UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS I - CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

KAIQUE DA SILVA IVO

TESTE DE API NO CONTEXTO DE EVOLUÇÃO DE MICROSSERVIÇOS: ESTUDO DE CASO NO SISTEMA REGPET

CAMPINA GRANDE
2025

KAIQUE DA SILVA IVO

TESTE DE API NO CONTEXTO DE EVOLUÇÃO DE MICROSSERVIÇOS: ESTUDO DE CASO NO SISTEMA REGPET

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Ciência da Computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de Bacharel em Computação.

Área de concentração: Engenharia de Software

Orientadora: Profa. Dra. Sabrina de Figueirêdo Souto

CAMPINA GRANDE
2025

I96t Ivo, Kaique da Silva.

Teste de API no contexto de evolução de microsserviços [manuscrito] : estudo de caso no sistema RegPet / Kaique da Silva Ivo. - 2025.

65 p. : il. colorido.

Digitado. Trabalho de Conclusão de Curso (Graduação em Computação) - Universidade Estadual da Paraíba, Centro de Ciências e Tecnologia, 2025. "Orientação : Profa. Dra. Sabrina de Figueirêdo Souto, Coordenação do Curso de Computação - CCT. "

1. Engenharia de software. 2. Teste de software. 3. DevOps. 4. Microsserviços. I. Título

21. ed. CDD 005.1

Elaborada por Bruno R. F. de Lima - CRB - 15/1021 BC/UEPB

KAIQUE DA SILVA IVO

TESTE DE API NO CONTEXTO DE EVOLUÇÃO DE MICROSSERVIÇOS: ESTUDO DE CASO NO SISTEMA REGPET

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Ciência da Computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de Bacharel em Computação

Aprovada em: 26/02/2025.

BANCA EXAMINADORA

Documento assinado eletronicamente por:

- **Sabrina de Figueirêdo Souto** (***.047.964-**), em **03/03/2025 22:40:43** com chave **afcc45d8f89911efb36906adb0a3afce**.
- **Ana Isabella Muniz Leite** (***.834.864-**), em **03/03/2025 22:44:36** com chave **3ad3b83cf89a11efba7a1a1c3150b54b**.
- **Adelito Borba Farias** (***.520.684-**), em **04/03/2025 08:50:47** com chave **e994baf2f8ee11efb6061a7cc27eb1f9**.

Documento emitido pelo SUAP. Para comprovar sua autenticidade, faça a leitura do QrCode ao lado ou acesse https://suap.uepb.edu.br/comum/autenticar_documento/ e informe os dados a seguir.
Tipo de Documento: Folha de Aprovação do Projeto Final
Data da Emissão: 04/03/2025
Código de Autenticação: 19e277

A Deus e aos meus pais, por todo amor incondicional, apoio e esforço, DEDICO.

AGRADECIMENTOS

Primeiramente, quero agradecer a Deus por me abençoar com esta oportunidade, nunca ter me deixado e por sempre ter me ajudado nos momentos mais difíceis da minha trajetória.

Também quero evidenciar a dedicação de todo o meu esforço, principalmente aos meus pais, por terem me incentivado e me apoiado de todas as formas a buscar essa direção.

Quero a agradecer a minha mãe Maria Aparecida, meu exemplo de força e garra, pelo enorme esforço que fez para me manter firme e forte, por nunca desistir de mim, pela motivação, pelos ensinamentos, pelo amor e cuidado.

Quero agradecer ao meu pai Milton, por nunca desacreditar de mim, sempre me acompanhar nos momentos de aprendizado, pelos ensinamentos, pelo seu esforço e companheirismo.

Agradeço a todos os meus familiares que me auxiliaram de alguma forma nessa trajetória.

Agradeço também a todos os meus amigos que também contribuíram, me auxiliando com sua parceria e lealdade.

Aproveito ainda para também agradecer a todos os professores que tive na UEPB, pelo conhecimento repassado, e em especial à minha orientadora e professora Sabrina Souto, pelos ensinamentos, apoio, pelas oportunidades e por sua paciência.

RESUMO

A evolução da demanda do mercado de software trouxe consigo a necessidade cada vez maior de construir um sistema de qualidade. Nova técnicas, estratégias e metodologias auxiliam desenvolver sistemas melhores em eficiência e escalabilidade, atributos importantíssimos para a evolução de uma aplicação. Nesse rumo, a arquitetura baseada em microsserviços ganhou destaque por beneficiar o sistema de várias formas, mas trazendo novos problemas por sua complexidade. Os desafios ao utilizar esse tipo de arquitetura impactam também na sua testabilidade, área essa que ainda possuem lacunas de pesquisa com relação a microsserviços, sendo necessária a cogitação de novas abordagens de teste. Especialmente no âmbito de evolução de sistema, em que o software sofre mudanças significativas, o DevOps é uma variável essencial, pois é indispensável nas operações de versionamento, implantação e monitoramento. Nesse sentido, o objetivo desse trabalho é alavancar a qualidade de software em um contexto de evolução contínua, através da eficientização de processos de teste e versionamento, e da redução da complexidade operacional. Para isso, propomos uma metodologia, que foi implementada no sistema RegPet, possibilitando a obtenção de resultados de teste com métricas de qualidade. Os resultados permitiram obter uma visão ampla sobre como os microsserviços do RegPet evoluíram em relação a qualidade do sistema e dos testes.

Palavras-chave: engenharia de software; teste de software; devops; microsserviços.

ABSTRACT

The evolution of the software market demand has brought with it the increasing need to build a quality system. New techniques, strategies and methodologies help develop systems that are more efficient and scalable, which are extremely important attributes for the evolution of an application. In this direction, the microservices-based architecture has gained prominence because it benefits the system in several ways, but it also brings new problems due to its complexity. The challenges of using this type of architecture also impact its testability, an area where there are still research gaps in relation to microservices, requiring the consideration of new testing approaches. Especially in the context of system evolution, where software undergoes significant changes, DevOps is an essential variable, as it is indispensable in versioning, deployment and monitoring operations. In this sense, the objective of this work is to leverage software quality in a context of continuous evolution, through the efficiency of testing and versioning processes, and the reduction of operational complexity. To this end, we propose a methodology, which was implemented in the RegPet system, enabling the obtaining of test results with quality metrics. The results allowed us to obtain a broad view of how RegPet's microservices evolved in relation to the quality of the system and tests.

Keywords: software engineering; software testing; devops; microservices.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

Página

1 INTRODUÇÃO

A maneira como as pessoas criam e desenvolvem os sistemas de software vem sendo cada vez mais inovada, com novas técnicas, metodologias e estratégias, e isso vem sendo influenciado pela crescente demanda no mercado, que necessita de soluções cada vez mais eficientes e escaláveis. Os desafios de desenvolver e manter um software mais complexo também refletem no âmbito de manter sua qualidade, principalmente em softwares que estão sempre evoluindo. As soluções inovadoras trazem diferentes interpretações e conceitos sobre o sistema, a arquitetura baseada em microsserviços, por exemplo, trouxe diversos benefícios, mas também um aumento considerável na complexidade junto a eles.

Por mais que satisfaça o cliente inicialmente, um software de qualidade geralmente não é desenvolvido na sua primeira versão e deixado de lado, pois variáveis como as expectativas dos usuários e as regras de negócio mudam inevitavelmente, assim como também erros no código podem ser descobertos, levando o sistema a executar comportamentos que não batem com os requisitos. Por isso o sucesso de um sistema não pode ser medido por sua versão inicial, mas deve ser levado em conta sua capacidade de evoluir continuamente, respondendo de forma eficaz às mudanças constantes nas demandas do mercado e nas necessidades dos usuários (Svahnberg 2003) (Sommerville 2011)

O conceito de evolução de software está ligado diretamente aos processos contínuos de melhoria, manutenção e extensão de um sistema para que ele continue útil e consiga atender às novas demandas, assim como melhorar em sua qualidade. Para as organizações seus sistemas representam um alto valor comercial, devido à alta dependência a eles e ao alto investimento financeiro feito nos mesmos. Por tanto, claramente elas precisam manter esses softwares úteis, e para isso eles precisam ser reconstruídos de acordo com suas demandas (Sommerville 2011).

Diante desse cenário, é importante destacar que se um sistema não evolue corretamente, sérios prejuízos são atribuídos a seus responsáveis, uma vez que o custo para evolução constitue maior parte do orçamento necessário (Sommerville 2011). Paralelo a isso, observamos o quão importante os testes de software são na garantia de qualidade do produto, dado que são uma das ferramentas fundamentais para a verificação e validação dos requisitos que podem mudar a cada evolução do sistema. Dessa forma, dando um enfoque maior nesse quesito, diminuem-se as chances de o sistema evoluir incorretamente.

Como o mercado atual se depara com a crescente necessidade de entregas mais rápidas, as equipes de desenvolvimento buscam alcançar eficiência no processo de codificar, testar e entregar. Os testes automatizados por sua vez facilitam essa ideia, já que proporcionam uma rapidez e eficiência maior. Esse tipo de teste é benéfico na hora de executar testes que exercitam funcionalidades do sistema, uma vez que independem da ação humana e utilizam o poder computacional da máquina para reproduzir um conjunto de ações

subsequentes e predeterminadas (Bernardo 2011).

Gerenciar a qualidade dos sistemas de microsserviços é um desafio, pois necessitam de maiores cuidados em sua evolução. Devido à alta complexidade, sua maneira de realizar comunicações internas para funcionamento geral difere dos sistemas mais comuns, acarretando desafios impactantes como o rastreamento de mudanças.

Segundo Lercher et al. (Lercher et al. 2024), esse desafio pode proporcionar aos provedores de microsserviços um acoplamento organizacional forte, pois existe uma dificuldade em disseminar informações de mudanças entre todas as equipes, além de que em alguns casos, de modo indireto, a dificuldade em rastrear mudanças faz com que os consumidores relutem para não atualizar sua versão, sendo necessário mais custo para os provedores manterem também a versão anterior.

Nesse contexto evolutivo de sistemas complexos, promover ao software a implementação de técnicas e estratégias de teste com seus demais tipos, assim como coletar métricas relacionadas a qualidade ao longo do seu ciclo de vida útil, se torna favorável para manter o controle sobre a qualidade do seu produto. O acompanhamento da evolução nesse formato permite a equipe desenvolvedora verificar se durante todo o período o produto esteve em conformidade com os requisitos do projeto e seus critérios de aceitação, podendo avaliar seus pontos fracos e fortes para aprender com eles e daí em diante garantir uma maior confiança em seu produto.

O estudo de caso incluso neste trabalho aborda o RegPet, que é um sistema de gerenciamento facilitador para clínicas veterinárias e que permite o registro e acompanhamento de animais e seus tratamentos. Se trata também de um software bastante complexo devido a sua natureza distribuída que utiliza a arquitetura baseada em microsserviços. Ao longo de suas versões, o sistema evoluiu continuamente tanto para incluir novas funcionalidades e melhorias como para correção de falhas, sendo cada evolução acompanhada por uma estratégia abrangente de testes do sistema, que garantiram maior qualidade e confiabilidade das versões entregues.

Quando se projeta um sistema de software monolítico, os desenvolvedores utilizam diversas atividades arquitetônicas para construir uma ideia de sistema que consiga satisfazer os requisitos elicitados. Porém, para que se produza um software de qualidade, durante essas atividades, é necessário considerar o impacto da arquitetura em fatores como os atributos de qualidade, que são essenciais para garantir uma boa evolução do sistema.

Com os sistemas de microsserviços não é diferente, mas demandam maiores desafios devido a sua natureza altamente distribuída, autônoma, independente e autocontida. É notório que sistemas desse tipo são mais complexos que os monolíticos e necessitam de mais atenção para serem mantidos.

A capacidade dos testes automatizados de encontrar indícios de funcionalidades afetadas por mudanças é uma das coisas que os torna importantes. Um ponto crítico na evolução de APIs baseadas em microsserviços é justamente a análise manual de impacto

das mudanças, que se torna um verdadeiro desafio especialmente em ambientes DevOps onde são acomodadas mudanças contínuas.

O estudo conduzido por Lercher et al. (Lercher et al. 2024) realizou entrevistas com diversos profissionais que trabalham com microsserviços, e revelou que sem ferramentas automatizadas os desenvolvedores precisam trabalhosamente revisar códigos, logs e documentações para identificar impactos de mudanças em seus consumidores de API. Além disso, o estudo também observou que os testes unitários podem ser insuficientes para detectar todas as implicações de uma mudança.

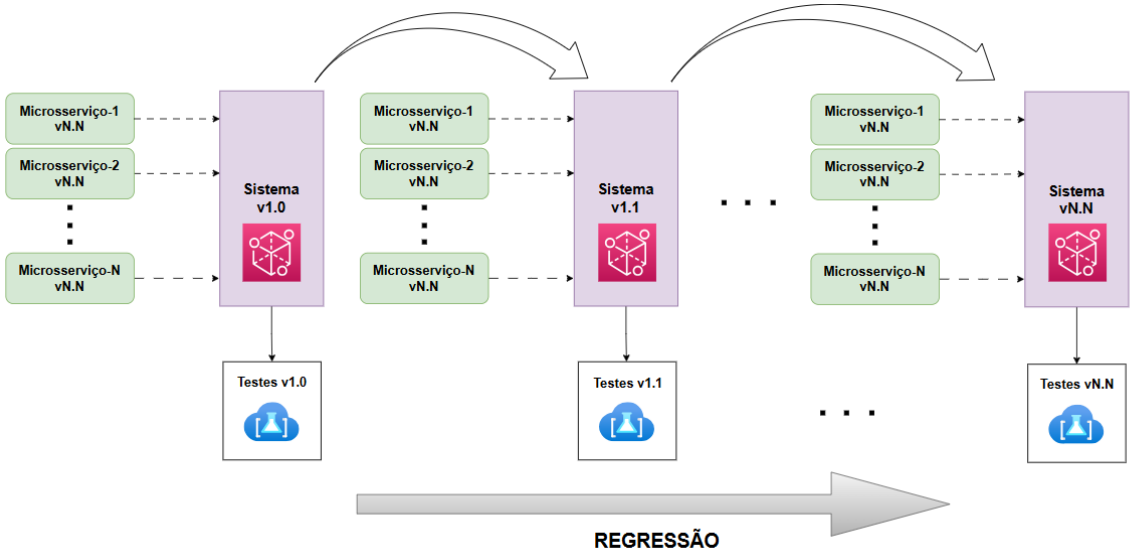
A situação se agrava ainda mais quando erros são cometidos na fase inicial do projeto de um sistema. Segundo um estudo de métodos mistos de Waseen et al. (Waseem et al. 2021), decompôr um sistema de microsserviços, ou seja, definir os limites de funcionalidade de cada microsserviço, é um dos desafios mais frequentemente enfrentados pelos profissionais. Os resultados da pesquisa feita no estudo permitiram inferir que desenvolver microsserviços com limites não tão claros podem proporcionar impactos negativos no monitoramento e nos testes do sistema.

Isso ocorre porque com a má delimitação de funcionalidade a comunicação entre os microsserviços se torna complexa e consequentemente dificulta seu rastreamento, dessa forma aumenta a probabilidade de falhas passarem despercebidas pelos testes.

De acordo com Ghani et al. (Ghani et al. 2019), na garantia de qualidade de microsserviços ainda são enfrentados problemas quanto a sua testabilidade, pois testes de integração e de contrato são fundamentais para garantir a comunicação correta entre os microsserviços já que permitem validar a interação entre módulos dos sistemas, porém ainda são abordagens negligenciadas no âmbito do estudo. Os autores também enfatizam que as pesquisas existentes sobre ferramentas e teste de microsserviços carecem de abordagens com aplicação empírica, destacando a falta de ferramentas específicas para testar esse tipo de software.

Diante deste cenário em que o desenvolvimento de microsserviços necessita de ferramentas de teste para auxílio na identificação de mudanças, é importante enfatizar a complexidade da integração de versões de sistemas desse tipo, uma vez que cada microsserviço evolui em seu próprio ritmo, recebendo versões próprias para integrarem a versão geral do sistema. Além disso, os testes também devem ser versionados, pois são criados para serem adequados para uma versão de sistema específica. Percebemos que essa barreira de muitas variáveis pode se tornar um obstáculo considerável à equipe que desenvolve e testa, impedindo-a de detectar se o sistema regrediu. A figura 1 representa o fluxo comum de evolução de um sistema desse tipo.

Figura 1 – Fluxo de evolução de um sistema de microsserviços



Fonte: Elaborado pelo autor, 2025.

1.1 Objetivo

Visto a problematização e a tendência dos microsserviços a se tornarem cada vez mais difíceis de se manter e testar, esse trabalho é justificado por uma abordagem que automatize a integração entre versões de serviços e a automação de testes em um único fluxo, tendo como objetivo geral alavancar a qualidade de software em um contexto de evolução contínua, através da eficientização de processos de teste e versionamento, e da redução da complexidade operacional.

Com base no objetivo geral proposto, foram estabelecidos os seguintes objetivos específicos:

- Desenvolver abordagem em três etapas: (1) configuração do ambiente dos serviços e dos testes; (2) script para conteinerização dos serviços, versionamento de serviços e testes, e geração de massas de dados para os testes; (3) evoluir script para executar testes versionados e coletar métricas de qualidade.
- Aplicar a abordagem desenvolvida no sistema RegPet.

1.2 Estrutura do trabalho

Este trabalho está dividido em 5 capítulos, sendo que o capítulo 1 traz a introdução contextualizando o tema e detalhando a motivação da realização do trabalho. O capítulo 2 apresenta o referencial teórico, em que são fornecidos conceitos fundamentais embasados e necessários para entendimento do contexto do trabalho. O capítulo 3 fornece explicação e o detalhamento da abordagem utilizada no desenvolvimento da abordagem, incluindo a

arquitetura proposta e os processos de implementação e configuração para funcionamento. O capítulo 4 consiste na aplicação da abordagem desenvolvida de cobertura de código e versionamento na API do RegPet. E por fim, O capítulo 5 apresenta conclusões finais do trabalho, em que é discorrido sobre o contexto da problematização e da aplicação da abordagem no RegPet, refletindo sobre os resultados obtidos e sobre as contribuições para a engenharia de software.

2 FUNDAMENTAÇÃO TEÓRICA

Nesse capítulo é fornecida uma base teórica, onde são apresentados conceitos importantes para a compreensão do contexto discutido nesse trabalho. No início são apresentados conceitos e ferramentas sobre qualidade de software, testes de software e cobertura de código, e mais adiante também conceitos e ferramentas relacionados ao DevOps.

2.1 Qualidade de Software

De acordo com Pressman e Maxim (Pressman e Maxim 2021), a qualidade de software significa a aplicação de uma gestão efetiva que consiga fornecer suporte a processos delicados no desenvolvimento do software como gerenciamento de mudanças e revisões técnicas, a fim de entregar valor tanto para o lado dos desenvolvedores quanto para o lado dos usuários, além de fazer o produto atender aos requisitos que se pede de maneira confiável e sem erros. No âmbito dos desenvolvedores, o valor se refere a gerar benefícios ao fabricante ao ter um software que exige menos manutenção e correção ao longo da evolução. Já para usuários a agregação de valor está na agilização de algum processo objetivado por meio do software.

