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

JOYCE LIMA AVELINO

ENTENDENDO FALHAS DE TESTES E2E NO CONTEXTO DE EVOLUÇÃO: ESTUDO DE CASO NO SISTEMA REGPET

CAMPINA GRANDE
2025

JOYCE LIMA AVELINO

ENTENDENDO FALHAS DE TESTES E2E NO CONTEXTO DE EVOLUÇÃO: 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 Bacharela em Computação.

Orientador: Prof. Dra Sabrina de Figueiredo Souto.

CAMPINA GRANDE
2025

A949e Avelino, Joyce Lima.
Entendendo falhas de testes E2E no contexto de evolução [manuscrito] : estudo de caso no sistema RegPet / Joyce Lima Avelino. - 2025.
57 f. : il. color.

Digitado.

Trabalho de Conclusão de Curso (Graduação em Ciência da computação) - Universidade Estadual da Paraíba, Centro de Ciências e Tecnologia, 2025.

"Orientação : Prof. Dra. Sabrina de Figueirêdo Souto, Departamento de Computação - CCT".

1. Falhas de testes. 2. Testes automatizados. 3. Evolução de Software. I. Título

21. ed. CDD 005.3

Elaborada por Pfeyffemberg de Moura Guimarães - CRB - 15/1020 BC

JOYCE LIMA AVELINO

ENTENDENDO FALHAS DE TESTES E2E NO CONTEXTO DE EVOLUÇÃO: 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 Bacharela em Computação

Aprovada em: 11/06/2025.

BANCA EXAMINADORA

Documento assinado eletronicamente por:

- **Sabrina de Figueirêdo Souto** (***.047.964-**), em **16/06/2025 10:14:53** com chave **e43ac4d64ab311f0aac41a7cc27eb1f9**.
- **Ana Isabella Muniz Leite** (***.834.864-**), em **16/06/2025 10:37:33** com chave **0f0120b84ab711f081631a7cc27eb1f9**.
- **Adelito Borba Farias** (***.520.684-**), em **16/06/2025 10:25:44** com chave **67eb55744ab511f0b5441a7cc27eb1f9**.

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: 18/06/2025
Código de Autenticação: 362797

Aos meus pais, pelo amor e força nessa jornada.

AGRADECIMENTOS

Muitas pessoas foram importantes para mim ao longo da graduação, e sou imensamente grata a todas elas. Em especial, gostaria de agradecer a:

À minha mãe, por sempre estar ao meu lado, me fortalecendo e apoiando em todos os momentos. Obrigada por todo o amor, cuidado e por sempre acreditar em mim.

Ao meu pai, pelo amor, incentivo e apoio durante todos esses anos de estudo.

A Edilson, que foi um presente que a graduação me deu. Obrigada pelo amor, carinho e suporte ao longo dessa jornada.

Aos meus colegas de turma e de trabalho do Nutes, por sempre me ajudar diante das dificuldades encontradas durante o curso.

Por fim, aos professores de computação, por todo o conhecimento compartilhado. Em especial, à professora Sabrina Souto, pelas oportunidades, paciência e por ter sido uma pessoa essencial no meu desenvolvimento como profissional.

"Conhecimento não é aquilo que você sabe, mas o que você faz com aquilo que você sabe."(Aldous Huxley)

RESUMO

Atualmente, as aplicações web possuem interfaces gráficas sofisticadas, que visam proporcionar uma boa experiência ao usuário, sendo a interação do usuário mediada principalmente pela interface gráfica, tornando essencial garantir seu funcionamento adequado e consistente. A aplicação de técnicas de teste de software, como os testes automatizados, é fundamental para identificar e reduzir falhas no desenvolvimento dessas aplicações. Entre os tipos de testes, destacam-se os testes automatizados End-to-End (E2E), que simulam o fluxo completo de uso de uma aplicação, desde a interface até o back-end. No entanto, a automação de testes para validação da interface gráfica (GUI) apresenta desafios, como a fragilidade dos casos de teste, que podem falhar devido a pequenas alterações na interface, mesmo sem modificações nas funcionalidades do sistema. Com isso, este trabalho tem como objetivo investigar as causas das falhas dos testes automatizados E2E, no contexto de evolução de software, através de um estudo de caso do sistema RegPet. Por fim, como resultado, foram vistas as falhas mais comuns encontradas e como nem sempre testes instáveis representam regressão do sistema.

Palavras-chave: falhas de testes; testes automatizados; evolução de software; testes E2E.

