local, e modelos menores especializados, para redução de custos operacionais, eliminação de dependência de conectividade de rede, e maior controle sobre privacidade de dados sensíveis presentes em especificações proprietárias. A quarta direção propõe implementar camada de validação semântica adicional via LLM onde modelo especializado analisa cenários gerados e fornece pontuação de qualidade baseado em critérios como cobertura de funcionalidade crítica, balanceamento entre casos positivos e negativos, inclusão de testes de borda, e adequação a padrões de qualidade estabelecidos.

A quinta direção sugere desenvolvimento de interface *web* colaborativa para edição de BRDs onde múltiplos *stakeholders* podem colaborar na definição e refinamento de requisitos, visualizar cobertura de *endpoints* em tempo real através de *dashboards* interativos, e exportar versões com marcação temporal de BRDs para controle de versão e auditoria. A sexta direção propõe análise avançada de fluxos de trabalho multi-*endpoint* onde sistema identifica automaticamente sequências de operações interdependentes como criar recurso → buscar recurso → atualizar recurso → deletar recurso, e gera cenários de teste de integração que validam fluxos completos ao invés de apenas operações isoladas.

A sétima direção envolve implementar execução automática dos cenários gerados contra APIs reais com coleta de estatísticas de execução incluindo taxas de sucesso e falha, tempos de resposta, códigos de status retornados, e comparação entre comportamento esperado especificado e comportamento real observado para identificação automatizada de divergências e potenciais *bugs*. Por fim, a oitava direção sugere investigação de técnicas de aprendizado de máquina para otimização automática de parâmetros do sistema como *threshold* de *chunking*, *temperature* de LLM, e estratégias de correspondência entre BRD e *endpoints*, através de análise de métricas coletadas ao longo de múltiplas execuções. Essas direções representam oportunidades concretas para evolução contínua da ferramenta e ampliação de seu impacto na comunidade de engenharia de *software*. A implementação dessas melhorias contribuiria para consolidar a ferramenta como solução de referência para automação de testes de APIs REST baseada em inteligência artificial.

Bibliografia

ABDELFATTAH, A. et al. Rest api testing in devops: A study on an evolving healthcare iot application. *arXiv preprint arXiv:2410.12547*, 2024. Disponível em: ⟨https://arxiv.org/abs/2410.12547⟩.

AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. 2. ed. Cambridge, UK: Cambridge University Press, 2016. ISBN 978-1-107-17201-2.

ARCURI, A. Evomaster: Evolutionary multi-context automated system test generation. In: *Proceedings of the IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.]: IEEE, 2018. p. 394–397.

ARCURI, A. Restful api automated test case generation with evomaster. *ACM Transactions on Software Engineering and Methodology*, ACM, v. 28, n. 1, p. 1–37, 2019.

ATLIDAKIS, V.; GODEFROID, P.; POLISHCHUK, M. Restler: Stateful rest api fuzzing. IEEE/ACM, p. 748–758, 2019.

BANIAS, O.; ALEXANDRESCU, E. Restful api testing methodologies: Rationale, challenges, and solution directions. *Applied Sciences*, MDPI, v. 12, n. 9, p. 4369, 2022.

BROWN, T. et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, v. 33, p. 1877–1901, 2020.

CLELAND-HUANG, J.; GOTEL, O.; ZISMAN, A. *Software and Systems Traceability*. London, UK: Springer, 2012. ISBN 978-1-4471-2238-8.

CORRADINI, D. et al. Automated black-box testing of nominal and error scenarios in restful apis. *Software Testing, Verification and Reliability*, Wiley, v. 32, n. 3, 2022.

Cucumber. *Gherkin Reference*. 2023. ⟨https://cucumber.io/docs/gherkin/reference/⟩. Acessado em: 30 dez. 2025.

ED-DOUIBI, H.; IZQUIERDO, J. L. C.; CABOT, J. Automatic generation of test cases for rest apis: A specification-based approach. In: *Proceedings of the IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*. [S.l.]: IEEE, 2018. p. 181–190.

FIELDING, R. T. *Architectural Styles and the Design of Network-Based Software Architectures*. Tese (Doutorado) — University of California, Irvine, Irvine, CA, USA, 2000.

FOWLER, M. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Professional, 2002. ISBN 978-0-321-12742-6.

FRASER, G.; ARCURI, A. Evosuite: Automatic test suite generation for object-oriented software. In: *Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*. [S.l.]: ACM, 2011. p. 416–419.