Sommerville (Sommerville 2011) destaca que a qualidade nos softwares difere e é mais difícil de ser avaliada do que a qualidade em produtos manufaturados, onde pode haver tolerâncias e especificações claras. No contexto de um software a complexidade se dá principalmente pela dificuldade em escrever especificações completas, pelas diferentes interpretações dos interessados no produto e pela dificuldade de medir alguns atributos de qualidade como a manutenibilidade, por exemplo.

2.2 Testes de Software

Segundo Neto (Neto 2007), o teste de software é um processo de natureza "destrutiva", pois objetiva executar um produto em busca de descobrir defeitos e suas causas. Utilizando esse tipo de conduta é possível detectar se esse produto atende ao que foi especificado para fazer, dessa forma se discerne se ele é confiável ou não.

Complementando essa ideia, Sommerville (Sommerville 2011) explica que esses processos fazem parte de conceitos mais amplos da qualidade de software chamados de verificação e validação (V&V), podendo servir de demonstração aos envolvidos no projeto que o software atinge suas especificações e requisitos definidos. Além disso, também possibilitam diferentes implementações de estratégias conforme o tipo de software, facilitando encontrar os demais erros e corrigi-los para a entrega final.

Essencialmente, os testes se dividem em dois tipos que avaliam diferentes aspectos do sistema: os testes funcionais e os não funcionais.

Os testes funcionais se concentram em avaliar se as funções do aplicativo se comportam o que é esperado e sob diferentes condições. A utilização desse tipo de teste permite

a descoberta de falhas funcionais antes da implantação do aplicativo. Eles podem ser categorizados em (Objective Solutions 2024):

- Testes de unidade: consistem em verifica partes isoladas do código para garantir que as menores unidades que compõe o sistema funcionem conforme esperado.
- Testes de integração: consistem em garantir que a interação entre diferentes componentes do sistema execute o que é esperado, visando verificar como as partes individuais se comportam em conjunto.
- Testes de sistema: avaliam o sistema como um todo, verificando se as funcionalidades encontradas trabalham devidamente em conjunto.
- Testes de aceitação: avaliam se o produto de software atende as necessidades do usuário.

Já os testes não funcionais não avaliam "o que"o sistema faz, são utilizados para avaliar "como"o sistema realiza suas funções, buscando observar a desempenho do mesmo. Eles se subdividem em:

- Testes de performance: testam como o software se comporta ao ser exposto a uma carga, avaliando sua capacidade e estabilidade de resposta.
- Testes de segurança: exploram as vulnerabilidades do sistema em situações de ataques cibernéticos.
- Testes de usabilidade: consistem em garantir que a interface do software seja agradável e intuitiva para seus usuários.
- Testes de confiabilidade: avaliam se o sistema consegue continuar funcionando sem falhas em diversas situações normalmente inesperadas pelos usuários.

2.2.1 Automação de Testes

Bernardo (Bernardo 2011) descreve automação de testes como o processo de desenvolvimento de testes automatizados, que por sua vez são roteiros de testes de software que executam funcionalidades do sistema e verificam automaticamente o cumprimento dos critérios. Assim como os testes manuais, os teste automatizados também visam melhorar a qualidade dos sistemas por meio da verificação e validação, porém expandem a área de estudo alterando paradigmas relacionados a implementação, manutenção e execução. Ou seja, é uma forma de tornar testes de software independentes da ação humana.

Já que esse tipo de teste é executado por um computador, é possível aproveitar de todo o auxílio que a máquina oferece. Nesse contexto, a maior velocidade de execução,

melhor delimitação de etapas, maior facilidade para reproduzir testes complexos, e possibilidade de execução paralela são benefícios adquiridos ao realizar testes automatizados (Bernardo 2011).

Além desses benefícios, Graham e Fewster (Graham e Fewster 1999) também enfatizam que a automação de testes comparada aos testes manuais também reduzem os esforços necessários para a execução de testes de regressão (testes já definidos executados em uma nova versão do sistema com intuito de verificar a estabilidade na evolução do produto), liberando testadores para tarefas mais estratégicas, além de reduzirem o tempo de lançamento do produto ao mercado devido a rápida execução dos testes.

De maneira resumida, testes mais abrangentes podem ser realizados com menos trabalho, resultando em melhorias na eficiência da equipe.

2.2.2 Teste de API

A sigla API significa Interface de Programação de Aplicações e se refere a um programa com uma coleção de ferramentas, definições e protocolos que servem para fornecer uma interface de comunicação entre duas aplicações (Red Hat 2024; Amazon Web Services 2024).

A arquitetura das APIs geralmente se baseiam em cliente e servidor, onde o servidor provisiona pontas do canal de comunicação (Endpoints), e o cliente os acessa por meio de um endereço URL (Amazon Web Services 2024; Grespi 2020). Essa integração funciona mediante solicitações e respostas e é necessária quando a aplicação cliente necessita de serviços que podem ser executados pela aplicação do servidor.

O teste de API se caracteriza como um teste funcional de integração, no qual em termos práticos a ideia principal é validar o trecho de código associado a URL disponibilizada pela API, ou seja, tratando a API como uma caixa preta, focando no que entra e no que sai (solicitações e respostas respectivamente), e verificando consequentemente o comportamento das respostas em diversas situações. (Bangare et al. 2012; Objective Solutions 2024).

Complementando, uma API é projetada para realizar tarefas específicas da sua funcionalidade, isso significa que quando são chamadas pelo cliente, os resultados, erros e exceções das suas respostas podem variar conforme as entradas fornecidas. Porém, quando integradas a outra aplicação, os testes unitários geralmente cobrem apenas caminhos limitados da API, podendo existir caminhos não exercitados que acabam resultando em comportamentos inesperados. Então, desenvolver estratégias mais elaboradas para testar APIs pela URL é essencial garantir a qualidade adequada de um produto (Bangare et al. 2012).

2.2.3 Ferramentas de Teste Automatizado

Ferramentas de teste automatizado são programas ou bibliotecas que possibilitam a execução de testes de software e seus demais tipos de forma automática. A característica

mais importante delas está na potencialização do processo de verificação e validação. Além disso, fornecendo uma gama de recursos, auxiliam a validação de comportamentos funcionais e não funcionais do sistema também de maneira automática. Consequentemente eliminam a necessidade de intervenção humana no processo de testamento, o que atrasaria o processo ágil de desenvolvimento de qualidade de um software (Digital.ai 2024).

No geral, existem diversos tipos de ferramentas de teste automatizado, cada qual direcionada para um tipo de teste diferente que validam diversos aspectos no software. São elas (Digital.ai 2024):

- Ferramentas de teste de unidade: ferramentas direcionadas para testes unitários, onde pretendem verificar funcionalidades pequenas e mais isoladas do código, auxiliando na identificação de erros logo no início do desenvolvimento.

- Ferramentas de teste funcional: sendo a categoria mais importante neste trabalho, elas são voltadas para o desenvolvimento de testes funcionais. Possibilitam simular as interações do usuário por meio de processos escritos em código, auxiliando na verificação da conformidade das funções do sistema com seus requisitos e consequentemente detectando erros antes de chegar ao usuário. Elas ainda dispõe de recursos para verificação automática de resultados e geração de relatórios dos mesmos.

- Ferramentas de teste de desempenho: servem para simular testes de desempenho, possibilitando medir o desempenho sob carga, e identificando gargalos e problemas de velocidade de resposta.

- Ferramentas de teste de acessibilidade: automatizam testes de frontend que verificam conformidades com os padrões de acessibilidade, ajudando a identificar barreiras para usuários com deficiência.

- Ferramentas de teste de segurança: auxiliam a automatização de testes que focam em checar pontos fracos de segurança, identificar vulnerabilidades e evitar riscos de ataques cibernéticos.

A capacidade dessas ferramentas de encontrar inconsistências em um sistema de fato é muito maior que a humana, isso é o que as torna de suma importância para o controle de qualidade. Além disso, como falado anteriormente a otimização do trabalho da equipe testadora e a melhora na gestão dos recursos no decorrer do projeto são outras vantagens indispensáveis para o desenvolvimento de software no mercado moderno, as quais são possibilitadas fazendo uso dessas ferramentas (Monitora Tec 2021).

2.2.3.1 Cypress

Cypress (Cypress.io 2024) é um framework desenvolvido para criar testes funcionais focados em frontend. Ele consegue testar tudo que rode em um navegador, fornecendo

suporte a diversas opções de recursos como testes de ponta a ponta (que executa todo um caminho pela aplicação), testes de componentes específicos, testes de acessibilidade e cobertura de interface. Por tanto é percebido que essa é uma ferramenta robusta que contribui amplamente para a qualidade de software.

Embora seja focado em testes de interface gráfica, o Cypress ainda disponibiliza plugins para adaptação de outros tipos de recurso, como por exemplo, os que foram usados no RegPet, nos permitindo direcionar o Cypress para realizar testes de backend (testes de API) e coletar métricas de cobertura de código.

2.2.3.2 Cucumber

O Cucumber (Coodesh 2025) é uma ferramenta muito importante para um software de qualidade, pois facilita a verificação na hora de acompanhar o atendimento aos requisitos de projeto, ele dá suporte a estratégia de Desenvolvimento Orientado ao Comportamento (BDD). Uma característica diferenciada do Cucumber é utilizar uma linguagem simples para validar se o sistema faz o que deve fazer, dessa forma ele também aproxima a comunicação entre desenvolvedores e pessoas que não possuem conhecimento técnico (Cucumber Ltd. 2024).

A forma de desenvolver software BDD (Cucumber Ltd. 2024) tem a função principal de diminuir a distância entre os que desenvolvem e os que pedem o sistema. Ao produzir uma documentação que fornece feedback do comportamento do sistema e do cumprimento de requisitos, essa estratégia facilita a colaboração e mitiga a lacuna existente na criação de um produto de software, que é justamente fazer o outro compreender a ideia que você está raciocinando.

O BDD é utilizado no Cucumber pela criação de cenários de teste escritos em Gherkin, que é uma linguagem que utiliza as palavras-chave "Given", "When"e "Then"para descrever o comportamento esperado do sistema (Cucumber Ltd. 2024). Na prática utiliza-se essas palavras para definir os passos que o teste deve realizar, e utilizando esse recurso aproxima-se a linguagem de implementação da linguagem de negócio.

No caso do RegPet, empregamos complementarmente o Cucumber ao Cypress. Onde o Cucumber é responsável por comunicar o que está sendo testado e o Cypress por dar "vida"aos testes. A integração dessas duas ferramentas permite uma abordagem mais descritiva e compreensível, conseguindo promover transparência e colaboração entre as equipes durante o processo do desenvolvimento de qualidade do RegPet (Pathak 2024).

2.2.4 Cobertura de Código

Na Ciência da Computação, a cobertura de código é uma métrica quantitativa obtida pelos testadores para medir estrategicamente a quantidade de código de um software exercitada com a execução de testes. A cobertura de código pode ser analisada em termos

de linhas, funções, ramos, e declarações. São diferentes métricas que podem orientar a equipe para analisar o alcance dos seus testes criados (Vieira 2020; JetBrains 2024; Ivanković et al. 2019; Schwering e Yeen 2023).

Isso significa que a ferramenta é importante em termos de qualidade de teste, pois ajuda a encontrar partes do código que não foram testadas. Portanto, ela auxilia a encontrar código mal feito ou inútil. Quanto maior a cobertura de código de um sistema, menor a probabilidade de um erro passar despercebido. Entretanto, a cobertura de código não implica necessariamente alta qualidade do sistema, pois não define a eficácia dos testes. Não há garantia de que um teste validará o que de fato tem de validar. Mesmo assim, sistemas com alta cobertura tendem a ser mais confiáveis do que aqueles com baixa cobertura (Vieira 2020; JetBrains 2024).

De acordo com Schwering e Yeen (Schwering e Yeen 2023), quatro dos tipos mais comuns de cobertura de código são:

- Cobertura de funções: analisa a porcentagem de funções executadas no código que foram chamadas pelos testes.
- Cobertura de linhas: captura a porcentagem de linhas executadas no código. Ou seja, se uma linha não foi executada significa que há partes do código que não foram alcançadas pelos testes.
- Cobertura de ramos: mede a porcentagem de ramificações executadas no código, como instruções condicionais de decisão. Ela determina a quantidade exercitada de caminhos entre todos os caminhos que o código possui.
- Cobertura de declarações: mede a porcentagem de declarações em linhas que o código executa com o teste. Levando em consideração que cada linha pode ter várias declarações, esse tipo de cobertura difere da cobertura de linha que considera apenas se a linha foi executada ou não, nesta são consideradas se as declarações foram executadas ou não.

2.2.5 Ferramentas de Cobertura de Código

Uma ferramenta de cobertura de código é um software que facilita a prática da cobertura de código pela equipe desenvolvedora, realizando a análise e medição da porcentagem de código executado durante a execução dos testes automatizados e apresentando as informações obtidas, como os demais tipos de cobertura, via relatório (Maas 2022). Elas se baseiam na instrumentação do código, que é uma técnica que permite que programas modifiquem outros programas ou até ele mesmo, antes ou durante sua execução (Cabral 2005). Utilizando esse mecanismo as ferramentas conseguem monitorar a execução do código a partir de um código adicional, conseguindo realizar sua análise.

No estudo de caso do trabalho foi utilizada a biblioteca Nyc (Istanbul 2024), que dispõe de ferramentas de instrumentação e geração de relatórios de cobertura de código para aplicações Node.js, o qual é o caso da aplicação do servidor do RegPet (Walls 2020). Node.js (Node.js) é um ambiente de execução para código na linguagem JavaScript e Typescript que possibilita a criação de diversos tipos de aplicações, incluindo servidores web, fornecendo uma maior facilidade para gerenciar pacotes de biblioteca como o Nyc.

Escolhemos o Nyc por sua fácil integração com Node.js, e por oferecer relatórios detalhados de métricas como cobertura de declarações, cobertura de ramos, cobertura de linhas e cobertura de funções (Istanbul 2024). Outro motivo para a escolha desta ferramenta foi a utilização do plugin "cypress/code-coverage"(Cypress.io 2024) para cobertura de código responsável por integrar as duas ferramentas (Cypress e Nyc) para instrumentação do código e coleta de informações .

2.3 DevOps

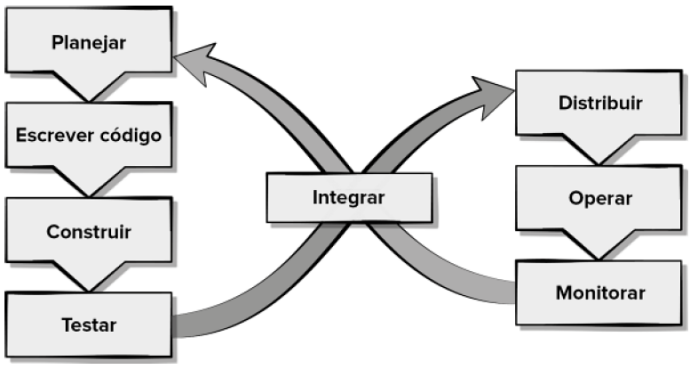
No mercado da Engenharia de Software existem debates que discutem DevOps como sendo um cargo ou não, mas em essência esse termo se refere as atividades resultantes da fusão dos processos de desenvolvimento e de operações, a necessidade de juntar essas duas áreas surgiu no âmbito de melhorar tarefas que oferecem suporte para construir, manter e escalar um software, elas incluem diagnóstico de problemas, gerenciamento de implantação, automação de tarefas e padronização de processos (Roche 2013).

2.3.1 Cultura e Práticas DevOps

Adotar práticas de DevOps no ciclo de vida de um software é de fundamental importância quando se trata de realizar entregas mais rápidas e qualificadas possíveis de suas atualizações. Essas práticas ajudam a otimizar a transição do produto de seu desenvolvimento para a produção, que em outras palavras seria entregar ao cliente, de maneira que tarefas manuais repetitivas que normalmente requerem mais tempo sejam eficientizadas. Por isso soluções que visam automatizar processos são sempre bem-vindas (Mohammad 2018).

De Acordo com Pressman e Maxim(Pressman e Maxim 2021), é definido um fluxo a ser seguido pelas equipes de DevOps que objetivam melhorar a eficiência, reduzir falhas e acelerar entregas. Tal fluxo é representado na figura 2

Figura 2 – Fluxo do trabalho do DevOps



Fonte: Pressman e Maxim(Pressman e Maxim 2021)

Ainda segundo Pressman e Maxim (Pressman e Maxim 2021), as principais práticas de DevOps são as seguintes:

- Desenvolvimento contínuo: de maneira ágil e integrada a equipe de desenvolvimento realiza entregas incrementais do produto de software para a equipe da garantia de qualidade testar.
- Teste contínuo: a utilização de ferramentas de teste automatizadas permitem uma atuação mais unificada de testar múltiplas partes do código em tempo real, verificando defeitos eficientemente antes da integração ser realizada.
- Integração contínua: os códigos produzidos pelos membros da equipe de desenvolvimento, com novos recursos, são frequentemente integrados ao código já existente e, em seguida, examinados automaticamente antes de serem integrados no ambiente de produção.
- Entrega contínua: esta é a etapa em que o ambiente de produção é preparado para receber o novo código já integrado.
- Monitoramento contínuo: se refere ao constante acompanhamento de membros das equipes de operações e desenvolvimento ao ambiente de produção, geralmente por meio de ferramentas automatizadas, possibilitando a identificação de problemas antes de sua exposição ainda nessa fase.

2.3.2 Monitoramento e Feedback Rápido

O monitoramento é uma abordagem que permite o acompanhamento contínuo de um software implantado no ambiente de produção, cujos objetivos são ajudar atingir uma alta disponibilidade do produto, minimizando métricas que envolvem o tempo de detecção,

mitigação e correção de problemas, e permitir o aprendizado da equipe sobre fatores que fortalecem ou enfraquecem o propósito do software a medida que ele evolue. Dessa forma uma equipe que utiliza dessa estratégia estabelece um feedback mais dinâmico e informativo sobre a utilização do software que eles desenvolvem (Microsoft 2024).

Segundo Soni (Soni 2015), a filosofia do DevOps se baseia no princípio de que quanto mais rápida for a identificação de falhas, mais rápida será a resposta a elas. Dessa forma é possível acelerar o processo de melhoria e evitar prejuízos, estabilizando o software mais rapidamente. Ademais, pode-se dizer que o monitoramento e feedback rápido é de suma importância nesse processo, já que colaboram para reduzir a lacuna existente entre as equipes, principalmente na comunicação de falhas.

2.3.3 Docker e Docker Compose

Docker (Red Hat 2024) é um software de conteinerização de aplicações que possibilita utilizar contêineres Linux como se fossem máquinas virtuais. A conteinerização é uma abordagem muito eficiente para segmentar soluções que executem de maneira independente, melhorando o uso da infraestrutura e oferecendo maior flexibilidade para serem gerenciadas e transferidas entre ambientes. Além disso, o Docker também facilita a implantação de aplicações uma vez que automatiza o processo incluindo todas as dependências do conjunto de serviço constituintes.