ABSTRACT

Currently, web applications feature sophisticated graphical user interfaces (GUIs) aimed at providing a high-quality user experience. As user interaction is primarily mediated through the GUI, ensuring its correct and consistent functioning becomes essential. The application of software testing techniques, such as automated testing, is fundamental to identifying and mitigating failures during the development of such systems. Among the various testing approaches, End-to-End (E2E) automated tests stand out, as they simulate the complete usage flow of an application, from the interface to the back-end. However, GUI test automation presents several challenges, particularly the fragility of test cases, which may fail due to minor interface changes, even when no functional alterations have occurred in the system. In this context, the present study aims to investigate the causes of E2E automated test failures in the context of software evolution, through a case study on the RegPet system. As a result, the most common failure patterns were identified, revealing that unstable tests do not always indicate actual regressions in the system.

Keywords: test failures; automated tests; software evolution; E2E tests.

LISTA DE ILUSTRAÇÕES

Figura 1 - Visão geral da abordagem 20
Figura 2 - Geração das massas de dados e execução dos testes 22
Figura 3 - Geração das massas de dados e execução dos testes 22

Figura 4 - Interface do administrador 26
Figura 5 - Interface do executante estadual 27
Figura 6 - Interface do gestor municipal 27
Figura 7 - Fluxograma de uma solicitação de procedimento 28
Figura 8 - Processo de testes 29
Figura 9 - Mapa mental do protetor 30
Figura 10 - Cenário de teste 31
Figura 11 - Arquivo com os indentificadores 31
Figura 12 - Arquivo `.spec` 32
Figura 13 - Modo interativo do Cypress 33
Figura 14 - Modo interativo do Cypress 34
Figura 15 - Bug no Bugzilla 36
Figura 16 - Lista de bugs no Bugzilla 36

Figura 17 - Variáveis de ambiente 38
Figura 18 - Variáveis de ambiente 38
Figura 19 - Massas de dados do administrador 40
Figura 20 - Resultados do administrador 41

Figura 21 - Visão geral dos resultados dos testes 42
Figura 22 - Categorização das falhas 43
Figura 23 - Categorias das falhas por versão 45
Figura 24 - Logs de erro 46
Figura 25 - Logs de erro por versão 47
Figura 26 - Relação entre as categorias de falhas e os logs de erro - V1.1 49
Figura 27 - Relação entre as categorias de falhas e os logs de erro - V1.2 50
Figura 28 - Relação entre as categorias de falhas e os logs de erro - V1.3 51
Figura 29 - Resultado dos testes em todas as versões 52

SUMÁRIO

Página

1 INTRODUÇÃO 11
1.1 Problemática 12
1.2 Objetivos gerais e específicos 13
1.3 Estrutura do trabalho 13

2 REFERENCIAL TEÓRICO 14
2.1 Teste de Software 14
2.1.1 Casos de testes 14
2.1.2 Erro, defeito e falha 14
2.1.3 Verificação e validação 15
2.2 Automação de testes 15
2.2.1 Cypress 15
2.2.2 Cucumber 16
2.3 Testes E2E 16
2.3.1 Fragilidade nos testes de GUI 17
2.4 Evolução e manutenção de testes 17
2.4.1 Testes de regressão 18
2.5 Ferramentas utilizadas no ambiente de testes automatizados . . . 18

3 METODOLOGIA 20
3.1 Configuração do ambiente de testes 20
3.2 Implantação local e versionamento 21
3.3 Geração das massas de dados e execução dos testes 21
3.4 Relatórios dos testes 22

4 O SISTEMA REGPET 24
4.1 O RegPet 24
4.2 Principais funcionalidades 24
4.3 Fronted do RegPet 25
4.3.1 Organização dos usuários 26
4.3.2 Funcionamento do sistema 28
4.4 Testes de E2E no RegPet 28
4.4.1 Planejamento dos testes 29
4.4.2 Definição dos testes 30
4.4.3 Execução dos testes 32
4.4.4 Análise dos resultados 33

4.4.5 Gerenciamento de bugs 34

5 ESTUDO DE CASO: ANÁLISE DE FALHAS EM TESTES E2E NA EVOLUÇÃO DO REGPET 37
5.1 Configuração do ambiente de testes 37
5.2 Implantação local e versionamento 38
5.3 Geração das massas de dados e execução dos testes 39
5.4 Relatório dos testes 40