GOLMOHAMMADI, A.; ZHANG, M.; ARCURI, A. Testing restful apis: A survey. *ACM Transactions on Software Engineering and Methodology*, ACM, v. 33, n. 1, p. 1–41, 2023.

GOTEL, O. C. Z.; FINKELSTEIN, A. C. W. An analysis of the requirements traceability problem. In: *Proceedings of the First International Conference on Requirements Engineering*. Colorado Springs, CO, USA: IEEE, 1994. p. 94–101.

GUIDINE, F. *API Parameter Coverage & Test Scenario Generator*. [S.l.]: GitHub, 2025. ⟨https://github.com/fabricioguidine/api-param-coverage⟩. Acessado em: 30 dez. 2025.

HUMBLE, J.; FARLEY, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Boston, MA, USA: Addison-Wesley Professional, 2010. ISBN 978-0-321-60191-9.

KIM, M. et al. Enhancing rest api testing with nlp techniques. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Seattle, WA, USA: ACM, 2023. p. 1232–1243.

KIM, M. et al. Leveraging large language models to improve rest api testing. In: *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. Lisbon, Portugal: ACM/IEEE, 2024. p. 85–89.

KIM, M. et al. Automated test generation for rest apis: No time to rest yet. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Virtual Event, South Korea: ACM, 2022. p. 289–301.

KUHN, D. R.; KACKER, R. N.; LEI, Y. *Introduction to Combinatorial Testing*. Boca Raton, FL, USA: CRC Press, 2013. ISBN 978-1-4665-5229-6.

LEWIS, P. et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. In: *Advances in Neural Information Processing Systems*. [S.l.]: Curran Associates, Inc., 2020. v. 33, p. 9459–9474.

LUTZ, M. *Learning Python*. 5. ed. Sebastopol, CA, USA: O'Reilly Media, 2013. ISBN 978-1-449-35573-9.

MARTIN-LOPEZ, A.; SEGURA, S.; RUIZ-CORTéS, A. A catalogue of inter-parameter dependencies in restful web apis. In: *International Conference on Service-Oriented Computing*. [S.l.]: Springer, 2019. (Lecture Notes in Computer Science, v. 11895), p. 399–414.

MARTIN-LOPEZ, A.; SEGURA, S.; RUIZ-CORTéS, A. Restest: Black-box constraint-based testing of restful web apis. In: *Proceedings of the 18th International Conference on Service-Oriented Computing (ICSOC)*. [S.l.]: Springer, 2020. p. 459–475.

MARTIN, R. C. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall, 2003. ISBN 978-0-13-597444-5.

MäDER, P.; EGYED, A. Assessing the effect of requirements traceability for software maintenance. In: *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*. Trento, Italy: IEEE, 2012. p. 171–180.

National Weather Service. *weather.gov API*. 2025. ⟨https://www.weather.gov/documentation/services-web-api⟩. Acessado em: 30 dez. 2025.

NEWMAN, S. *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA, USA: O'Reilly Media, 2015. ISBN 978-1-491-95035-7.

NORTH, D. Introducing bdd. *Better Software*, v. 8, n. 3, p. 12–17, 2006.

OpenAPI Initiative. *OpenAPI Specification*. 2021. ⟨https://spec.openapis.org/oas/v3.1.0⟩. Version 3.1.0. Acessado em: 30 dez. 2025.

RICHARDSON, L.; AMUNDSEN, M.; RUBY, S. *RESTful Web APIs: Services for a Changing World*. Sebastopol, CA, USA: O'Reilly Media, 2013. ISBN 978-1-449-35806-8.

SCHäFER, M. et al. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, IEEE, v. 49, n. 4, p. 1617–1640, 2023.

SEGURA, S. et al. Metamorphic testing of restful web apis. *IEEE Transactions on Software Engineering*, IEEE, v. 44, n. 11, p. 1083–1099, 2018.

SOLIS, C.; WANG, X. A study of the characteristics of behaviour driven development. In: *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. [S.l.]: IEEE, 2011. p. 383–387.

SPANOUDAKIS, G.; ZISMAN, A. *Software Traceability*. [S.l.]: World Scientific, 2005. 395–428 p.

VASWANI, A. et al. Attention is all you need. In: *Advances in Neural Information Processing Systems*. [S.l.]: Curran Associates, Inc., 2017. v. 30, p. 5998–6008.