O Docker Compose (Docker Inc. 2024) por sua vez faz parte do ecossistema Docker e é a ferramenta responsável por orquestrar cada parte das aplicações multi-contêineres. Ao utilizar o Docker Compose a equipe consegue simplificar o gerenciamento de serviços, redes e volumes de toda sua aplicação. Com isso a replicação do ambiente do aplicativo é facilitada ao se tornar mais portátil.

2.3.4 Arquitetura de Microsserviços

O estilo arquitetônico de microsserviços é uma abordagem para o desenvolver aplicações que executam no lado do servidor seguindo um padrão que a organiza como um conjunto de pequenos serviços, cada um executando de forma independente em seu próprio processo e com sua própria gestão de memória e execução (Lewis e Fowler 2014). Portanto, ao seguir esse modelo de arquitetura, as equipes do desenvolvimento dispõem de uma facilitação para superar diversos desafios significativos ao realizar entregas de novas versões do produto.

Valente (Valente 2020) destaca os seguintes benefícios obtidos ao utilizar microsserviços:

- Descentralização dos Times: como cada serviço é independente e pequeno o suficiente para ser gerenciado por uma equipe ágil, é possível também que as equipes se dividam para trabalhar com serviços específicos, fazendo com que cada uma trabalhe com menos dependências de outra.

- Redução de Riscos de Efeito Colateral: devido à decomposição do sistema em módulos, e levando em consideração que eles são executados em processos distintos e sem compartilhamento de memória, as chances de um módulo causar problemas em outro são drasticamente reduzidas.

- Escalabilidade Específica: microsserviços também permitem que a equipe consiga escalar cada serviço separadamente, então quando algum serviço estiver enfrentando problemas de desempenho por exemplo os desenvolvedores podem escolher apenas ele em específico para ser replicado, o que otimiza o uso de recursos e o custo de infraestrutura.

- Flexibilidade Tecnológica: cada serviço pode ser desenvolvido utilizando a linguagem de programação, o banco de dados e frameworks específicos que facilite a implementação da sua funcionalidade. Dessa forma a equipe responsável pode optar pelas melhores ferramentas sem restrições impostas pelo restante da aplicação.

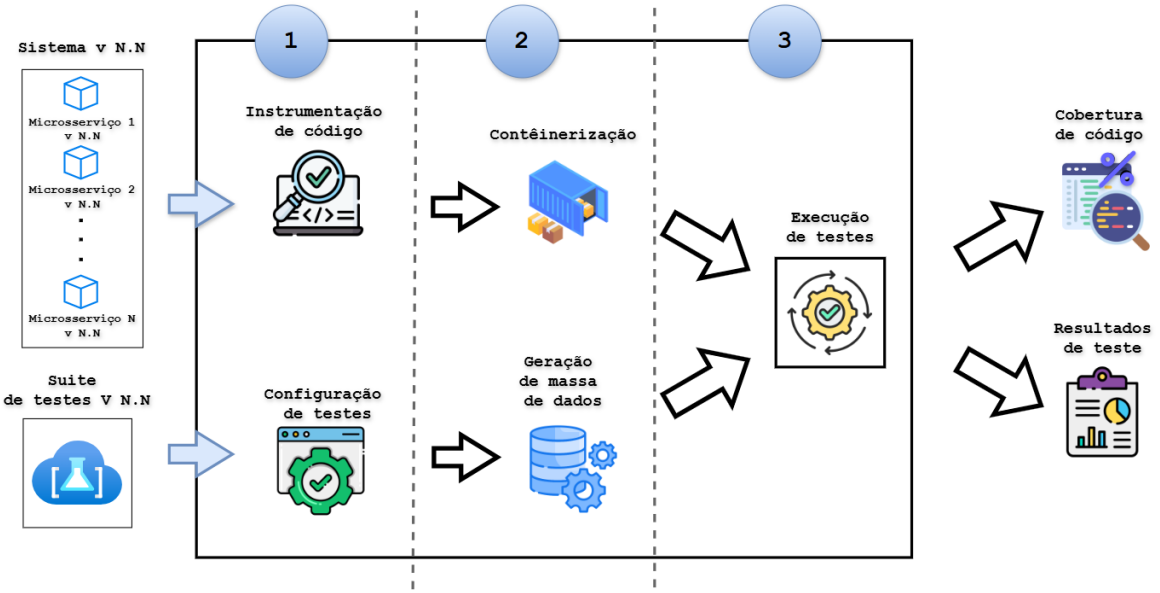
- Resiliência a Falhas: diferentemente de sistemas monolíticos que quando sofrem uma falha em determinada parte geralmente o sistema inteiro enfrenta problemas, na arquitetura de microsserviços as falhas são parciais, aumentando as chances de quando uma parte do sistema falhar as outras funcionalidades ainda estejam de intactas.

3 ABORDAGEM

Este capítulo visa apresentar a abordagem proposta para automatizar a integração entre versões de serviços e a automação de testes em um único fluxo. Para isso, são necessárias as seguintes fases: desenvolver um passo-a-passo para configuração do ambiente dos serviços e dos testes, automatizar a conteinerização dos serviços, versionamento de serviços e testes, e geração de massas de dados para os testes, evoluir para executar testes versionados e coletar métricas de qualidade. Por fim, utilizar a abordagem desenvolvida através de sua aplicação no sistema RegPet.

A figura 3 ilustra uma visão geral da abordagem proposta, onde a abordagem realiza um processo que se divide em 3 fases principais para realização do objetivo pretendido: (1) configuração dos ambientes para obtenção da cobertura de código, (2) empacotamento do sistema e geração de massa de dados para teste e (3) execução de testes e organização de resultados.

Figura 3 – Visão geral da abordagem

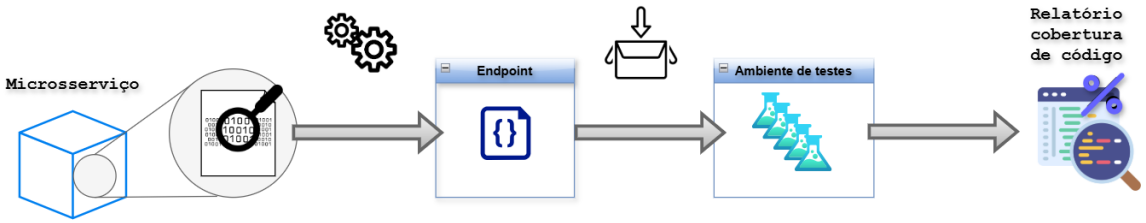


Fonte: Elaborado pelo autor, 2025.

3.1 Configuração de Ambientes

A fase inicial consiste em realizar a preparação dos ambientes de teste e de API para serem capazes de extrair as métricas de cobertura do código dos microsserviços exercitados pelos testes. A figura 4 ilustra esse processo de configuração, que se divide em duas etapas: (i) configurar os microsserviços para disponibilizar os dados e (ii) configurar o framework de testes para acessar e coletar os dados para a construção dos relatórios.

Figura 4 – Configuração de ambientes



Fonte: Elaborado pelo autor, 2025.

3.1.1 Configuração e Instrumentação dos Microsserviços

Na etapa de instrumentação dos microsserviços, deve-se realizar a adaptação do código-fonte para permitir a coleta de métricas de cobertura. Este processo envolve a instalação das dependências necessárias para análise de cobertura, modificações na maneira como o serviço inicializa e configuração de endpoints. Devido à natureza distribuída dos microsserviços, cada um deve ser preparado individualmente, para manter assim suas características de independência.

Os microsserviços podem ser configurados para seu código ser monitorado por meio de instrumentação. Com esse método é possível obter os dados sobre as linhas do código que serão executadas pelo teste em andamento. Além disso, como não é possível enviar esses dados diretamente para o ambiente que executa os testes, ainda mais em ambientes conteinerizados, os serviços devem implementar um endpoint extra, no qual serão expostos todos os dados em um formato parametrizado. Com essa estratégia, o ambiente de testes poderá requisitar o endpoint para obter os dados de cobertura e construir os devidos relatórios.

3.1.2 Configuração do Ambiente de Testes

Nesta etapa, também são realizadas as instalações e configurações das dependências necessárias no ambiente de testes. O framework de testes utilizado deve ser preparado para acessar o endpoint de dados de cobertura, e instrumentado para que após a execução de cada teste sejam coletados os respectivos dados do serviço que está sendo testado e armazenados localmente, atualizando a construção do relatório no próprio ambiente.

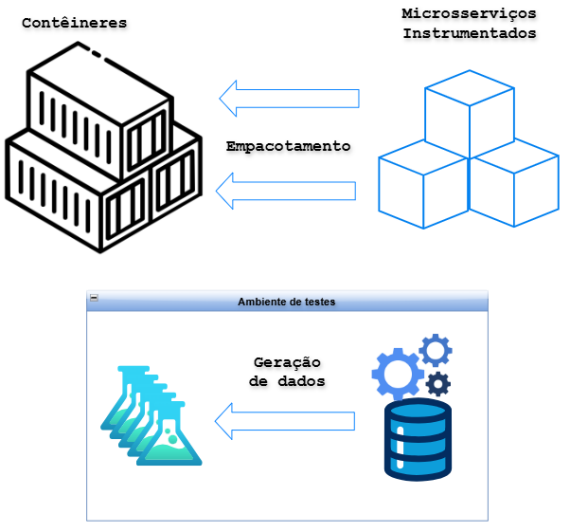
3.2 Conteinerização e Geração de Massas de dados

A fase de conteinerização e preparação de dados diz respeito a necessidade de empacotar e inicializar a versão específica do microsserviço, e preparar a versão adequada do ambiente de testes para a execução. Essa etapa é essencial, pois garante a construção

da versão do microsserviço que é necessária para os testes e a disponibilidade de dados apropriados para validação de suas funcionalidades.

A figura 5 representa essa fase, onde as versões da API e dos microsserviços serão atualizadas (com auxílio de uma ferramenta de versionamento). Tendo a versão pronta, uma ferramenta de conteinerização levantará a imagem da aplicação e os microsserviços já instrumentados serão empacotados. Nesse momento, a geração de massa de dados deve ser inicializada, deixando dados prontos para serem consumidos pelas suítes de testes.

Figura 5 – Conteinerização e Geração de Massas de dados

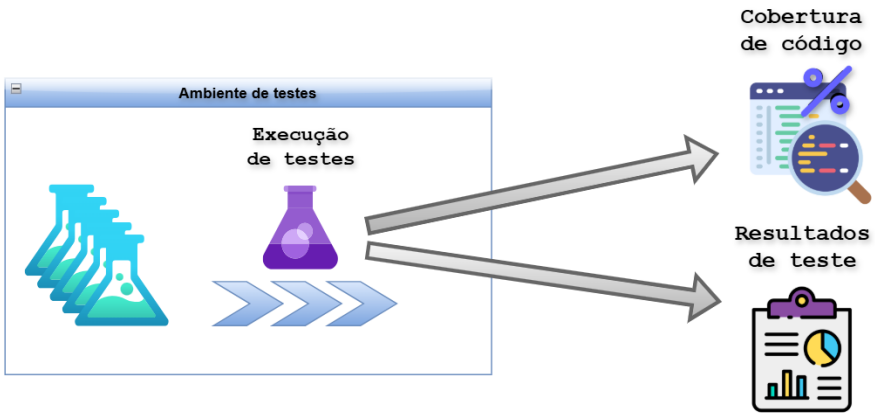


Fonte: Elaborado pelo autor, 2025.

3.3 Execução de Testes e Geração de Relatórios

A execução de testes e geração de relatórios constituem a fase final da abordagem, a figura 6 ilustra essa fase. Considerando que os testes automatizados são executados e com as etapas anteriores tendo sido realizadas corretamente, o fluxo obtido trabalhará coletando dados de cobertura de cada microsserviço. Nessa execução os resultados dos testes e os dados de cobertura devem ser processados para gerar relatórios que fornecem a visão sobre a qualidade e abrangência dos testes.

Figura 6 – Execução de Testes e Geração de Relatórios



Fonte: Elaborado pelo autor, 2025.

Para isso, os serviços já devem estar sendo executados dentro do contêiner com o código já sendo monitorado pela instrumentação. Dessa forma, assim que os testes forem inicializados, as requisições feitas executarão as funcionalidades dos serviços e com isso a ferramenta de instrumentação saberá exatamente quais partes do código foram cobertas durante a execução dos testes, fornecendo os dados sobre a execução.

Assim que terminar a execução de uma suíte, existindo a viabilidade de obter esses dados, as ferramentas de teste, realizam a interação com a interface do endpoint e aguardam a atualização dos dados respectivos a interação da suíte. Após a atualização ser feita, o framework de testes atualiza os relatórios em seu ambiente.

4 ESTUDO DE CASO: COBERTURA DE CÓDIGO EM TESTES DE API NA EVOLUÇÃO DO REGPET

Este capítulo apresenta um estudo de caso realizado no projeto RegPet, que serviu como base para a aplicação da abordagem proposta no capítulo anterior. Inicialmente, o sistema RegPet será apresentado em termos de funcionalidades e arquitetura. Em seguida, será exposto como os testes de API são feitos no RegPet, incluindo o processo de testes. Por fim, será detalhado como a abordagem proposta foi aplicada ao RegPet.

4.1 RegPet

O RegPet é uma aplicação web que atua como uma ferramenta de regulação de procedimentos voltados para a causa animal, desenvolvido pelo NUTES (Núcleo de Tecnologias Estratégicas em Saúde) e utilizado pelo governo do estado da Paraíba.

A Causa Animal se refere ao conjunto de ações e iniciativas voltadas para a proteção, cuidado, bem-estar e defesa dos direitos dos animais. O movimento busca garantir que os animais domésticos sejam tratados de forma ética, responsável e com dignidade, de forma a minimizar o sofrimento causado por abandonos, maus-tratos ou negligência.

O objetivo do sistema RegPet consiste em concentrar atividades relacionadas a realização de procedimentos cirúrgicos com animais no estado da Paraíba. O sistema é baseado nos seguintes módulos descritos na figura 7:

Figura 7 – Módulos do RegPet

Fonte: Secretaria de Estado da Saúde (Paraíba 2023)

Descrição dos módulos:

- Animais: Registro detalhado dos animais tutelados, incluindo informações como

espécie, raça, idade.

- ONG e Protetores Independentes: Cadastro dos responsáveis legais pelos animais.
- Municípios: Gestão dos dados dos setores de Zoonose e Causa Animal.
- Causa Animal: Registro de ações de monitoramento do bem-estar animal.
- Administrativo (Não aparece na figura): Responsável pela geração de relatórios sobre todo o estado da população animal do Estado da Paraíba.

O funcionamento dessa ferramenta facilitadora foca na regulação, reserva e execução de procedimentos medicinais para os animais, onde protetores (individuais, ONGs ou projetos) cadastrados solicitam serviços veterinários (principalmente castrações), que são validados e gerenciados por veterinários da gerência da causa animal. A aplicação ainda permite o acompanhamento de tais procedimentos em clínicas e castra-móveis (Unidades Móveis de Castração Animal), gerenciamento de rotas de atendimento e geração de estatísticas para facilitando o controle e a transparência das ações para o bem-estar animal.

4.1.1 Funcionalidades do sistema

Dentre as funcionalidades principais que permitem a realização do seu objetivo, o RegPet possui:

- Login e Controle de Acesso: O sistema oferece a autenticação de usuários que os redireciona para uma visualização personalizada de acordo com seu perfil, garantindo que cada tipo de usuário tenha acesso apenas às funcionalidades cabíveis a ele. Os tipos de perfis são administrador, protetor, regulador, executante e gestor, e ainda subtipos deles, dependendo da região e escopo da atuação.
- Criação de Solicitações: Os protetores podem submeter solicitações para procedimentos veterinários, preenchendo formulários com as informações detalhadas sobre o animal. Os reguladores e gestores municipais também podem criar solicitações, especialmente para animais errantes.
- Validação de Solicitações: Os usuários reguladores (municipal ou estadual) analisam as solicitações de procedimentos, verificando documentos, validando informações e decidindo sobre a elegibilidade do procedimento, podendo validar ou invalidar cada solicitação.
- Reserva de Vagas: O sistema permite que os reguladores reservem vagas disponíveis em clínicas locais ou em castra-móveis, de acordo com seu escopo de atuação (municipal ou estadual).

- Gerenciamento de Usuários: O usuário administrador pode criar novos usuários, editar informações cadastrais, alterar permissões e excluir usuários, controlando o acesso e a base de usuários do sistema.
- Cadastro de Locais e Vagas: Os usuários executantes podem cadastrar locais de execução de procedimentos (clínicas) e castra-móveis, e as vagas disponíveis, seguindo as definições estabelecidas em um edital governamental.
- Finalização de Solicitações: Após a realização do procedimento, os executantes podem finalizar a solicitação, enviando comprovações, laudos e documentos que registram a conclusão do atendimento veterinário.
- Geração de Relatórios: Alguns tipos usuários podem gerar relatórios personalizados sobre os procedimentos, permitindo análises gerenciais, estatísticas de atendimento e acompanhamento das ações.
- Filtros Personalizados: O sistema oferece ferramentas de filtragem nas listagens de solicitações, entidades de execução e usuários.
- Suporte a Animais Errantes: Uma funcionalidade especial permite o registro e acompanhamento de solicitações para animais sem tutor, facilitando o controle populacional e os cuidados veterinários para esses animais.

4.1.2 Arquitetura do RegPet

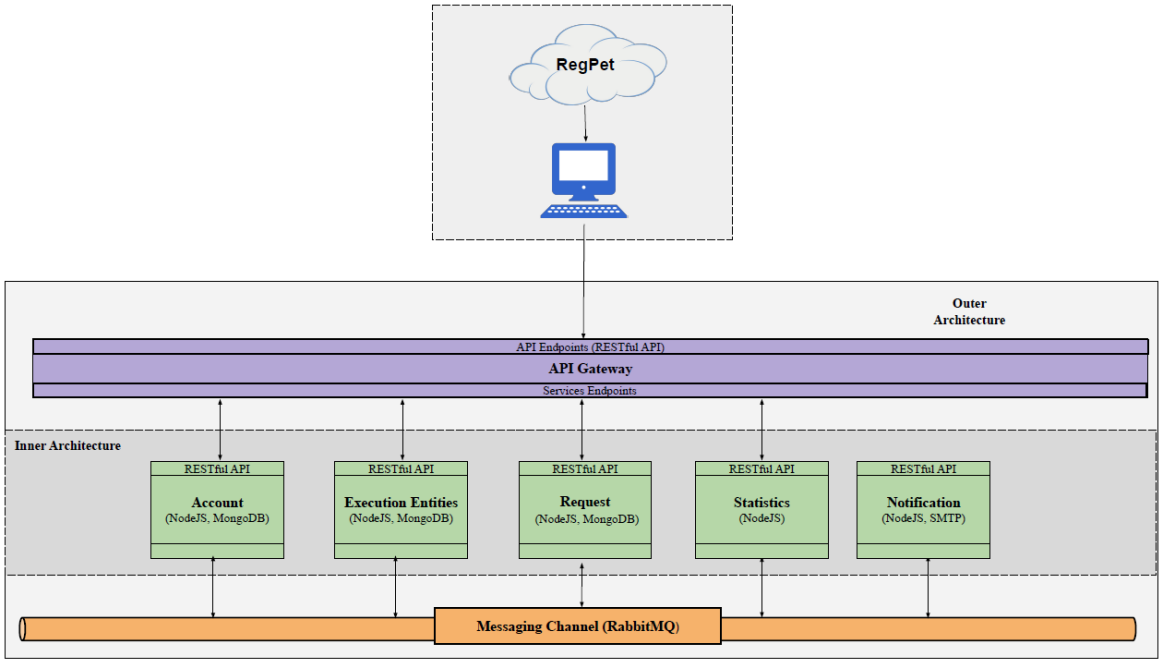
O RegPet foi desenvolvido utilizando a arquitetura baseada em microsserviços, que fornece benefícios em quesitos como escalabilidade e manutenção. Cada serviço é responsável por uma funcionalidade específica do sistema, esses módulos se comunicam entre si para fornecer as funcionalidades que o sistema possui. As subseções a seguir apresentam os principais aspectos da arquitetura:

4.1.2.1 Visão Geral da Arquitetura

Cada microsserviço é independente e associado a uma especificação de funcionalidade diferente, como gerenciamento de usuários, solicitações de procedimentos, gerenciamento de entidades de execução e vagas, envio de notificações, e geração de estatísticas. Além disso, para gerenciar a comunicação com a interface, os serviços se comunicam com a API-Gateway que é um único módulo que funciona como o portão principal para o acesso aos serviços, sendo responsável também por lidar com questões de segurança, formando assim a API que serve a aplicação front-end (interface em que o cliente interage).