6 RESULTADOS 42
6.1 Objetivo 1: Mapear as causas das falhas em cada versão 43
6.1.1 Quais são as causas das falhas nos testes? 43
6.1.2 Qual a distribuição das causas das falhas por versão? 45
6.2 Objetivo 2: Mapear as falhas em cada versão 45
6.2.1 Quais são os tipos das falhas nos testes? 46
6.2.2 Qual a distribuição dos logs de erro por versão? 47
6.3 Objetivo 3: Analisar a relação entre causa e falha entre as versões 48
6.3.1 Qual a relação entre as categorias de falhas e os logs de erro? 48
6.4 Objetivo 4: Analisar regressão entre as versões 52
6.4.1 Qual a quantidade de testes que passavam na versão anterior e começaram a falhar na próxima? 52
6.5 Discussões 53

7 CONCLUSÃO 55

REFERÊNCIAS 55

1 INTRODUÇÃO

Atualmente, as Aplicações Web não só possuem robustas estruturas arquiteturais, como também modernas interfaces gráficas com diferentes tipos de funcionalidades que buscam priorizar a boa experiência do usuário.

Dado que grande parte da interação do usuário com o sistema é mediada pela interface gráfica (TSAI et al., 2006), assegurar que ela funcione de forma adequada e consistente é fundamental para garantir a qualidade do sistema como um todo. Assim como outras partes da aplicação, a interface gráfica também está suscetível a erros e por sua natureza, tais erros podem ser simplesmente percebidos pelo usuário final (OLIVEIRA; DELAMARO; NUNES, 2009).

Nesse contexto, é importante destacar que é inevitável que haja mudanças na aplicação, uma vez que os requisitos estão em constante mudança e desenvolvimento, principalmente em cenários de uso da metodologia ágil, onde o ambiente tecnológico e as expectativas dos usuários evoluem rapidamente. A mudança é uma característica inerente aos sistemas de software, impulsionada por fatores como transformações nos negócios, surgimento de novas tecnologias e alteração nas prioridades dos clientes (SOMMERVILLE, 2011). Com isso, os sistemas não podem ser avaliados unicamente pela sua versão inicial, mas sim pela sua capacidade de evoluir e se adequar às necessidades do cliente e da equipe de desenvolvimento.

A ausência de uma evolução adequada do sistema pode gerar prejuízos significativos para os clientes, uma vez que os custos relacionados à manutenção e à adaptação do software correspondem a uma parte considerável do orçamento total do projeto (SOMMERVILLE, 2011). Nesse sentido, a utilização de técnicas de testes de Software pode ser benéfica para reduzir o número de falhas e erros encontrados no desenvolvimento das aplicações (OLIVEIRA; DELAMARO; NUNES, 2009).

Uma das técnicas de testes de Software, são os testes automatizados, eles são utilizados para validar o comportamento de diferentes funcionalidades do sistema de forma eficiente e repetitiva. Dentre os tipos de testes automatizados, os testes End-to-End (E2E) se destacam por simularem o fluxo completo de uso da aplicação, verificando a integração entre todos os componentes do sistema, desde a interface gráfica até as interações com o back-end (ALIAN et al., 2024)). Apesar de ser possível realizar testes E2E manualmente, a automação de testes é uma escolha preferível, dado que com ela é possível criar suítes de testes mais bem estruturadas e confiáveis, criando conjuntos de scripts repetíveis para cobrir todas as funcionalidades de uma determinada GUI (COPPOLA; MORISIO; TORCHIANO, 2018).

1.1 Problemática

Uma vez que os requisitos podem, rapidamente, mudar em sistemas que são desenvolvidos com o uso da metodologia ágil, a interface gráfica também pode ser diretamente afetada com essas modificações. Mudanças na posição dos elementos, remoção de páginas, adição de novas funcionalidades são alguns dos motivos pelo o qual os testes automatizados com foco em validação do frontend da aplicação são suscetíveis a constante manutenção e falhas.

Estudos da literatura mostram que o processo de automação de testes para a GUI contém diversas armadilhas, podendo ser tão difíceis quanto outras formas tradicionais de realizar testes (PRADHAN, 2012). Dado que os casos de teste de GUI são particularmente frágeis, ou seja, podem falhar mesmo que pequenas alterações sejam realizadas na interface, sem modificações nas funcionalidades do sistema (COPPOLA; MORISIO; TORCHIANO, 2018).

Isso faz com que mais manutenções precisem ser realizadas nos casos teste, gerando mais custos para o projeto e retrabalho. Sendo uma das atividades mais caras no processo de testar um software, o de manutenção, ou seja, retestar a aplicação depois dela ter sido modificada. Esse processo é conhecido como teste de regressão e ela pode consumir até 50% do orçamento de manutenção de um software (HARROLD; ORSO, 2008).