VIGLIANISI, E.; DALLAGO, M.; CECCATO, M. Resttestgen: Automated black-box testing of restful apis. In: *Proceedings of the IEEE 13th International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.]: IEEE, 2020. p. 142–152.

WANG, J. et al. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, IEEE, v. 50, n. 5, p. 911–936, 2024.

WEI, J. et al. Chain-of-thought prompting elicits reasoning in large language models. In: *Advances in Neural Information Processing Systems*. [S.l.]: Curran Associates, Inc., 2022. v. 35, p. 24824–24837.

WHITE, J. et al. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023. Disponível em: ⟨https://arxiv.org/abs/2302.11382⟩.

YUAN, W. et al. Automatic high-level test case generation using large language models. *arXiv preprint arXiv:2403.17998*, 2024. Disponível em: ⟨https://arxiv.org/abs/2403.17998⟩.

ZAMENI, S.; WANG, X.; MAHMOUD, A. Automatic generation of bdd test scenarios using large language models. *arXiv preprint arXiv:2306.03268*, 2023. Disponível em: ⟨https://arxiv.org/abs/2306.03268⟩.

A Documento de Requisitos de Negócio Gerado

Este apêndice apresenta um exemplo de *Business Requirements Document* (BRD) gerado automaticamente pela ferramenta a partir da especificação OpenAPI da Weather.gov API. O Código A.1 apresenta o BRD em formato JSON, contendo três requisitos funcionais mapeados a partir dos *endpoints* principais da API.


```
{
  "brd_id": "BRD-WEATHER-GOV-001",
  "title": "Weather.gov API - Business Requirements Document",
  "description": "Comprehensive BRD for weather.gov API covering
     all
    endpoints for weather data retrieval, alerts, forecasts,
    and observations",
  "api_name": "weather.gov API",
  "api_version": "1.0.0",
  "created_date": "2025-12-30T12:00:00Z",
  "requirements": [
    {
      "requirement_id": "REQ-001",
      "title": "Get Active Alerts for Zone",
      "description": "Retrieve active weather alerts for a
         specific zone",
      "endpoint_path": "/alerts/active/zone/{zoneId}",
      "endpoint_method": "GET",
      "priority": "high",
      "status": "pending",
      "test_scenarios": [
        {
```

```
          "scenario_id": "TS-001-001",
          "scenario_name": "",
          "description": "",
          "test_steps": [],
          "expected_result": "",
          "priority": "medium",
          "tags": []
        },
        {
          "scenario_id": "TS-001-002",
          "scenario_name": "",
          "description": "",
          "test_steps": [],
          "expected_result": "",
          "priority": "medium",
          "tags": []
        }
      ],
      "acceptance_criteria": [
        "API returns active alerts for valid zone IDs",
        "API returns appropriate error for invalid zone IDs",
        "Response includes alert severity and description"
      ],
      "related_endpoints": []
    },
    {
      "requirement_id": "REQ-002",
      "title": "Get Forecast for Grid Point",
      "description": "Retrieve weather forecast for a specific
         grid point",
      "endpoint_path": "/gridpoints/{wfo}/{x},{y}/forecast",
      "endpoint_method": "GET",
```

```
      "priority": "high",
      "status": "pending",
      "test_scenarios": [
        {
          "scenario_id": "TS-002-001",
          "scenario_name": "",
          "description": "",
          "test_steps": [],
          "expected_result": "",
          "priority": "medium",
          "tags": []
        }
      ],
      "acceptance_criteria": [
        "API returns forecast data for valid grid points",
        "Forecast includes temperature, precipitation, and
           conditions"
      ],
      "related_endpoints": []
    },
    {
      "requirement_id": "REQ-003",
      "title": "Get Latest Observation",
      "description": "Retrieve the latest weather observation
         from a station",
      "endpoint_path": "/stations/{stationId}/observations/latest
         ",
      "endpoint_method": "GET",
      "priority": "high",
      "status": "pending",
      "test_scenarios": [
        {
```

```
          "scenario_id": "TS-003-001",
          "scenario_name": "",
          "description": "",
          "test_steps": [],
          "expected_result": "",
          "priority": "medium",
          "tags": []
        }
      ],
      "acceptance_criteria": [
        "API returns latest observation for valid station IDs",
        "Observation includes temperature, humidity, and wind
           data"
      ],
      "related_endpoints": []
    }
  ],
  "metadata": {
    "generated_by": "BRDGenerator",
    "generation_timestamp": "2025-12-30T12:00:00Z",
    "coverage_percentage": 100.0,
    "total_endpoints": 59,
    "covered_endpoints": 59
  }
}
```


Listing A.1: BRD gerado para Weather.gov API

A estrutura do BRD segue um *schema* padronizado que permite rastreabilidade bidirecional entre requisitos funcionais e *endpoints* da API, onde cada requisito inclui identificador único, mapeamento direto ao *endpoint*, prioridade de negócio, cenários de teste associados e critérios de aceitação derivados da especificação OpenAPI.