Os microsserviços do sistema foram desenvolvidos utilizando a linguagem Typescript que usa o framework Node.js. A figura 8 é um diagrama que representa a arquitetura da API RegPet.

Figura 8 – Arquitetura da API do RegPet



Fonte: Elaborado pelo autor, 2025.

4.1.2.2 Comunicação entre Microsserviços

Para a comunicação interna de serviços é utilizado o canal de comunicação que é disponibilizado pelo serviço de mensageria assíncrona RabbitMQ, o qual utiliza o protocolo AMQP para troca de mensagens. Por exemplo, o microsserviço Request, responsável por funcionalidades que remetem às solicitações, envia mensagens para o microsserviço responsável pelas funcionalidades que envolve locais de execução Execution Entities, sempre que uma solicitação de um animal é aprovada, permitindo recuperar qual a clínica que aquela solicitação foi direcionada para informar ao frotn-end.

4.1.2.3 Persistência de dados e descrição dos serviços

Cada serviço possui sua própria instância de banco de dados, com essa estratégia garante-se independência e flexibilidade quanto ao armazenamento de dados do sistema. A Tabela 1 apresenta os bancos utilizados por cada serviço e descrição do mesmo.

Tabela 1 – Bancos de dados dos microsserviços do RegPet

Microsserviço	**Banco de Dados**	**Descrição**
Account	MongoDB	Gerencia usuários e seus tipos.
Request	MongoDB	Gerencia as solicitações de castração.
Execution Entities	MongoDB	Gerencia vagas e locais de atendimento.
Statistics	Não possui	Realiza consultas para gerar estatísticas relevantes sobre os procedimentos.
Notification	Não possui	Envia notificações aos usuários.

Fonte: Elaborado pelo autor, 2025.

4.1.2.4 Infraestrutura e Implantação

Os microsserviços são executados dentro de um ambiente conteinerizado, utilizando Docker e Docker Compose, dessa maneira facilitando a implantação e o gerenciamento. Cada microsserviço é configurado com seus próprios contêineres, que incluem:

- Aplicação: Código-fonte e dependências.
- Banco de Dados: Armazenamento persistido de dados.
- Mensageria: RabbitMQ para comunicação assíncrona de microsserviços.

4.2 Testes de API no RegPet

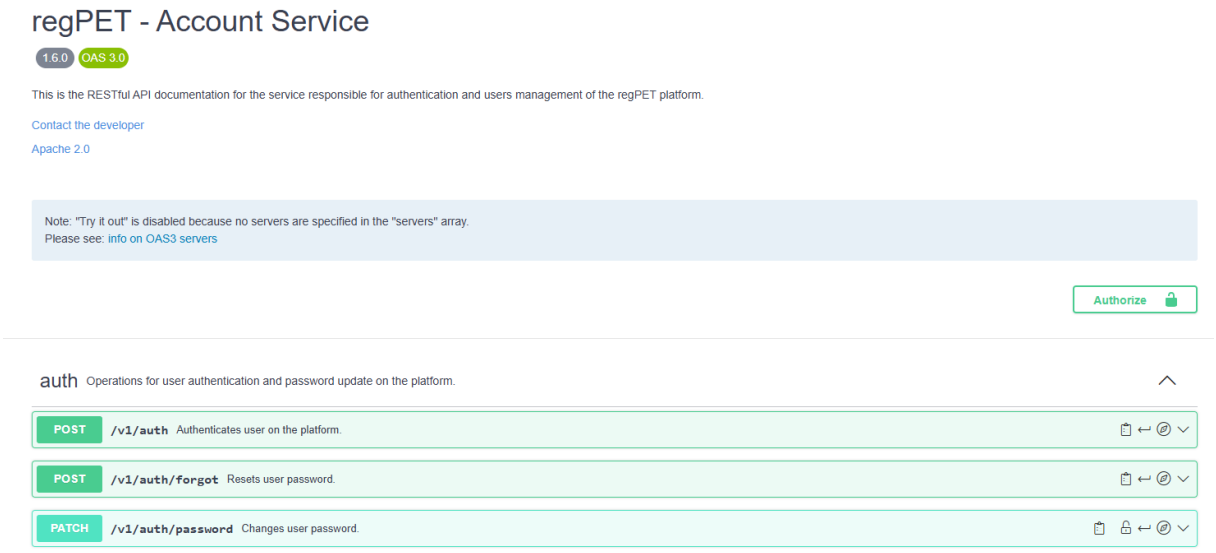
Esta seção apresentará de forma detalhada o tipo de testes que esse trabalho focou (Testes de API) nos serviços do RegPet. Em seguida, explicará o processo utilizado para fazer os testes de API do RegPet.

Os testes de API são conversam diretamente com a API-Gateway, pois ela é o elemento central que o direciona as requisições para os microsserviços. Essa arquitetura não apenas otimiza o roteamento de solicitações, mas também eleva significativamente a segurança do sistema, já que concentra e controla o ponto de entrada, simplificando o gerenciamento de permissões.

Nesse viés, para documentação das rotas expostas pela API-Gateway que se conectam às funcionalidades implementadas nos microsserviços, foi utilizado o Swagger (SmartBear Software 2024), uma ferramenta de código aberto que permite a geração de uma documentação interativa para descrever as interfaces de APIs RESTful, utilizando a especificação de padronização OpenAPI.

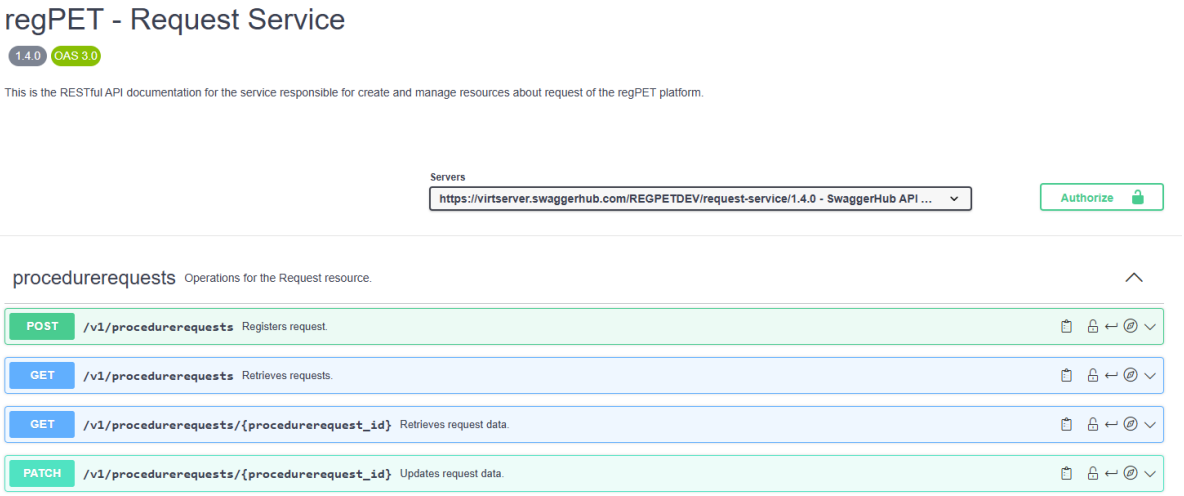
Nas Figuras 9, 10 e 11 12 são apresentadas as visualizações da documentação Swagger dos microsserviços testáveis em termos de funcionalidade do RegPet até o momento em que o trabalho foi realizado.

Figura 9 – Documentação do serviço Account



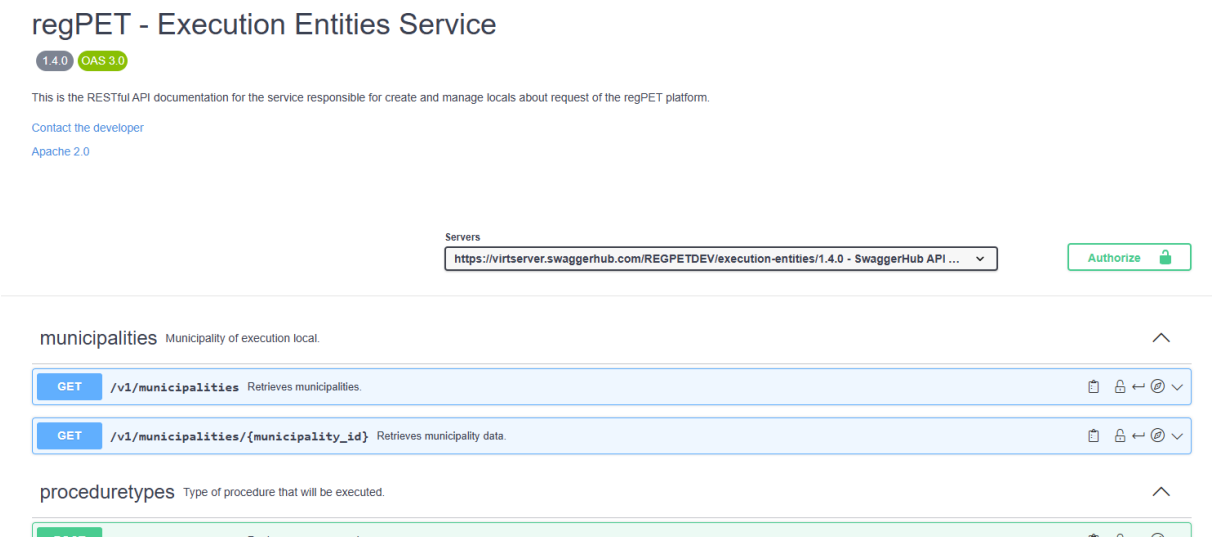
Fonte: Elaborado pelo autor, 2025.

Figura 10 – Documentação do serviço Request



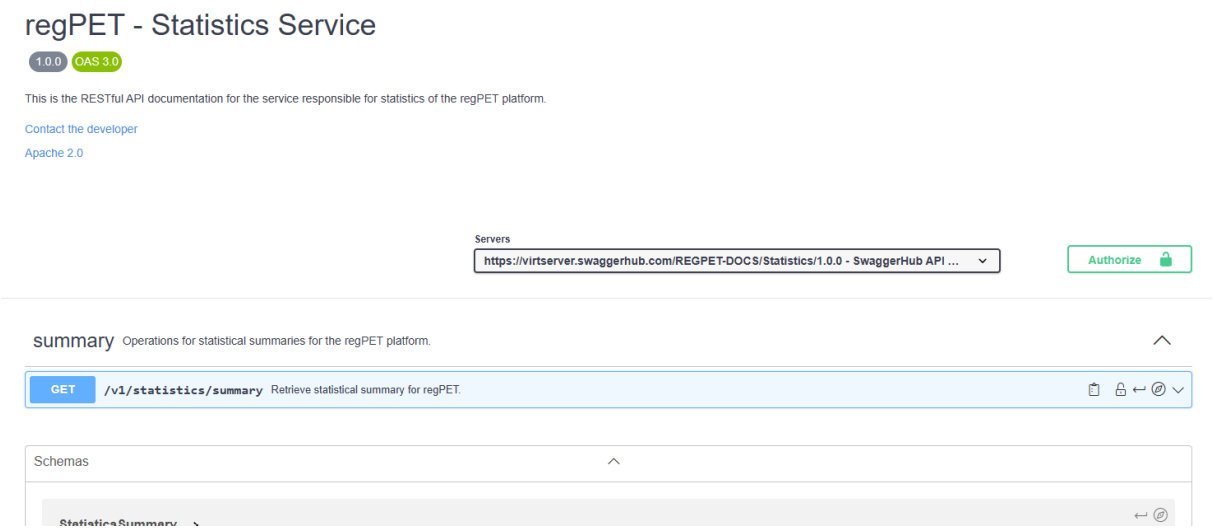
Fonte: Elaborado pelo autor, 2025.

Figura 11 – Documentação do serviço Execution Entities



Fonte: Elaborado pelo autor, 2025.

Figura 12 – Documentação do serviço Statistics



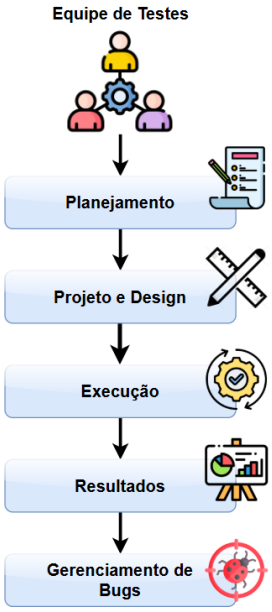
Fonte: Elaborado pelo autor, 2025.

Embora o serviço Statistics seja um serviço testável, ele não foi levado em consideração nesse estudo, pelo fato de ter sido implementado apenas na última versão da API e por ter apenas um endpoint, o que significa irrelevância para o que o trabalho pretende.

A figura 13 a seguir representa o processo utilizado para testar o RegPet. O processo inicia pelo planejamento que define as estratégias, estabelece os critérios de aceitação e mapeia os casos de teste. Na fase de projeto e design, os casos de teste são elaborados e os cenários necessários são preparados. Em seguida, na etapa de execução, os testes são aplicados ao sistema. Os resultados obtidos são analisados para obter-se uma visão

sobre a qualidade do software. Por fim, no gerenciamento de bugs, os bugs detectados são registrados e acompanhados até a resolução.

Figura 13 – Representação do processo de teste do RegPet



Fonte: Elaborado pelo autor, 2025.

4.2.1 Plano de testes

O planejamento dos testes para o RegPet foi realizado com base nas diretrizes que serão detalhadas a seguir, buscando um rigor metodológico para assegurar a qualidade:

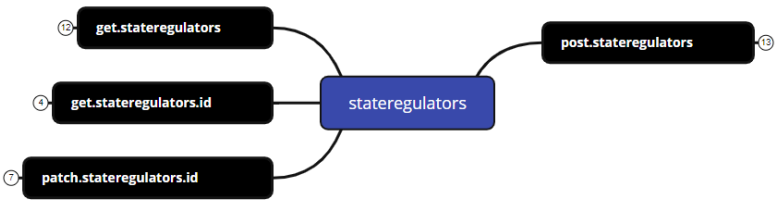
- **Escopo:** Abrangeu todas as funções disponibilizadas pela API Gateway e pelos microsserviços associados, conforme descrito na documentação oficial gerada com o Swagger, disponível em Swagger(Account, Request, Execution) e Swagger(Statistics).
- **Estratégia dos testes:** Envolveu a realização de testes funcionais para validar se os parâmetros, corpo das requisições e respostas estão conforme as descrições na documentação.
- **Objetivo dos testes:** Realizar a verificação de que todas as funções documentadas estão funcionando corretamente antes da implantação em ambiente de produção.
- **Ambiente dos testes:** Garantiu que todo o código de teste foi criado utilizando Node.js, o framework de testes Cypress, a ferramenta Cucumber, implementados usando um ambiente de desenvolvimento integrado (IDE) e executados em máquinas pelas pessoas envolvidas.

- **Critérios dos testes:**
  - Cobertura de testes: 100% das funcionalidades descritas na documentação devem ser testadas.
  - Taxa de sucesso: pelo menos 70% dos testes devem ser bem-sucedidos.
  - Erros críticos ou falhas bloqueadoras não são aceitáveis em funcionalidades essenciais.
- **Regras e responsabilidade:** Um testador será responsável por definir o plano de teste e, juntamente com outros testadores, registrar e executar casos de teste, e relatar bugs.
- **Entregáveis de teste:**
  - Antes dos testes: casos de teste.
  - Durante os testes: scripts de teste, dados de teste, logs de execução e registros de erros.
  - Após os testes: relatórios de teste detalhados e reportação de bugs.

Utilizando como base as documentações Swagger foram utilizados mapas mentais como uma abordagem de levantamento e mapeamento dos possíveis fluxos de teste. Com o auxílio da ferramenta Xmind [1], cada caso de teste foi especificado de forma a abranger todas as funcionalidades da API.

A seguir, a figura 14 representa um dos mapas mentais do serviço Account, mas especificamente o nó central e suas funcionalidades. Já a figura 15 mostra de forma detalhada os casos de testes planejados para uma funcionalidade específica.

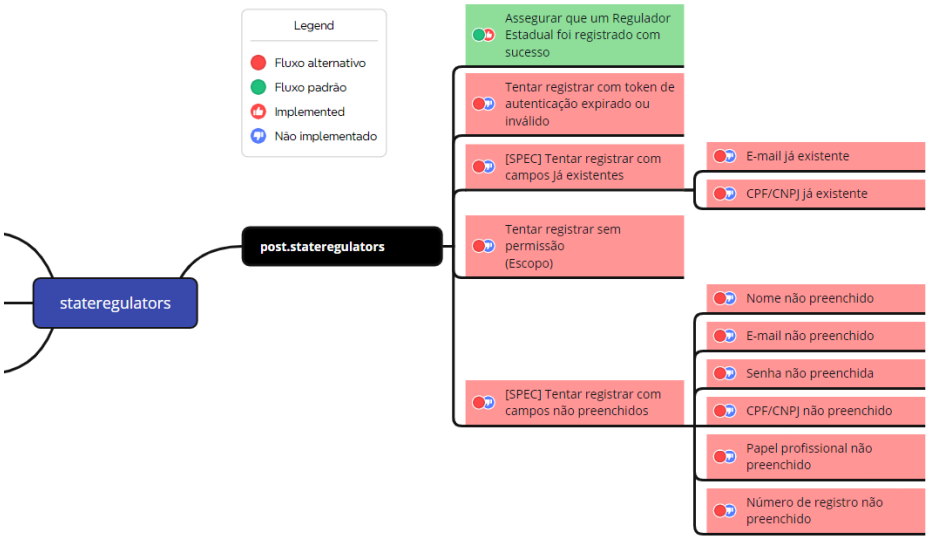
Figura 14 – Mapa mental do serviço Account para endpoints do usuário Regulador Estadual.



Fonte: Elaborado pelo autor, 2025

[1] Xmind link

Figura 15 – Mapa mental detalhado do serviço Account para endpoints do usuário Regulador Estadual



Fonte: Elaborado pelo autor, 2025.

Levando em consideração que em cada serviço existem diferentes endpoints (rotas no canal de comunicação com a API), os mapas mentais foram organizados por serviço e por endpoint. E os casos de teste foram dispostos de acordo com seu endpoint e seu método HTTP (informa o tipo de operação que deverá ser realizada), de maneira que facilite para a equipe testadora reconhecer os casos que já foram ou não implementados.

Os casos são classificados em duas modalidades, os de fluxo padrão que se referem aos fluxos desejados pelo usuário, os quais permitem a realização da funcionalidade, e os de fluxo alternativo que contraditoriamente se referem ao fluxo em que ocorrem erros e a funcionalidade não consegue ser reproduzida.

Casos que são do tipo "SPEC"são casos que envolvem campos de atributos relacionados a entidade armazenada, e que consequentemente vão possuir um número maior de possibilidades devido aos diferentes campos. Essa estratégia é utilizada para não sobrecarregar os mapas com muita informação, facilitando o reconhecimento.

4.2.2 Design dos testes

Como estratégia para design dos testes da API RegPet foi utilizada a abordagem BDD (Behavior Driven Development) para ser possível o alinhamento dos requisitos técnicos e de negócio, garantindo que todos os envolvidos tenham uma compreensão compartilhada sobre o funcionamento esperado do sistema.

Nesse contexto, cada caso de teste foi implementado seguindo a seguinte estrutura:

1. **Preparação do cenário**: onde é definido o contrato na linguagem Gherkin que irá descrever os passos que o teste irá executar.

2. **Execução da operação**: quando é feito o mapeamento de passos definidos no contrato para funções de código.

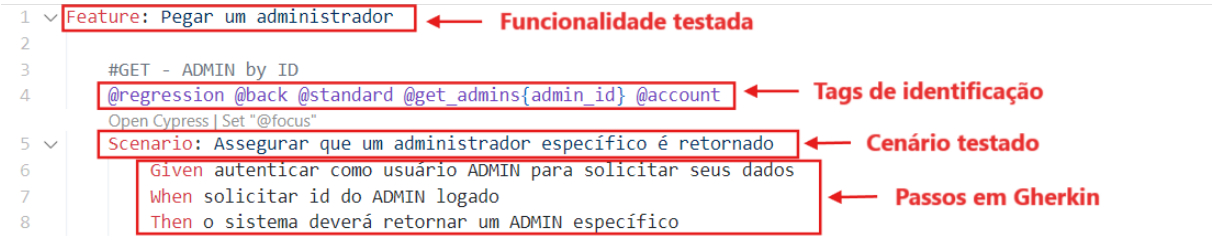
3. **Verificação dos resultados**: a resposta obtida pela requisição principal do teste é então comparada com uma resposta esperada, de acordo com a documentação da API.

Os Itens dessa estrutura podem ser observados nas figuras 16 17 a seguir:

4.2.2.1 1) Preparação do cenário

A figura 16 demonstra a definição do contrato construído na linguagem Gherkin, que contém: a identificação da funcionalidade a ser testada, as tags associadas ao teste (auxilia na rastreabilidade), a identificação do cenário testado e os passos que serão executados.

Figura 16 – Cenário de caso de teste do serviço Account para o usuário Administrador



Fonte: Elaborado pelo autor, 2025.

4.2.2.2 2) Execução da operação e 3) Verificação dos resultados

Os cenários escritos em Gherkin são convertidos em testes automatizados por meio do mapeamento de cada passo para funções de código, que utilizam o framework Cypress para a execução. Isso garante que os testes sejam diretamente conectados às funcionalidades reais da aplicação. A figura 17 a seguir demonstra o arquivo de teste principal que mapeia o Gherkin para o código do teste, que conterá: os passos que preparam os pré-requisitos, e o passo que realizará a requisição final juntamente com as verificações das respostas.

Figura 17 – Implementação do cenário da figura 16


```
//GET ONE ADMIN
Given("autenticar como usuário ADMIN para solicitar seus dados", () => {
  //Realiza login do Admin e pega seu id
  requestGetAdminsAdminIDStandard.loginAdmin().then((getDataResponse) => {
    this.tokenAuth = getDataResponse;
  });

});

When("solicitar id do ADMIN logado", () => {
  requestId
    .CapturarId(Config.USER_ADMIN, Config.PASS_ADMIN)
    .then((getDataResponse) => {
      this.idAdmin = getDataResponse
    });
});

Then("o sistema deverá retornar um ADMIN específico", () => {
  const id = this.idAdmin;
  const token = this.tokenAuth;
  requestGetAdminsAdminIDStandard
    .getOneAdmin(id, token)
    .then((getDataResponse) => {
      assertionsGetAdminsAdminIDStandard.notNull(getDataResponse);
      assertionsGetAdminsAdminIDStandard.shouldContainStatus(
        getDataResponse,
        200
      );
      assertionsGetAdminsAdminIDStandard.shouldContainDuration(
        getDataResponse,
        1500
      );
      assertionsGetAdminsAdminIDStandard.shouldContainStatusText(
        getDataResponse,
        "OK"
      );
      assertionsGetAdminsAdminIDStandard.verifyAdminId(getDataResponse, id);
      assertionsGetAdminsAdminIDStandard.verifyTimeStampsExists(
        getDataResponse
      );
    });
});
```

Fonte: Elaborado pelo autor, 2025.

Esse processo possibilitou a inclusão de validações de:

- Casos positivos e negativos para os endpoints.
- Respostas esperadas, como códigos de respostas HTTP (ex.: 200, 400, 404).
- Formatos e conteúdos das respostas (ex.: objetos JSON com campos obrigatórios).

O uso do BDD no RegPet trouxe diversos benefícios, como a documentação "viva", já que os cenários em Gherkin funcionam como uma descrição atualizada do comportamento do sistema.

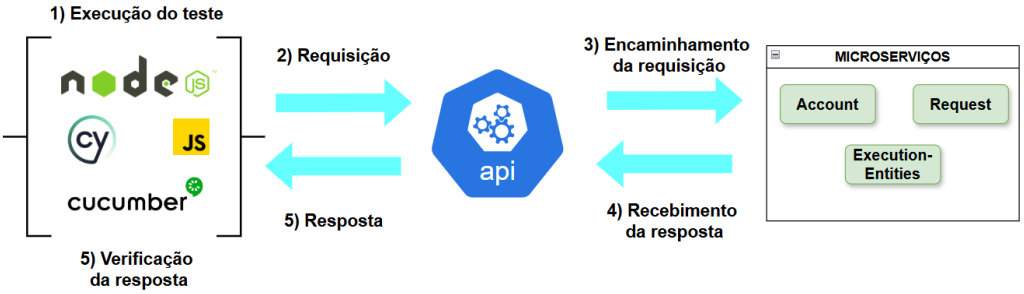
4.2.3 Execução dos testes

Os testes foram executados em um ambiente Node.js utilizando o framework Cypress, que forneceu suporte para automação. Cada caso de teste foi projetado para simular requisições feitas à API e validar as respostas recebidas.

Um adendo importante de destacar é que devido ao Cypress ser um framework criado especificamente para testes de front-end foi utilizado o plugin cypress-plugin-api para simplificar e melhorar a experiência ao testar APIs, fornecendo uma interface mais amigável para interagir com endpoints e verificar as respostas.

O processo de execução dos testes é composto por seis etapas, conforme ilustrado na Figura 18. Essa execução é realizada com o suporte do Cypress e da biblioteca do Cucumber.

Figura 18 – Fluxo de execução dos testes



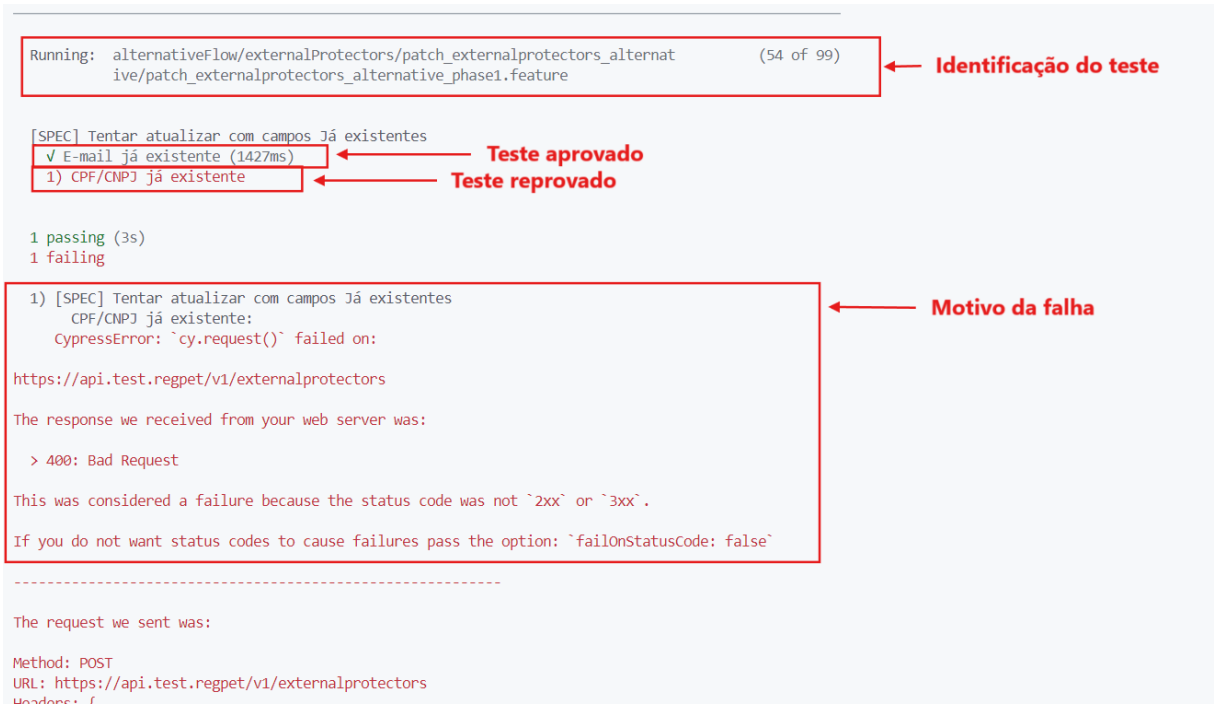
Fonte: Elaborado pelo autor, 2025.

O fluxo começa pelo teste com o envio de uma requisição para a API, que redireciona a solicitação ao microsserviço responsável por processar a funcionalidade. Após processar a solicitação, o microsserviço retorna uma resposta, que é encaminhada ao código de teste. O código compara a resposta obtida com o resultado esperado, com base no que o testador definiu olhando para na documentação da API.

Para observar a execução dos testes e seus resultados o Cypress dispõe de duas formas principais que executam quando chamamos certo script, a primeira é via linha de comando (modo headless), em que é mais focada na execução de múltiplos testes e não exibe detalhes enquanto o teste executa, apenas sobre o motivo das falhas, já a segunda forma consiste em uma interface mais complexa, onde é possível executar testes específicos de maneira interativa e observar cada passo executado pelo teste detalhadamente.

A figura 19 mostra a execução de um caso de teste com dois cenários feita por linha de comando, onde um teste é aprovado e o outro é reprovado. Também é exibido o motivo da falha do teste reprovado.

Figura 19 – Execução dos testes por linha de comando



Fonte: Elaborado pelo autor, 2025.

Ao final da execução pela linha de comando é exibido um sumário sobre os testes executados, informando a quantidade de testes em cada caso, quantos passaram, quantos falharam e quantos foram pulados (testes marcados como obsoletos) conforme ilustra a figura 20.

Figura 20 – Sumário da execução por linha de comando

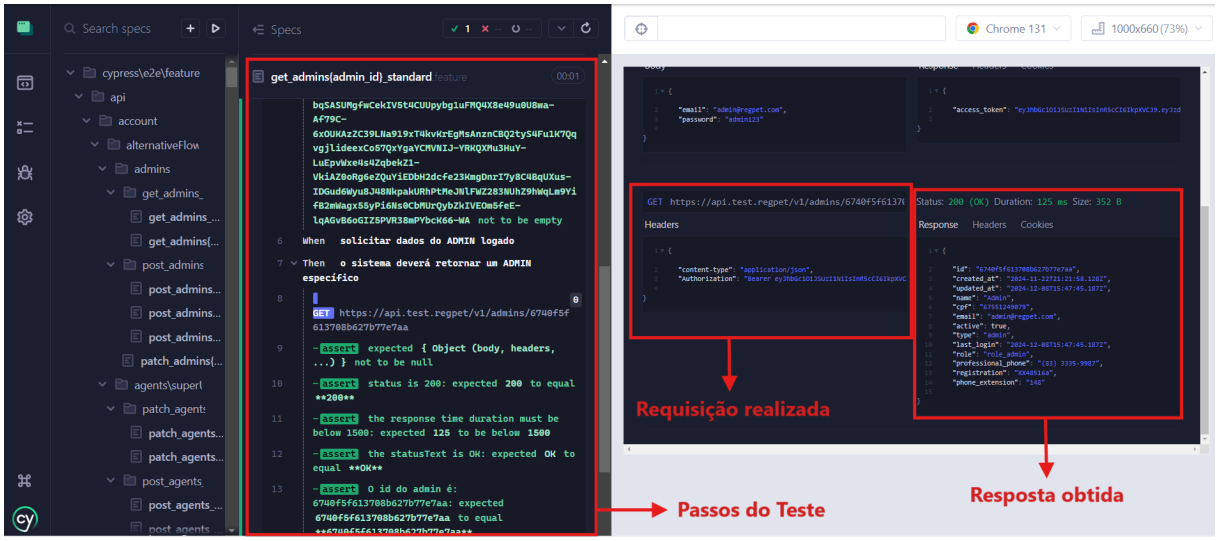
```
(Run Finished)
```

	Spec		Tests	Passing	Failing	Pending	Skipped
√	alternativeFlow/admins/patch_admins{admin_id}_alternative.feature	00:04	5	5	-	-	-
√	alternativeFlow/auth/post_auth_alternative.feature	00:04	16	16	-	-	-
√	alternativeFlow/externalProtectors/get_externalprotectors_alternative.feature	00:06	4	4	-	-	-
√	alternativeFlow/externalProtectors/get_externalprotectors{externalprotector_id}_alternative.feature	00:04	3	3	-	-	-
√	alternativeFlow/clinicManagers/get_clinicmanagers_alternative.feature	00:01	1	1	-	-	-
√	alternativeFlow/clinicManagers/get_clinicmanagers{clinicmanager_id}_alternative.feature	00:04	3	3	-	-	-
√	alternativeFlow/clinicManagers/patch_clinicmanagers_alternative.feature	00:07	3	3	-	-	-
√	alternativeFlow/clinicManagers/patch_clinicmanagers_alternative_phase1.feature	00:05	2	2	-	-	-
√	alternativeFlow/clinicManagers/patc	00:08	5	5	-	-	-

Fonte: Elaborado pelo autor, 2025.

A figura 21 mostra a execução de um caso de teste feita por interface, em que o teste é aprovado, exibindo detalhadamente os passos realizados pelo teste, bem como uma das requisições realizadas e sua resposta.

Figura 21 – Execução dos testes por interface

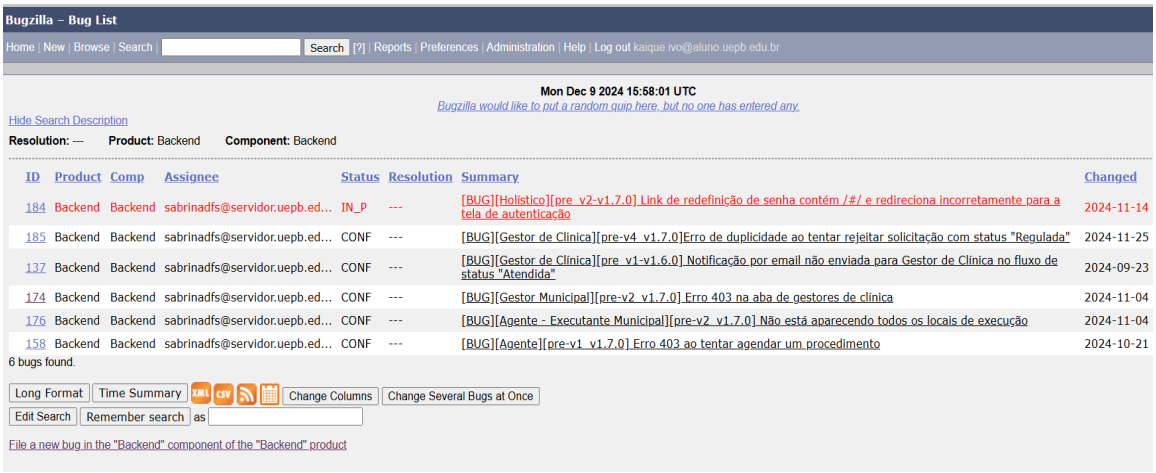


Fonte: Elaborado pelo autor, 2025.

4.2.4 Rastreamento dos bugs

O acompanhamento dos bugs detectados foi feito utilizando o Bugzilla, um software de rastreamento de erros de código aberto. Sempre que um bug era detectado durante a análise dos registros de teste e dos resultados das execuções, ele era registrado na ferramenta. A Figura 22 apresenta uma listagem com bugs reportados.

Figura 22 – Lista de bugs reportados na ferramenta Bugzilla

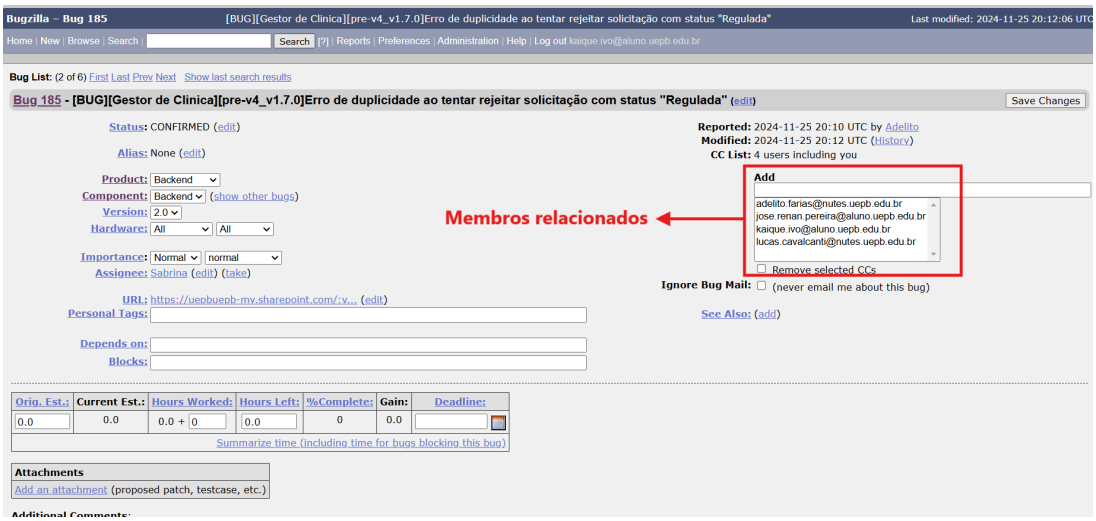


Fonte: Elaborado pelo autor, 2025.