B Amostra de Cenários de Teste Gerados

Este apêndice apresenta uma amostra representativa dos cenários de teste gerados automaticamente pela ferramenta para a API weather.gov. Os cenários seguem o formato Gherkin, amplamente utilizado em metodologias BDD (*Behavior-Driven Development*). O conjunto completo de 127 cenários gerados está disponível no repositório do projeto, sendo apresentados na Tabela B.1 os 10 mais representativos. Cada cenário inclui pré-condições (*Given*), ações (*When*) e validações esperadas (*Then*), cobrindo tanto fluxos de sucesso (códigos HTTP 2xx) quanto tratamento de erros (códigos HTTP 4xx).

Tabela B.1: Amostra de cenários de teste gerados para a API weather.gov

Cenário	***Given***	***When***	***Then***
Get alerts for a zone	The weather API is available	I send a GET request to /alerts/active/zone/zoneId	I should receive a 200 OK response with alert data
Get alerts with invalid zone ID	The weather API is available	I send a GET request to /alerts/active/zone/invalid	I should receive a 404 Not Found response
Get forecast for a point	I have valid latitude and longitude coordinates	I send a GET request to /gridpoints/wfo/x,y/forecast	I should receive a 200 OK response with forecast data

Cenário	***Given***	***When***	***Then***
Get forecast with invalid coordinates	I have invalid coordinates	I send a GET request to /gridpoints/wfo/x,y/forecast	I should receive a 400 Bad Request response
Get observation stations	The weather API is available	I send a GET request to /stations	I should receive a 200 OK response with station list
Get observation for a station	I have a valid station ID	I send a GET request to /stations/stationId/observations/latest	I should receive a 200 OK response with observation data
Get observation with invalid station ID	I have an invalid station ID	I send a GET request to /stations/stationId/observations/latest	I should receive a 404 Not Found response
Get zone forecast	I have a valid zone ID	I send a GET request to /zones/forecast/zoneId/forecast	I should receive a 200 OK response with zone forecast
Get zone list	The weather API is available	I send a GET request to /zones	I should receive a 200 OK response with zone list
Get point information	I have valid latitude and longitude	I send a GET request to /points/point	I should receive a 200 OK response with point information

Nota: A ferramenta exporta os cenários em formato CSV para integração com *pipelines* de CI/CD e *frameworks* BDD. O arquivo completo com os cenários gerados está disponível no repositório do projeto em `/outputs/scenarios/`. O formato permite conversão automatizada para arquivos `.feature` através de *scripts* auxiliares, viabilizando execução direta em *frameworks* como Cucumber, Behave ou SpecFlow.

C Estrutura Completa do Projeto

Este apêndice apresenta a estrutura completa de diretórios do projeto **API Parameter Coverage & Test Scenario Generator**, organizada hierarquicamente com descrições funcionais de cada componente principal. A arquitetura modular adotada segue convenções estabelecidas de projetos Python de código aberto (LUTZ, 2013), facilitando navegação, manutenção e contribuições da comunidade. A organização segue o princípio de separação de responsabilidades, onde cada módulo encapsula funcionalidades coesas e relacionadas, minimizando acoplamento entre componentes e maximizando a coesão interna de cada unidade (MARTIN, 2003).

C.1 Organização de Diretórios de Nível Raiz

A Tabela C.1 apresenta os diretórios e arquivos principais localizados na raiz do repositório, cada um com função específica no ciclo de desenvolvimento e execução da ferramenta.