Ao selecionar um bug na lista, o responsável pela sua resolução terá acesso aos detalhes completos, conforme ilustrado na Figura 23. Cada bug registrado inclui uma relação de membros designados para o gerenciamento e a correção do problema. Assim que um bug é

reportado ou alterado na ferramenta, um e-mail de notificação é enviado automaticamente a todos os membros listados como responsáveis.

Figura 23 – Bug reportado na ferramenta Bugzilla.



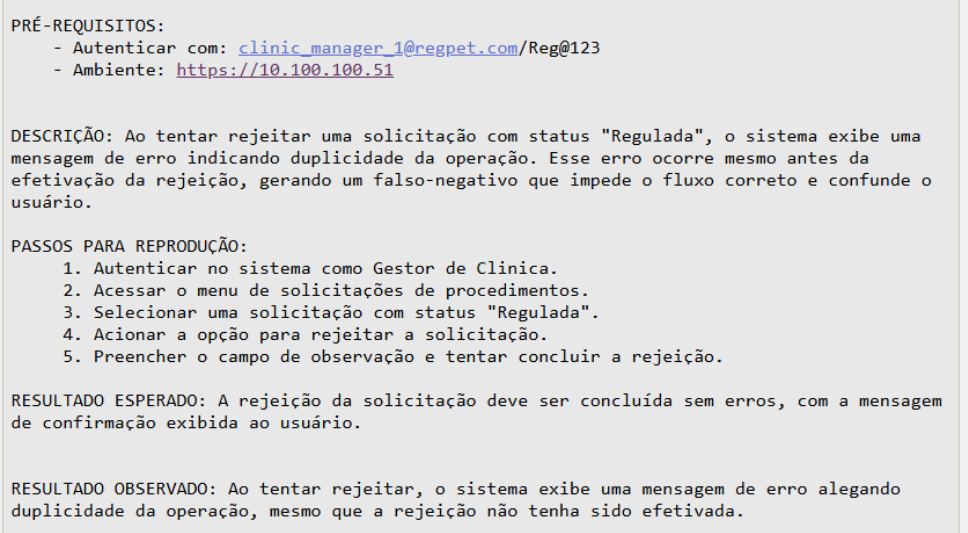
Fonte: Elaborado pelo autor, 2025.

Ainda quando um bug é selecionado na ferramenta, todas as informações de descrição explicitadas na reportação daquele determinado comportamento são mostradas, incluindo os seguintes pontos:

- Pré-requisitos: lista as condições ou configurações necessárias para reproduzir o bug, como o ambiente, dados de entrada específicos, permissões de usuário, ou versões do sistema.
- Descrição: contém uma visão geral clara e concisa do problema relatado. Deve incluir o que ocorre, em que módulo ou funcionalidade do sistema, e o impacto do bug.
- Passos para reprodução: um guia passo a passo que descreve exatamente como reproduzir o bug, garantindo que qualquer membro da equipe possa replicar o problema.
- Resultado esperado: descreve o comportamento correto do sistema se o bug não estivesse presente.
- Resultado observado: documenta o que aconteceu durante o teste, evidenciando o comportamento inesperado do sistema. Pode incluir mensagens de erro ou logs.

A figura 24 ilustra como essas informações são detalhadas no Bugzilla.

Figura 24 – Descrição de um bug no Bugzilla



PRÉ-REQUISITOS:
- Autenticar com: clinic_manager_1@regpet.com/Reg@123
- Ambiente: https://10.100.100.51

DESCRIÇÃO: Ao tentar rejeitar uma solicitação com status "Regulada", o sistema exibe uma mensagem de erro indicando duplicidade da operação. Esse erro ocorre mesmo antes da efetivação da rejeição, gerando um falso-negativo que impede o fluxo correto e confunde o usuário.

PASSOS PARA REPRODUÇÃO:
1. Autenticar no sistema como Gestor de Clinica.
2. Acessar o menu de solicitações de procedimentos.
3. Selecionar uma solicitação com status "Regulada".
4. Acionar a opção para rejeitar a solicitação.
5. Preencher o campo de observação e tentar concluir a rejeição.

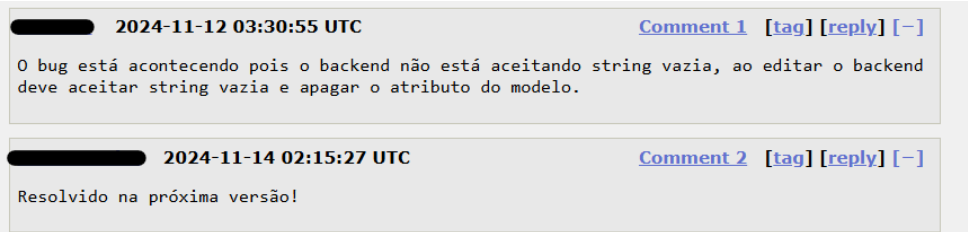
RESULTADO ESPERADO: A rejeição da solicitação deve ser concluída sem erros, com a mensagem de confirmação exibida ao usuário.

RESULTADO OBSERVADO: Ao tentar rejeitar, o sistema exibe uma mensagem de erro alegando duplicidade da operação, mesmo que a rejeição não tenha sido efetivada.

Fonte: Elaborado pelo autor, 2025.

Assim que o bug é registrado, o Bugzilla notifica todos os participantes envolvidos no processo de correção por meio de e-mail. Esse e-mail inclui informações detalhadas sobre o bug reportado, como descrição, prioridade e responsáveis. Assim que o registro do bug é alterado ou resolvido, o responsável pela ação adiciona um comentário sobre a mesma no Bugzilla. A Figura 25 apresenta o histórico de comentários realizados por membros responsáveis.

Figura 25 – Comentários dos responsáveis pelas alterações no registro do bug



[redacted] 2024-11-12 03:30:55 UTC Comment 1 [tag] [reply] [−]

O bug está acontecendo pois o backend não está aceitando string vazia, ao editar o backend deve aceitar string vazia e apagar o atributo do modelo.

[redacted] 2024-11-14 02:15:27 UTC Comment 2 [tag] [reply] [−]

Resolvido na próxima versão!

Fonte: Elaborado pelo autor, 2025.

4.2.5 Testes de API

Para implementação e execução dos casos de testes da API RegPet foi utilizada a abordagem BDD (Behavior Driven Development) que significa Desenvolvimento Orientado por Comportamento e visa focar na colaboração entre as equipes de desenvolvimento, qualidade e interessados no produto do negócio, o objetivo do BDD é alinhar os requisitos técnicos e de negócio, garantindo que todos os envolvidos tenham uma compreensão compartilhada sobre o funcionamento esperado do sistema.

4.3 Aplicação da Abordagem ao RegPet

Esta seção apresenta como a abordagem 3 foi aplicada ao sistema RegPet. Detalhando as ferramentas utilizadas para cumprir as fases especificadas. A abordagem foi implementada através de uma sequência de configurações e um script[2] para automatizar o restante das fases, o mesmo foi executado após as devidas configurações feitas em todas as versões dos ambientes.

A preparação dos ambientes foi feita em duas etapas: (i) configurar os microsserviços para disponibilizar dados sobre a execução do código em um novo endpoint, (ii) configurar o framework de testes para acessar o endpoint disponibilizado, coletar os dados e construir os relatórios.

4.3.1 Configuração e Instrumentação dos Microsserviços

Nessa etapa são realizadas atividades de configuração no código dos microsserviços, como instalação de dependências, modificação de script de inicialização e modificação do ambiente conteinerizado. Como os microsserviços são independentes, foi necessário fazê-las em cada um.

4.3.1.1 Dependências Adicionadas

Foram utilizadas as bibliotecas Nyc (v17.0.0) e cypress/code-coverage (v3.12.42). A biblioteca Nyc foi escolhida para realizar a instrumentação do código-fonte, enquanto o cypress/code-coverage foi empregado para configurar e disponibilizar um endpoint que fornece os dados de cobertura em formato "JSON"(Javascript Object Notation).

As dependências foram declaradas no arquivo "package.json"dos microsserviços, que é o arquivo principal de gerenciamento de dependências de um projeto Node.js, e foram instaladas utilizando o gerenciador de pacotes do Node.js o "npm"(Node Package Manager).

4.3.1.2 Instrumentação no Comando de Inicialização

O comando padrão de inicialização do serviço foi modificado para incluir a instrumentação do código. A instrução "node dist/server.js"foi substituída por "nyc –silent node dist/server.js", permitindo que o serviço fosse iniciado com a instrumentação do Nyc ativa.

4.3.1.3 Configuração de Dependências

As configurações do Nyc foram ajustadas para definir quais arquivos seriam instrumentados, com o intuito de delimitar a instrumentação para as partes relevantes do código,

[2]Script link

negligenciando arquivos que possuem variáveis de ambiente ou testes unitários, por exemplo, que não fazem parte do fluxo de execução dos testes de integração.

A utilização do plugin cypress/code-coverage teve o propósito de integrar ao servidor um middleware, componente atrelado a um endpoint que consegue interceptar requisições e respostas de uma API e fazer verificações ou alterações. O middleware foi o responsável por fornecer os dados vindo do código instrumentado pelo Nyc.

4.3.1.4 Configuração no Ambiente Conteinerizado

Para evitar restrições de bloqueio a middlewares extras do serviço como o que foi configurado, possibilitando o acesso ao endpoint específico, foram ajustados os arquivos Dockerfile e docker-compose.yml, expondo as portas necessárias para que o Cypress pudesse acessar os endpoints de cobertura. Além disso, nesses arquivos também foi modificado a estrutura hierárquica de pastas dos microsserviços, como uma estratégia para ajudar o Nyc (dessa vez atuando pelo lado do ambiente de testes) encontrar o código-fonte que será copiado para fora do contêiner com a mesma estruturação de pastas, pois sem isso a ferramenta não conseguirá acessar os arquivos dentro do contêiner para referenciá-los.

4.3.2 Configuração do Framework Cypress no Ambiente de Testes

Com os microsserviços da API configurados, a etapa seguinte foi preparar o ambiente de testes para coletar e processar os dados de cobertura para construção dos relatórios.

4.3.2.1 Adição de Dependências

No lado dos testes, as mesmas ferramentas Nyc e cypress/code-coverage foram adicionadas e instaladas do mesmo modo. A versão do framework de testes Cypress utilizada foi a v13.15.1, esta versão foi escolhida por ser a mais atual compatível com o plugin responsável pela integração do Cucumber o "badeball/cypress-cucumber-preprocessor".

4.3.2.2 Configuração dos Scripts de Teste

Foram definidos scripts de inicialização de testes no arquivo package.json para executar os testes de cada serviço individualmente em modo headless. O modo headless permite executar os testes no Cypress sem abrir sua interface gráfica, e rodando em segundo plano, sendo mais leve e rápido para a máquina. Além disso, por mais que a interface fique ausente, esse método utiliza os mecanismos de renderização do navegador. O navegador configurado para renderizar os testes foi o Firefox, devido à maior estabilidade.

4.3.2.3 Definições de Configuração do Framework de Testes

Os principais arquivos que definem como o framework funcionará foram modificados para importação do plugin de cobertura, além de ser adicionado a task (comandos utilizados pelo Cypress para iniciar uma tarefa) disponibilizada pelo mesmo, que executa as atividades de cobertura de código. Ademais, também foi preciso o ajuste de variáveis importantes para o direcionamento correto do framework para executar a cobertura de código, essas variáveis incluem a URL base definida para apontar ao endpoint de cobertura do microsserviço. Como as ferramentas não foram desenvolvidas especificamente para sistemas de microsserviços, a URL do endpoint é a mesma para todos os microsserviços, fazendo com que a dicotomização entre os serviços seja apenas pela porta em que cada um é executado.

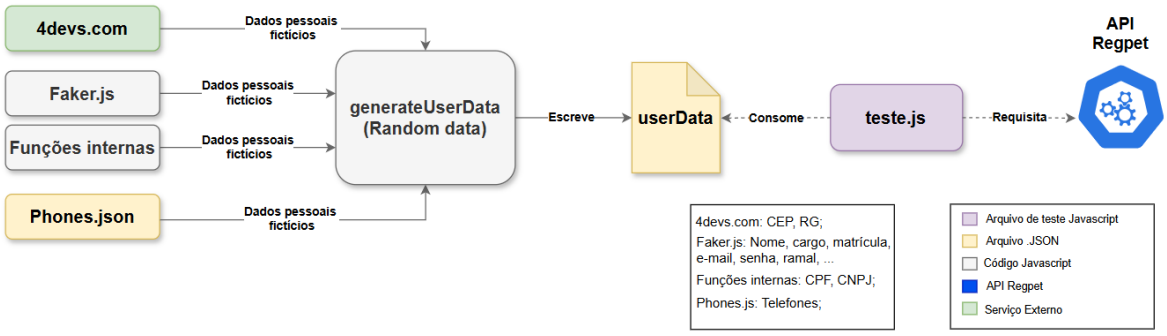
4.3.3 Conteinerização e Geração de Massas de dados

Essa fase é onde a versão escolhida da API será construída, com o auxílio da ferramenta de conteinerização Docker e sua extensão Docker Compose o ambiente da API com todos os módulos é empacotado e levantado. Como cada microsserviço possui suas próprias versões, o controle de versionamento da ferramenta utilizada Git (Git 2025) faz com eles sejam empacotados de acordo com a versão geral adequada do sistema.

Sendo assim, no ambiente de testes é executado o script responsável por integrar os processos, realizando inicialmente o versionamento e levantamento dos microsserviços, e inicializando o gerador de massas de teste que o sistema RegPet possui. Essa estratégia é uma forma de conseguir dados parametrizados para serem consumidos pelos testes ao realizarem diversas requisições no sistema. A geração de massas de dados para testes de API é uma estratégia importante no processo de validação de funcionalidades, pois ela auxilia os testadores em um de seus desafios que é encontrar dados ideais para testar partes do sistema que precisam de dados específicos (**?**), uma vez que não podemos utilizar dados reais do ambiente de produção segundo a Lei Geral de Proteção de Dados (Brasil 2018). Nos testes, os dados fictícios gerados simulam as entradas que o cliente realiza através da interface web.

A figura 26 a seguir representa o funcionamento interno de como os dados para teste são gerados no RegPet.

Figura 26 – Representação da geração de dados no RegPet



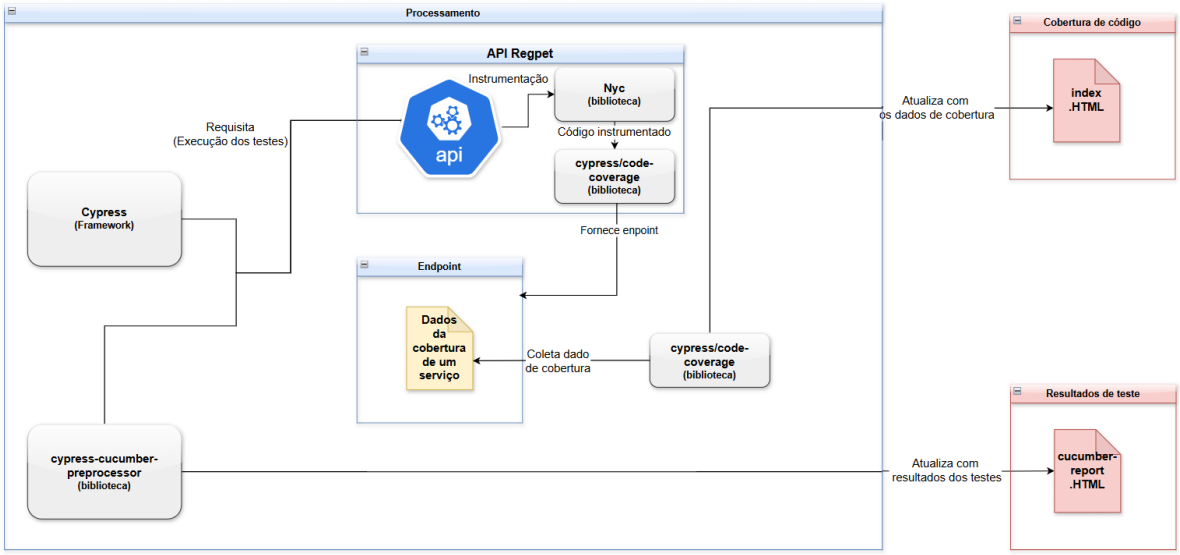
Fonte: Elaborado pelo autor, 2025.

4.3.4 Execução de Testes e Geração de Relatórios

Nessa última fase da aplicação metodológica é onde os testes de integração do RegPet são executados e seus resultados tanto das verificações de teste quanto de cobertura de código são extraídos. Esse processo iterativo se caracteriza pela atuação conjunta do framework de testes, microsserviços e as demais ferramentas utilizadas.

Na figura 27, foi elaborado um diagrama ilustrando o fluxo de instrumentação, execução de testes e coleta de dados. Ele demonstra a interação entre os microsserviços instrumentados e conteinerizados na API RegPet, o Cypress, as ferramentas e o endpoint configurado, evidenciando o processo de geração dos relatórios de cobertura de código e de resultados de teste.

Figura 27 – Representação da execução dos testes e do funcionamento da obtenção dos relatórios



Fonte: Elaborado pelo autor, 2025.

É importante destacar que os relatórios que contém os resultados das verificações dos testes (resultados de teste) são gerados nativamente pela integração das bibliotecas responsáveis pelo Cypress e pelo Cucumber, já os que contém a cobertura de código desses testes são o resultado da interação da biblioteca do Nyc que observa o código-fonte para gerar dados (ambiente da API) e do plugin cypress/code-coverage específico para cobertura de código que implementa o middleware e o endpoint (ambiente da API) para coletar os dados e gerar os relatórios no ambiente de testes. Essas últimas ferramentas trabalham em ambos os lados, formando uma estratégia essencial para obter dados de cobertura de um ambiente conteinerizado como o RegPet.

4.4 Resultados dos relatórios

A abordagem foi aplicada a quatro versões do RegPet e, como resultado, além da automação do processo de execução de testes em diferentes versões (conforme exposto nas seções anteriores), coletamos métricas relacionas às essas execuções. A Tabela 2 expõe as versões dos microsserviços, bem como suas características: quantidade de endpoints (#ENDPOINTS) e quantidade de testes (#TESTES) e testes obsoletos (#T_OBSOLETOS).

Tabela 2 – Características das versões dos serviços

SERVIÇO	#ENDPOINTS	#TESTES	#T_OBSOLETOS
Account v1.0	39	174	0
Account v2.0	39	188	0
Account v3.0	36	231	8
Account v4.0	40	262	8
Request v1.0	9	40	0
Request v2.0	9	40	0
Request v3.0	11	49	1
Request v4.0	12	51	2
Execution-Entities v1.0	17	63	0
Execution-Entities v2.0	17	63	0
Execution-Entities v3.0	17	63	0
Execution-Entities v4.0	22	87	0

Fonte: Elaborado pelo autor, 2025.

Já a Tabela 3 apresenta e as métricas coletadas durante a execução dos testes nas versões dos serviços, tais como: quantidade (#FALHAS) e porcentagem (%FALHAS) de falhas, cobertura de linhas de código (#COBERTURA) e tempo de execução (#TEMPO_EXEC).

4.4.1 Análise das Falhas dos Testes

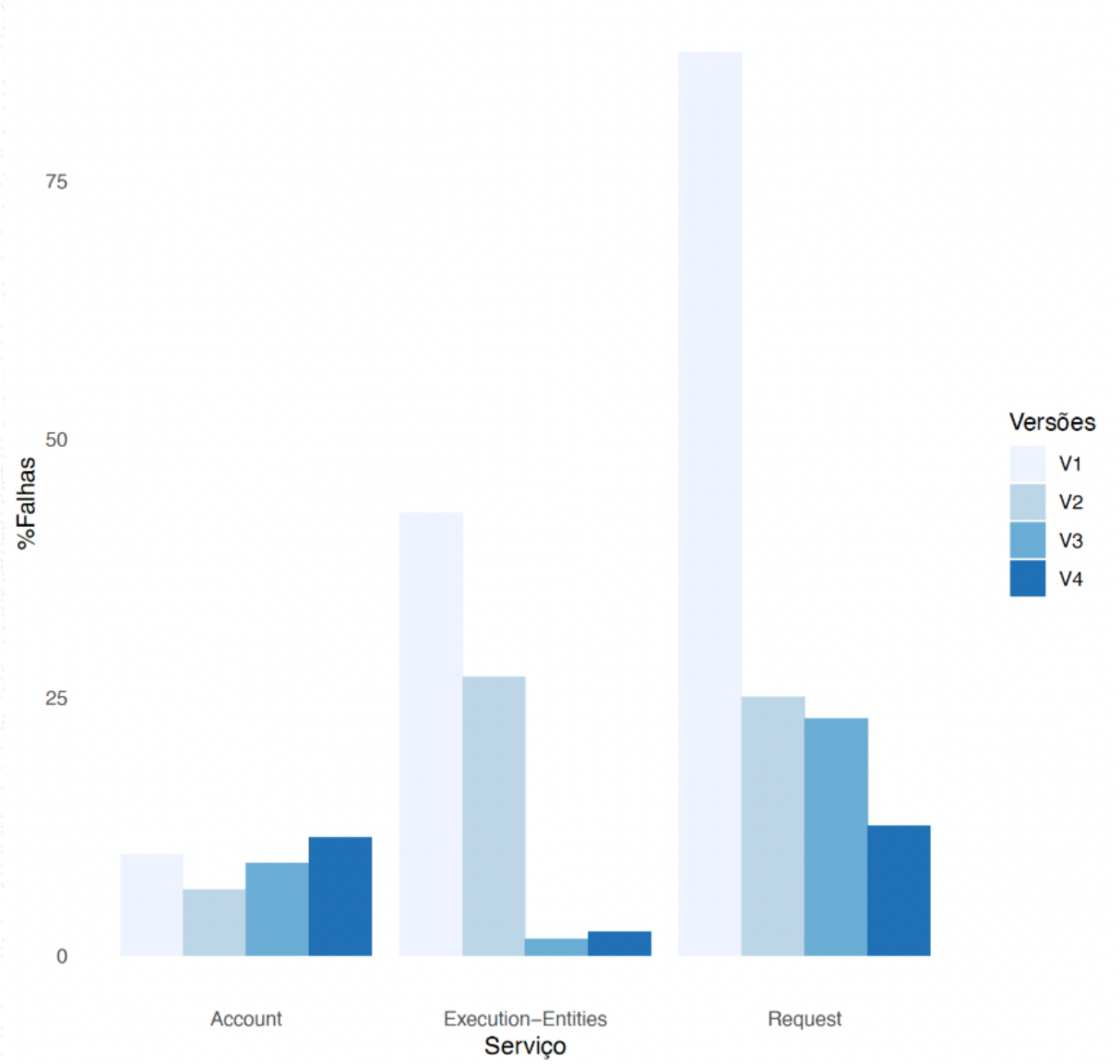
Nesta sub-seção analisaremos as falhas dos testes ao longo das versões dos três microsserviços apresentados. O gráfico da figura 28 a seguir permite uma melhor visualização para a análise da evolução dos módulos sob o aspecto de falhas.

Tabela 3 – Resultados dos relatórios

SERVIÇO	#FALHAS	%FALHAS	#COBERTURA	#T_EXEC
Account v1.0	17 em 174 testes	9,77%	81,63%	00:29:25
Account v2.0	12 em 188 testes	6,38%	82,66%	00:30:34
Account v3.0	20 em 223 testes	8,97%	78,89%	00:41:37
Account v4.0	29 em 254 testes	11,42%	81,18%	00:45:11
Request v1.0	35 em 40 testes	87,50%	49,4%	00:04:15
Request v2.0	10 em 40 testes	25,00%	71,5%	00:04:57
Request v3.0	11 em 48 testes	22,92%	69,4%	00:06:05
Request v4.0	6 em 49 testes	12,50%	76,43%	00:10:23
Execution-Entities v1.0	27 em 63 testes	42,86%	69,63%	00:07:35
Execution-Entities v2.0	17 em 63 testes	26,98%	73,59%	00:08:45
Execution-Entities v3.0	1 em 63 testes	1,59%	73,85%	00:08:49
Execution-Entities v4.0	2 em 87 testes	2,30%	73,47%	00:13:25

Fonte: Elaborado pelo autor, 2025.

Figura 28 – Gráfico sobre as falhas nos testes dos microsserviços do RegPet



Fonte: Elaborado pelo autor, 2025.

4.4.1.1 Account

A evolução do serviço Account em relação às falhas e à quantidade de testes mostra um cenário misto. Inicialmente, na mudança da primeira versão (v1.0) para a segunda (v2.0) a quantidade de testes aumentou de 174 para 188, as falhas diminuíram de 17 para 12 e a taxa de falhas também diminuiu de 9,77% para 6,38%, o que indica uma provável melhoria na qualidade dos testes ou no código.

No entanto, na terceira versão, apesar de um aumento significativo para 231 testes, as falhas subiram para 20, elevando a taxa para 8,97%. Esse crescimento pode estar relacionado à introdução de novos cenários de teste que identificaram problemas previamente desconhecidos ou a regressões causadas por mudanças no código. Isso faz sentido, pois nesse momento também houve redução na quantidade de endpoints e o surgimento de testes obsoletos, que são indícios de que houve aumento de complexidade do serviço.

Na versão v4.0, com 262 testes, as falhas aumentaram para 29, atingindo a maior taxa de falhas (11,42%). Isso sugere que, embora o número de testes tenha crescido, a eficácia deles pode não estar acompanhando a complexidade do serviço, especialmente com a adição de novos endpoints. Portanto, enquanto os testes do Account demonstram um esforço contínuo para validar as funcionalidades, a qualidade desses testes precisa de um pouco mais de atenção para garantir que as novas funcionalidades e as mudanças não introduzam regressões.

4.4.1.2 Request

Já o módulo Request se destaca por ter uma melhoria significativa ao longo das atualizações, mesmo começando com uma situação um tanto alarmante. Na versão v1.0, com 40 testes, foram registradas 35 falhas, resultando em uma taxa de falhas chamativa de 87,50%, indicando sérios problemas de qualidade do sistema, mas sem dispensar a ideia de testes não tão confiáveis. Na v2.0, ainda com 40 testes, as falhas caíram drasticamente para 10, reduzindo a taxa para 25,00%, o que demonstra um esforço eficaz para corrigir os problemas identificados. Na v3.0, com 49 testes, os resultados aumentaram em 1 falha, mas devido à proporção com os novos testes, resultou em uma taxa de 22,92%, mostrando uma melhoria contínua. Por fim, na v4.0, com 51 testes, as falhas caíram para 6, atingindo a menor taxa de falhas (12,50%).

A evolução desse microsserviço indica que ele passou por um processo que envolveu mais atenção para o mesmo. A redução consistente das falhas, mesmo com o aumento do número de testes e de endpoints, sugere que a equipe conseguiu identificar e resolver problemas críticos, além de implementar testes de confiança.

4.4.1.3 Execution Entities

O microsserviço Execution Entities também revelou um boa evolução no quesito falhas, mesmo apresentando uma quantidade de testes e endpoints indiferentes nas primeiras versões, seu percentual de falhas foi de 42,86% a 2,3%, se destacando como o serviço que mais reduziu falhas.

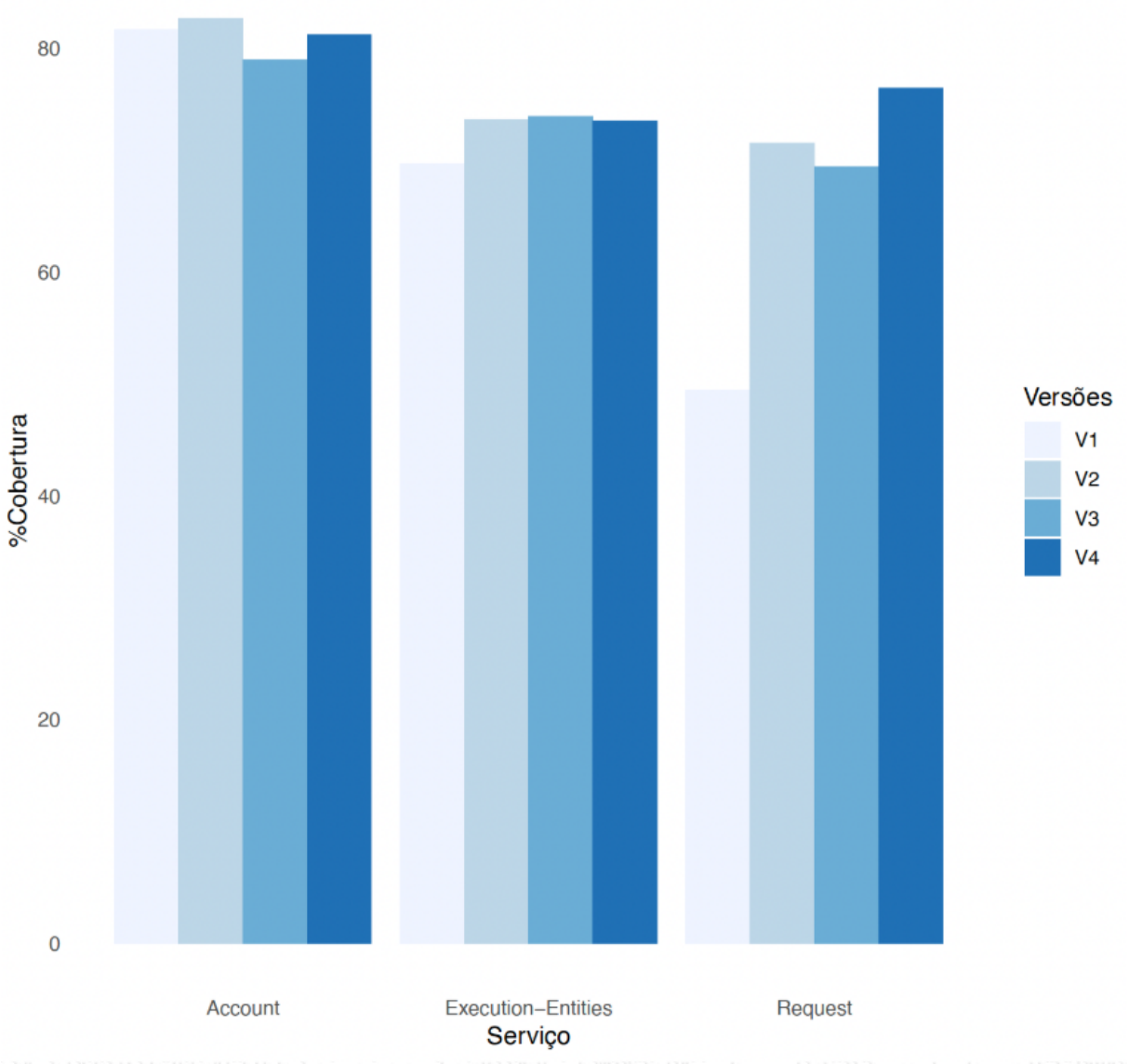
Nas versões v1.0, v2.0 e v3.0 com o mesmo número de 63 testes, as falhas foram de 27 para 17 e depois para 1, resultando em taxas de 42,86%, 26,98% e 1,59% respectivamente, um marco notável que reflete uma melhoria drástica nas verificações do serviço. Por fim, na v4.0, com 87 testes, as falhas aumentaram rasoavelmente para 2, mas a taxa de falhas permaneceu baixa, em 2,30%.

Essa variação nos termos de falhas de testes, permitiu inferir que a equipe conseguiu manter uma boa manutenção dos testes já existentes quando a quantidade de endpoints não mudou, e mesmo quando mudou bruscamente os testes novos e já existentes não tiveram comportamento extremamente crítico.

4.4.2 Análise da Cobertura do Código

Nesta sub-seção analisaremos a cobertura de linhas de código resultante da execução dos testes ao longo das versões dos três microsserviços apresentados. Com o gráfico da figura 29 é possível obter uma visão macro sobre os dados de cobertura para a análise da evolução dos módulos.

Figura 29 – Gráfico sobre a cobertura de código dos microsserviços do RegPet



Fonte: Elaborado pelo autor, 2025.

4.4.2.1 Account

Em termos de cobertura de código, o Account teve uma trajetória relativamente estável, mas com oportunidades de melhoria. Na v1.0, a cobertura foi de 81,63%, indicando uma base sólida de testes em relação ao código existente. Na v2.0, a cobertura aumentou levemente para 82,66% mesmo com a quantidade de testes subindo, o que reflete uma melhor qualidade dos testes, mas nesse momento os endpoints se manteram na mesma quantia, o que pode significar número de testes ainda insuficientes. Na v3.0, houve uma pequena queda para 78,89%, possivelmente devido à introdução de novos códigos ou endpoints que não foram totalmente cobertos pelos testes. Na v4.0, a cobertura recuperou-se para 81,18%, mostrando um esforço para corrigir a queda anterior, mas

ainda abaixo do pico da v2.0.

Essa estabilidade em torno de 80% sugere que os testes desse microsserviço sejam consideravelmente eficazes, mas o crescimento limitado indica que novos códigos ou funcionalidades podem não estar sendo totalmente cobertos. Mas uma vez na transição da segunda versão para a terceira foi observado que mudanças bruscas no serviço podem ter acontecido, pois assim como no aspecto de falhas a cobertura de código quando parecia melhorar, pioraram mesmo que a quantidade de testes aumentara.

4.4.2.2 Request

Assim como no aspecto de falhas, a evolução do Request no quesito cobertura de código mostrou melhoria significativa ao longo do tempo, refletindo um esforço real para aumentar a abrangência dos testes. Da versão v1.0 para a v2.0 a cobertura foi de 49,4% para 71,5%, após isso uma leve queda na v3.0 que pode estar relacionada à introdução de novos códigos ou funcionalidades incompletamente acobertadas pelos testes. Na última versão a cobertura alcançou 76,43%, o maior valor registrado, indicando uma melhoria contínua na abrangência dos testes.

O salto de 49,4% para 76,43%, indica que a equipe priorizou a implementação de testes para cobrir mais caminhos alternativos no código. No entanto, esses dados mostram que ainda há espaço para melhorias nos testes, especialmente com a adição de novos endpoints e funcionalidades.

4.4.2.3 Execution Entities

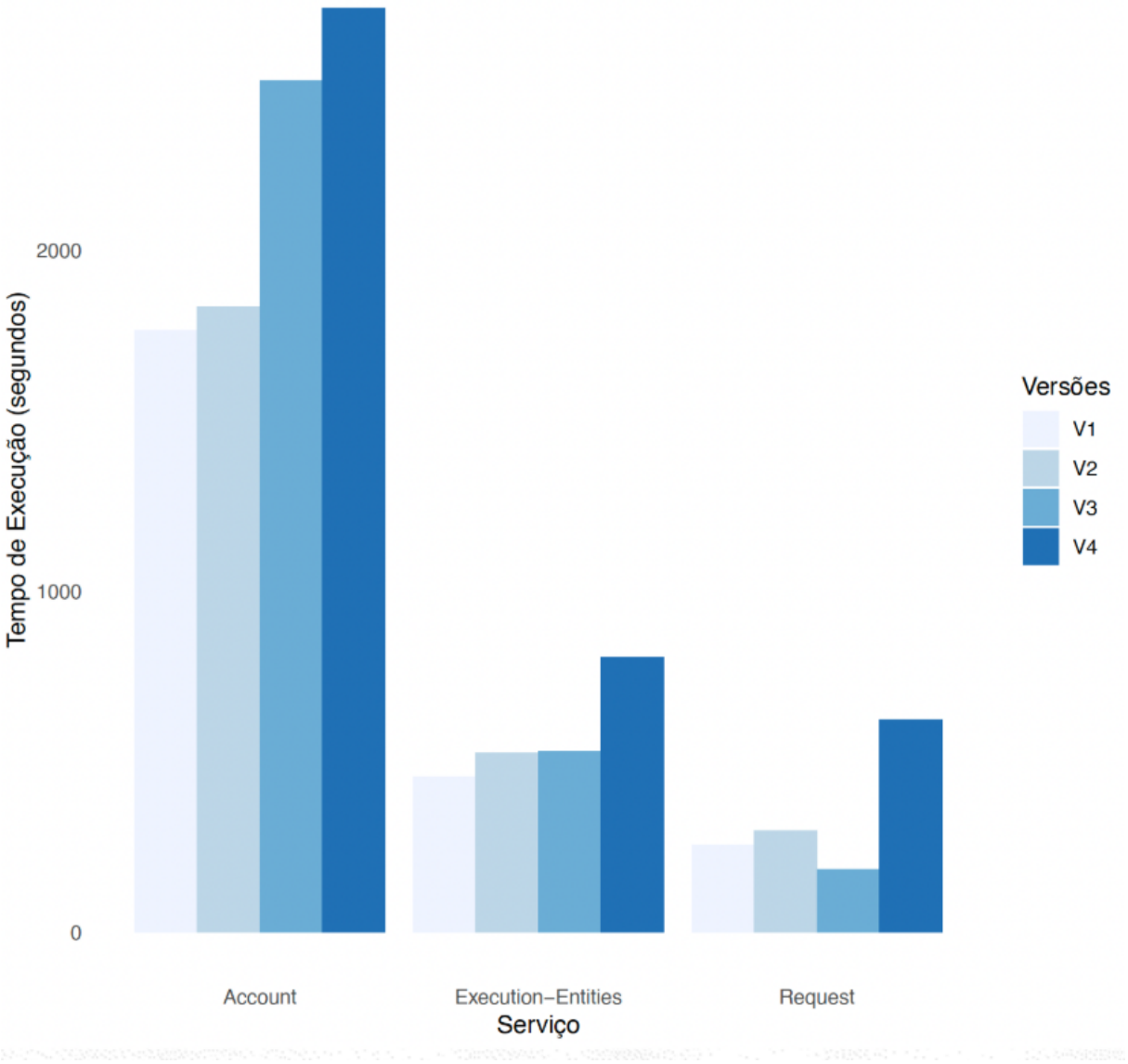
No aspecto da cobertura de código do Execution Entities, assim como o Account, também demonstrou consistência na implementação de testes, inicialmente alcançando uma cobertura de 69,63% e depois estagnando entre 73% e 74%, essas últimas aconteceram mesmo quando o número de endpoints e testes aumentaram, isso pode ter sido possível graças aos testes terem sido consistentes também no quesito falhas.