Tabela C.1: Estrutura de diretórios e arquivos de nível raiz

Caminho	**Descrição Funcional**
`src/`	Código-fonte principal contendo todos os módulos da aplicação
`tests/`	Suíte completa de testes unitários, de integração e BDD
`output/`	Diretório de artefatos gerados durante execuções do *pipeline*
`docs/`	Documentação técnica complementar do projeto
`main.py`	Ponto de entrada principal e orquestrador do *pipeline*
`requirements.txt`	Especificação de dependências Python com versões
`pytest.ini`	Configuração do *framework* pytest para execução de testes
`.env.example`	*Template* de variáveis de ambiente para configuração
`LICENSE`	Licença MIT do projeto de código aberto
`README.md`	Documentação principal com instruções de uso

D Estrutura do Diretório Source

O diretório `src/modules/` contém a implementação de todos os módulos funcionais do sistema, organizados por domínio de responsabilidade conforme princípios de *design* estabelecidos (FOWLER, 2002). A Tabela D.1 apresenta a especificação detalhada de todos os componentes implementados. Os módulos estão agrupados por domínio funcional: `swagger/` para ingestão e validação de especificações, `engine/` para algoritmos centrais e integração com LLMs, `brd/` para gerenciamento de requisitos de negócio, `workflow/` para orquestração do *pipeline*, `utils/` para utilitários compartilhados, e `cli/` para interface de linha de comando. Essa organização modular facilita a manutenção independente de cada componente e permite a evolução incremental do sistema sem impacto em funcionalidades existentes.

Tabela D.1: Especificação dos componentes do sistema

Módulo	**Componente**	**Responsabilidades Funcionais**
`swagger/`	`sch_fetcher`	*Download* de especificações OpenAPI via HTTP/HTTPS com suporte a URLs remotas. Implementa detecção automática de formato através de análise de extensão, *header Content-Type* e estrutura sintática. Inclui tratamento robusto de erros de rede com lógica de *retry* e *backoff* exponencial configurável.
`swagger/`	`sch_validator`	Validação estrutural de especificações contra padrões Swagger 2.0 e OpenAPI 3.x. Realiza detecção automática de versão através de chaves raiz, verificação de campos obrigatórios conforme especificação relevante, e normalização de campos opcionais com valores padrão.

Continua na próxima página

Tabela D.1 – Continuação da página anterior

Módulo	**Componente**	**Responsabilidades Funcionais**
`engine/ algorithms/`	`processor`	Extração sistemática de metadados da API incluindo informações gerais, *endpoints* disponíveis e métodos HTTP suportados. Processa componentes reutilizáveis como *schemas*, *parameters* e *responses*. Implementa resolução recursiva de referências $ref para expansão de estruturas aninhadas.
`engine/ algorithms/`	`analyzer`	Análise profunda de estrutura de *schemas* com extração exaustiva de parâmetros diferenciados por localização (*path*, *query*, *header*, *body*) e tipo de dados. Calcula domínios de iteração baseados em *constraints* e produz métricas de complexidade estrutural.
`engine/ algorithms/`	`csv_generator`	*Parsing* de cenários Gherkin em texto livre retornados por LLM. Identifica blocos sintáticos de *Feature*, *Scenario* e *Steps* individuais. Extrai metadados como *tags* organizacionais. Converte para formato CSV estruturado com colunas padronizadas.
`engine/ analytics/`	`mtc_collector`	Registro centralizado de métricas operacionais coletadas durante execução do *pipeline*. Rastreia tempo de execução total e por etapa, uso de *tokens* LLM separados por *prompt* e *completion*, métricas de complexidade de entrada, e indicadores de qualidade de saída.

Continua na próxima página

Tabela D.1 – Continuação da página anterior

Módulo	**Componente**	**Responsabilidades Funcionais**
`engine/ analytics/`	`alg_tracker`	Rastreamento detalhado de cada algoritmo executado durante processamento. Registra identificação única, *timestamp* de execução, complexidade computacional estimada, transformações de dados realizadas com tamanhos de entrada/saída, e indicadores de qualidade.
`engine/ llm/`	`prompter`	Construção de *prompts* contextualizados e estruturados para geração de cenários. Implementa integração unificada com múltiplos provedores LLM (OpenAI, Groq, Anthropic, Google, Azure). Gerencia estratégia de *chunking* adaptativo para APIs grandes e controle de limites de *tokens*.
`brd/`	`brd_schema`	Definição formal de *schemas* JSON para validação de estrutura de BRDs. Implementa classes de dados tipadas para representação de requisitos individuais e documentos completos. Mantém constantes de configuração e valores padrão do módulo.
`brd/`	`brd_loader`	Operações de entrada e saída para arquivos BRD em formato JSON. Implementa leitura com validação de formato, escrita de BRDs gerados com formatação consistente, e persistência em diretórios configuráveis com nomenclatura padronizada.

Continua na próxima página

Tabela D.1 – Continuação da página anterior

Módulo	**Componente**	**Responsabilidades Funcionais**
`brd/`	`brd_parser`	*Parsing* de documentos de requisitos não estruturados em múltiplos formatos. Suporta extração de texto de PDF via PyPDF2, Word via python-docx, e formatos texto como TXT, CSV e Markdown. Processa texto extraído via LLM para estruturação.
`brd/`	`brd_validator`	Validação de conformidade de BRDs contra *schema* formal definido. Verifica presença de campos obrigatórios, valida formato de identificadores, e gera relatórios detalhados de conformidade com indicação de problemas encontrados.
`brd/`	`brd_generator`	Geração automática de BRD a partir de análise de especificação OpenAPI. Constrói *prompts* especializados para inferência de requisitos funcionais via LLM. Aplica *threshold* de cobertura configurável para controle de granularidade do documento gerado.
`brd/`	`sch_cross_ref`	Análise sistemática de cobertura cruzada entre requisitos do BRD e *endpoints* da API. Implementa múltiplas estratégias de correspondência incluindo correspondência exata e por palavras-chave. Identifica lacunas de cobertura e gera relatórios de rastreabilidade.
`workflow/`	`scn_generator`	Orquestração de alto nível do processo completo de geração de cenários de teste. Coordena execução sequencial entre módulos *processor*, *analyzer*, LLM e *csv_generator*. Gerencia estado do *pipeline* e passagem de dados entre etapas com *logging* detalhado.

Continua na próxima página

Tabela D.1 – Continuação da página anterior

Módulo	**Componente**	**Responsabilidades Funcionais**
`workflow/`	`cvg_handler`	Orquestração do fluxo de análise de cobertura entre BRD e especificação. Coordena carregamento/geração de BRD, execução de referência cruzada, e filtragem de *endpoints* baseada em requisitos funcionais para processamento subsequente.
`utils/`	`llm_provider`	Abstração unificada para integração com múltiplos provedores de LLM. Implementa detecção automática de provedor através de análise de formato de chave API. Configura parâmetros específicos como *temperature* e *max_tokens*. Trata erros específicos de cada provedor.
`utils/`	`out_manager`	Gerenciamento completo de arquivos e diretórios de saída. Cria estrutura de diretórios com marcação temporal para organização cronológica. Implementa nomenclatura padronizada de arquivos e escrita de relatórios de *analytics* em formato estruturado.
`utils/`	`validators`	Coleção de funções de validação genéricas reutilizáveis por múltiplos módulos. Inclui validação de URLs, verificação de formatos de arquivo, validação de *schemas* JSON, e funções auxiliares de verificação de tipos e estruturas.
`cli/`	`cli_utils`	Interface interativa de linha de comando para execução da ferramenta. Implementa menus de seleção de opções, validação de entrada do usuário, formatação de mensagens de progresso, e coleta estruturada de parâmetros de configuração do *pipeline*.

E Cenários para Endpoint de Alertas Meteorológicos

O Código E.1 apresenta cenários gerados para o *endpoint* de alertas meteorológicos, incluindo casos positivos e negativos (MARTIN-LOPEZ; SEGURA; RUIZ-CORTéS, 2019).

```
Feature: Weather Alerts by Zone
  As a weather monitoring application
  I want to retrieve active alerts for specific zones
  So that I can notify users of weather warnings
  @api @alerts @positive @critical
  Scenario: Retrieve active severe weather alerts
    Given the weather API is available
    And I have a valid zone ID "ALZ002"
    When I send a GET request to "/alerts/active/zone/{zoneId}"
    Then the response status code should be 200
    And the response should be in JSON format
    And the response should contain "features" array
    And each alert should have "severity" property
  @api @alerts @negative @validation
  Scenario: Attempt to get alerts with malformed zone ID
    Given the weather API is available
    And I have a malformed zone ID "12345"
    When I send a GET request to "/alerts/active/zone/{zoneId}"
    Then the response status code should be 400 or 404
  @api @alerts @negative @boundary
  Scenario: Attempt to get alerts with empty zone ID
    Given the weather API is available
    And I have an empty zone ID ""
    When I send a GET request to "/alerts/active/zone/{zoneId}"
    Then the response status code should be 400
    And the response should indicate invalid parameter
```

Listing E.1: Cenários gerados para *endpoint* de alertas meteorológicos