Chamamos atenção sobre a evolução da versão v1.0 para a v2.0 especificamente, em que embora muitos testes tenham parado de falhar, a cobertura de código se manteve quase inalterada. Isso nos traz uma evidência empírica de que o fato de mais testes passarem não necessariamente implica em uma maior cobertura de código, dessa forma corroborando com as observações apresentadas na subseção 2.2.4 do capítulo 2.

4.4.3 Análise do Tempo de Execução

Nesta sub-seção analisaremos o tempo de execução dos testes ao longo das versões dos três microsserviços apresentados. O gráfico da figura 30 representa de forma clara os dados de tempo de execução para a análise da evolução dos módulos.

Figura 30 – Gráfico sobre o tempo de execução dos testes dos microsserviços do RegPet



Fonte: Elaborado pelo autor, 2025.

4.4.3.1 Account

Ao longo das versões do serviço Account em relação ao tempo de execução dos testes é observado um aumento estável, refletindo o crescimento dos testes em complexidade e volume. Em sua primeira versão (v1.0), o tempo de execução foi de 29 minutos e 25 segundos, com 174 testes. Na segunda (v2.0), houve aumento para 188 testes, fazendo o tempo subir para 30 minutos e 34 segundos, um incremento moderado. Já na transição para a terceira versão (v3.0), o número de testes saltou para 231, e o tempo de execução aumentou significativamente para 41 minutos e 37 segundos, indicando que o crescimento no volume de testes impactou diretamente a duração da execução. Por fim, na última

(v4.0), com 262 testes, o tempo atingiu um pico de 45 minutos e 11 segundos, o maior registrado.

Esse aumento progressivo no tempo de execução está alinhado com o crescimento do número de casos de testes e, possivelmente, com a mudança na quantidade de endpoints.

4.4.3.2 Request

Assim como no Account, o serviço de Request experienciou um aumento de tempo na execução dos testes alinhado com o crescimento do número de testes e de endpoints, especialmente nas últimas versões.

Embora inicialmente o tempo ainda seja relativamente baixo em comparação com outros serviços (4 a 6 minutos), o crescimento acelerado na última versão (de 6min para 10min) sendo que só aumentou em 1 teste, também pode indicar que os testes estão se tornando cada vez mais complexos.

4.4.3.3 Execution Entities

Da mesma maneira que nos outros dois microsserviços, o Execution Entities mostrou um aumento gradual em relação ao tempo de execução dos testes, com pequenas variações, mas ainda também alinhados com o crescimento do número de casos de teste e de enpoints.

Nas versões com a mesma quantidade de testes o tempo aumentou pouco significativamente, abrindo espaço para cogitar sobre manutenções no código de testes exitente que refletiram o aumento da complexidade dos mesmos. No entanto, na transição para a última versão (v4.0) que veio com 5 endpoints a mais e consequentemente também aumentou em casos de teste, o tempo de execução deu um salto relativamente maior.

4.4.4 Considerações finais

Primeiramente, é importante considerar que o sistema RegPet começou a versionar os testes de sistema a partir da versão referente a primeira versão de testes que a abordagem deste trabalho utilizou, devido a este acontecimento a primeira versão de testes sofreu certas inconsistências e o serviço Request foi o mais afetado por isso. A partir desse raciocínio enfatizamos que no período em que a versão do sistema foi lançada, como não existia versão anterior de testes para se tomar como referência e utilizar abordagens de detecção de regressão na versão de sistema lançada naquele período, os testes dessa versão foram criados "às cegas" com relação a nova versão, olhando apenas para as funcionalidades novas.

Devido a esse acontecimento, o Request apresentou muitas falhas nos testes, como se pôde observar nos resultados. Em contrapartida, como as seguintes versões de teste já possuíam versões anteriores, de acordo com os resultados obtidos que fornecem uma visão ampla da evolução do sistema, percebemos que foi possível reverter esse cenário.

Nesse sentido, conseguimos extrair uma lição sobre o quão importante é para a garantia de qualidade do software utilizar o versionamento de testes de forma inteligente para conseguir extrair informações de como o sistema regrediu com uma nova atualização.

Outro ponto importante de enfatizar foi percebido com os dados de evolução do Account, que foram os indícios de mudança na complexidade de funcionamento desse serviço. Observa-se que até a terceira versão a cobertura de código foi inversamente proporcional a taxa de falhas, porém quando transitado para a última versão (v4.0) esse comportamento não foi seguido, mas ambas as métricas pioraram. Paralelo a isso, a quantidade de endpoints diminuiu depois aumentou ao longo das duas últimas versões, além de surgirem testes obsoletos. Tais indicadores apontam para um possível aumento brusco de complexidade nesse módulo do sistema.

É possível perceber o mesmo comportamento de inversão proporcional dessas métricas nos outros serviços também. No Execution Entities, serviço reparado como o que evoluiu mais consistente em termos dos parâmetros analisados, em todas as evoluções isso foi consistente. Já no Request, o cenário se assemelhou ao do Account, onde quando o número de endpoints teve uma mudança brusca as métricas não mudaram de maneira inversa, com exceção da atualização para a última versão, o que pode ser devido pelo fato de aumentar em somente 2 endpoints.

Analisando este cenário, não podemos dispensar a hipótese de que a cobertura de código tende a ser inversamente proporcional a taxa de falhas quando o sistema não muda em seu funcionamento bruscamente, o que faz sentido, pois se a qualidade dos testes é boa eles tendem a verificar mais caminhos críticos do código, consequentemente aumentando a métrica da cobertura.

5 CONCLUSÃO

Contemplando o cenário atual em que o uso da arquitetura de microsserviços vem se popularizando, é evidente a implicância de mais opções de ferramentas, estratégias e abordagens específicas para esse tipo de sistema. Tal arquitetura é de fato um grande benefício para o contexto de evolução de um software, trazendo diversos benefícios, como maior facilidade de escalar e de manter, por exemplo. Entretanto, assim como outras formas de desenvolver software, essa também possui seus obstáculos quando o sistema enfrenta mudanças.

Como observado anteriormente, sistemas baseados em microsserviços são frequentemente assombrados por desafios como a análise do impacto de mudanças e a negligência de atenção ao delimitar suas funcionalidades, os quais podem causar consequências diretas ou indiretas no monitoramento e na testabilidade. Ainda destaca-se também a carência de estudos empíricos com ferramentas específicas para esse tipo de software.

Nesse contexto, este trabalho desbravou as possibilidades a respeito da qualidade de sistemas distribuídos, apresentando problematizações encontradas na literatura e uma abordagem metodológica para alavancar a qualidade em sistemas desse nível em uma situação de evolução de suas versões, tendo como estudo de caso o sistema RegPet.

A metodologia proposta mostrou potencial ao se deparar com desafios da evolução de software, especificamente com relação à complexidade de gestão de versões dos microsserviços, execução de testes de integração e coleta de métricas importantes sobre a qualidade.

As etapas de configuração para obter a cobertura de código dos microsserviços foi importante, pois estabelece uma estrutura que viabiliza o monitoramento e a evolução vivenciada pelo sistema. A instrumentação individual dos serviços e implementação do endpoint específico para coleta de métricas, possibilita criar um ambiente controlado e reproduzível usando a conteinerização, o que garante consistência para as execuções de teste. A implementação da automatização por meio de um script pôde proporcionar ganhos em eficiência operacional, já que permitiu integrar a alternância entre diferentes versões do sistema e seus respectivos testes ao processo do fluxo de testes de integração, que envolveu geração de dados de teste e coleta de métricas de cobertura.

Nesse viés, a estratégia não apenas simplificou processos complexos que seriam trabalhosos se realizados manualmente, mas também forneceu uma visão mais clara e organizada sobre a qualidade do código e a efetividade dos testes ao longo da evolução do sistema.

O estudo aplicado ao RegPet conseguiu obter resultados realistas sobre a situação do sistema. Os relatórios gerados referentes aos três microsserviços em quatro versões do sistema possibilitou a criação de tabelas representativas, as quais fornecem a visão geral

de evolução para ser analisada. Com a análise, foi observado que no geral os serviços Request e Execution Entities conseguiram uma evolução relevante, diminuindo suas falhas e aumentando em cobertura, o que indica testes ficando melhores e mais abrangentes. Já o Account, apesar de iniciar com métricas melhores, teve um leve decaimento, apontando mais possibilidade de regressão. Porém, isso pode ser devido esse serviço ser maior em número de endpoints e testes, aumentando a probabilidade de ser mais complexo.

O trabalho também evidenciou a importância do versionamento adequado dos testes, como demonstrado pelos resultados iniciais do microsserviço Request. Ademais, também foi perceptível uma ligeira ligação entre falhas nos testes e a cobertura alcançada.

Sendo assim, a abordagem utilizada permite em oportunidades futuras ser incorporada continuamente pelo sistema RegPet, em caso de futuros lançamentos de versões do sistema e de testes, contribuindo para obter um monitoramento na evolução do seu produto. Se guiando por essa estratégia facilitadora, as equipes do sistema de acompanhamento de procedimentos animais poderão direcionar melhor a sua atenção para funcionalidades específicas percebidas como mais frágeis.

Além disso, tal abordagem também pode ser aplicadas em outros contextos que envolvem o desenvolvimento de sistemas que se assemelhem com o RegPet. Por fim, estudos futuros podem utilizar do que foi desenvolvido para expandir abordagens que ofereçam eficiência para o gerenciamento da qualidade em sistemas baseados em microsserviços em evolução contínua.

REFERÊNCIAS

AMAZON WEB SERVICES. *What is an API?* 2024. Acesso em 18 de nov. de 2024. Disponível em: ⟨https://aws.amazon.com/pt/what-is/api/⟩.

BANGARE, S. L. et al. Automated api testing approach. *International Journal of Engineering Science and Technology*, Engg Journals Publications, v. 4, n. 2, p. 673–676, 2012.

BERNARDO, P. C. *Padrões de testes automatizados*. Dissertação (Mestrado) — Universidade de São Paulo, 2011.

BRASIL. Legislation, *Lei Geral de Poteção de Dados. n. 13.709, de 14 de agosto de 2018*. Brasília, DF, 2018. Acesso em: 13 fev. 2025. Disponível em: ⟨http://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/l13709.htm⟩.

CABRAL, B. M. B. *Instrumentação de Código na Plataforma .NET*. [S.l.]: Universidade de Coimbra, 2005.

COODESH. *O que é Cucumber?* 2025. Acesso em 30 jan. 2025. Disponível em: ⟨https://coodesh.com/blog/dicionario/o-que-e-cucumber/⟩.

CUCUMBER LTD. *Behavior-Driven Development (BDD)*. 2024. Acesso em 14 dez. 2024. Disponível em: ⟨https://cucumber.io/docs/bdd/⟩.

CUCUMBER LTD. *Cucumber Guides: Overview*. 2024. Acesso em 14 dez. 2024. Disponível em: ⟨https://cucumber.io/docs/guides/overview/⟩.

CYPRESS.IO. *Measure Code Coverage in Cypress*. [S.l.], 2024. Acesso em 22 nov. 2024. Disponível em: ⟨https://docs.cypress.io/app/tooling/code-coverage⟩.

CYPRESS.IO. *Why Cypress?* 2024. Acesso em: 21 nov. 2024. Disponível em: ⟨https://docs.cypress.io/app/get-started/why-cypress⟩.

DIGITAL.AI. *Automated Testing Tools*. 2024. Acesso em: 21 nov. 2024. Disponível em: ⟨https://digital.ai/pt/glossary/automated-testing-tools/⟩.

DOCKER INC. *Docker Compose*. 2024. Acesso em: 02 nov. 2024. Disponível em: ⟨https://docs.docker.com/compose⟩.

GHANI, I. et al. Microservice testing approaches: A systematic literature review. *International Journal of Integrated Engineering*, Universiti Tun Hussein Onn Malaysia Publisher's Office, v. 11, n. 8, p. 65–80, 2019. Disponível em: ⟨http://penerbit.uthm.edu.my/ojs/index.php/ijie⟩.

GIT. *Distributed Version Control System*. 2025. Acesso em: 13 fev. 2025. Disponível em: ⟨https://git-scm.com/⟩.

GRAHAM, D.; FEWSTER, M. *Software Test Automation: Effective Use of Text Execution Tools*. [S.l.]: Addison Wesley, 1999.

GRESPI, T. *Testes de API - Parte 1: Entendendo e botando a mão na massa com Postman*. 2020. Acesso em 18 nov. 2024. Disponível em: ⟨https://medium.com/@thiagogrespi/testes-de-api-parte-1-entendendo-e-botando-a-m%C3%A3o-na-massa-com-postman-b365923b83e1⟩.

ISTANBUL. *JavaScript Test Coverage Made Simple*. 2024. Acesso em 22 nov. 2024. Disponível em: ⟨https://istanbul.js.org/⟩.

IVANKOVIĆ, M. et al. Code coverage at google. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2019. p. 955–963.

JETBRAINS. *Code Coverage in CI/CD - Concepts*. 2024. Acesso em: 19 nov. 2024. Disponível em: ⟨https://www.jetbrains.com/pt-br/teamcity/ci-cd-guide/concepts/code-coverage/⟩.

LERCHER, A. et al. Microservice api evolution in practice: A study on strategies and challenges. *The Journal of Systems and Software*, Elsevier, v. 215, 2024. Disponível em: ⟨https://doi.org/10.1016/j.jss.2024.112110⟩.

LEWIS, J.; FOWLER, M. *Microservices: a definition of this new architectural term*. 2014. Disponível em: ⟨https://martinfowler.com/articles/microservices.html⟩.

MAAS, D. *Code Coverage: Entendendo a Métrica de Cobertura de Código*. 2022. Acesso em: 21 nov. 2024. Disponível em: ⟨https://blog.dyegomaas.com.br/posts/code-coverage/⟩.

MICROSOFT. *What is Monitoring?* 2024. Acesso em: 1 nov. 2024. Disponível em: ⟨https://learn.microsoft.com/pt-br/devops/operate/what-is-monitoring⟩.

MOHAMMAD, S. M. Improve software quality through practicing devops automation. *Sikender Mohsienuddin Mohammad,"IMPROVE SOFTWARE QUALITY THROUGH PRACTICING DEVOPS AUTOMATION", International Journal of Creative Research Thoughts (IJCRT), ISSN*, p. 2320–2882, 2018.

MONITORA TEC. *Ferramentas de teste de software: quando e por que automatizar?* 2021. Acesso em: 21 nov. 2024. Disponível em: ⟨https://www.monitoratec.com.br/blog/ferramentas-de-teste-de-software/⟩.

NETO, A. Introdução a teste de software. *Engenharia de Software Magazine*, v. 1, p. 22, 2007.

NODE.JS. *JavaScript runtime*. Acesso em 22 nov. 2024. Disponível em: ⟨https://nodejs.org/en/⟩.

OBJECTIVE SOLUTIONS. *Teste de Integração: entenda sua importância e benefícios*. 2024. Acesso em: 18 nov. 2024. Disponível em: ⟨https://www.objective.com.br/insights/teste-de-integracao/⟩.

OBJECTIVE SOLUTIONS. *Testes funcionais: saiba porque é fundamental para entrega de software de qualidades*. 2024. Acesso em: 30 jan. 2025. Disponível em: ⟨https://www.objective.com.br/insights/testes-funcionais/⟩.

PARAíBA. *Resolução CIB-PB nº 814/2023 - Programa de incentivo à causa animal*. 2023. Acesso em: 13 fev. 2025. Disponível em: ⟨https://paraiba.pb.gov.br/diretas/saude/arquivos-1/cib-2023/resolucao-cib-pb-no-814-2023-programa-de-incentivo-a-causa-animal.pdf⟩.

PATHAK, K. *Cypress with Cucumber for Seamless End-to-End Testing*. 2024. Acesso em 14 dez. 2024. Disponível em: ⟨https://testgrid.io/blog/cypress-with-cucumber/⟩.

PRESSMAN, R. S.; MAXIM, B. R. *Engenharia de software-9*. [S.l.]: McGraw Hill Brasil, 2021.

RED HAT. *O que é Docker?* 2024. Acesso em: 02 nov. 2024. Disponível em: ⟨https://www.redhat.com/pt-br/topics/containers/what-is-docker⟩.

RED HAT. *What Are Application Programming Interfaces?* 2024. Acesso em 18 nov. 2024. Disponível em: ⟨https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces⟩.

ROCHE, J. Adopting devops practices in quality assurance. *Communications of the ACM*, ACM New York, NY, USA, v. 56, n. 11, p. 38–43, 2013.

SCHWERING, R.; YEEN, J. *Four common types of code coverage*. 2023. Acesso em: 19 nov. 2024. Disponível em: ⟨https://web.dev/articles/ta-code-coverage⟩.

SMARTBEAR SOFTWARE. *Swagger - API Development Tools for Everyone*. 2024. Acesso em 05 dez. 2024. Disponível em: ⟨https://swagger.io/⟩.

SOMMERVILLE, I. *Engenharia de software*. [S.l.]: Pearson Prentice Hall, 2011.

SONI, M. End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery . In: *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*. Los Alamitos, CA, USA: IEEE Computer Society, 2015. p. 85–89. Disponível em: ⟨https://doi.ieeecomputersociety.org/10.1109/CCEM.2015.29⟩.

SVAHNBERG, M. *Supporting Software Architecture Evolution: Architecture Selection and Variability*. Tese (Doctoral Dissertation) — Blekinge Institute of Technology, 2003.

VALENTE, M. T. Engenharia de software moderna. *Princípios e Práticas para Desenvolvimento de Software com Produtividade*, v. 1, n. 24, 2020.

VIEIRA, R. *Um pouco sobre cobertura de código e cobertura de testes*. 2020. Acesso em: 19 nov. 2024. Disponível em: ⟨https://medium.com/liferay-engineering-brazil/um-pouco-sobre-cobertura-de-c%C3%B3digo-e-cobertura-de-testes-4fd062e91007⟩.

WALLS, D. *Code Coverage in 2 Minutes with NYC*. 2020. Acesso em 22 nov. 2024. Disponível em: ⟨https://dev.to/danywalls/code-coverage-in-2-minutes-with-nyc-130m⟩.

WASEEM, M. et al. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *The Journal of Systems and Software*, Elsevier, v. 182, 2021. Disponível em: ⟨https://doi.org/10.1016/j.jss.2021.111061⟩.