O sistema abordado no trabalho é o RegPet(RegPet, 2025), que é uma aplicação web voltada para a regulação de procedimentos da causa animal. Ele é um sistema que está em constante evolução, por conta disso, as mudanças na interface são frequentemente motivadas por ajustes nas funcionalidades e por feedbacks dos usuários.Tais alterações, ainda que pequenas, como a renomeação de um botão ou a reorganização de um componente visual, frequentemente resultam na quebra de testes automatizados E2E, mesmo que a funcionalidade principal permaneça inalterada.

Essa fragilidade dos testes compromete não apenas a eficiência da automação, mas também a confiabilidade do processo de verificação contínua, uma vez que falhas recorrentes nos testes podem gerar falsos positivos, aumentar o esforço de manutenção e desviar a atenção da equipe de problemas realmente críticos.A consequência é o acúmulo de testes quebrados, desativados ou ignorados, o que compromete a cobertura de testes e abre brechas para regressões em produção.

Diante do exposto, é importante compreender os motivos que tornam os testes voltados à validação do frontend da aplicação tão suscetíveis a falhas em contextos de evolução contínua. A análise dessas falhas e suas causas durante a evolução do sistema é fundamental para identificar padrões recorrentes de quebra, propor boas práticas de desenvolvimento e manutenção de testes automatizados, além de orientar melhorias no processo de validação de interface.

1.2 Objetivos gerais e específicos

Dada a problematização, o presente trabalho tem como objetivo geral compreender quais são as falhas de testes mais comuns em testes automatizados E2E no contexto de evolução contínua do sistema RegPet, que podem ser catalisadores para que mais manutenções ocorram nos casos de testes. Nesse contexto, o trabalho tem como objetivos específicos:

1. Mapear as causas das falhas em cada versão do sistema;
2. Mapear as falhas em cada versão do sistema;
3. Analisar a relação entre causa e falha entre as versões;
4. Analisar regressões entre as versões.

1.3 Estrutura do trabalho

O trabalho foi dividido em 7 capítulos, em que o primeiro capítulo é a introdução, com a apresentação do tema, problemática e objetivos do trabalho. No segundo capítulo foi descrita a fundamentação teórica, com os principais conceitos necessários para entendimento e embasamento da pesquisa. O capítulo 3 mostra a metodologia utilizada, com a descrição de todos os passos necessários para atingir os objetivos do trabalho.

No quarto capítulo está a apresentação do sistema utilizado no estudo de caso, bem como todas suas características e funcionalidades. Já o capítulo 5, fala sobre a aplicação da metodologia no sistema usado para o estudo de caso, descrevendo como cada passo da metodologia foi aplicado à aplicação. No capítulo 6 ficaram todos os resultados alcançados com a aplicação da metodologia, assim como as discussões.Por fim, no capítulo 7 está a conclusão do trabalho, em que é retomado o contexto da pesquisa e apresentado reflexões sobre os resultados obtidos e possíveis trabalhos futuros.

2 REFERENCIAL TEÓRICO

Neste capítulo são apresentados os conceitos mais relevantes e necessários para o desenvolvimento deste trabalho, sendo apresentado conceitos sobre testes de software, automação de testes e ferramentas utilizadas.

2.1 Teste de Software

O desenvolvimento de um Software envolve várias etapas, e uma delas é a realização de testes. Esse é um processo que visa garantir se os requisitos estabelecidos foram cumpridos e implementados adequadamente no sistema, seu objetivo é mostrar falhas em um produto, de forma que as causas dessas falhas sejam identificadas e possam ser corrigidas pela equipe de desenvolvimento antes da entrega final (NETO; DIAS, 2007).

Um teste pode ser visto como "[...] o processo de operar um sistema ou componente sob determinadas condições, observando ou registrando resultados, realizando a avaliação de algum aspecto do sistema ou componente" (IEEE, 1990). Com isso, o propósito dos testes podem se diferenciar conforme o contexto em que são aplicados, levando em conta aspectos como o artefato que está sendo testado, o estágio do teste, os riscos envolvidos, o modelo de ciclo de vida de desenvolvimento de software adotado, além de fatores relacionados ao ambiente de negócios, como a organização da empresa, a concorrência no mercado e o prazo para lançamento do produto (BSTQB, 2023).

2.1.1 Casos de testes

Um caso de teste pode ser compreendido como um artefato essencial do processo de verificação de software, contendo informações como um identificador único, objetivo do teste, pré-condições, entradas, saídas esperadas e pós-condições. Esse conjunto de dados permite avaliar se a funcionalidade do sistema está em conformidade com os requisitos especificados (JORGENSEN, 2013).

A execução de um caso de teste envolve a preparação do ambiente com as condições necessárias, aplicação das entradas, observação e comparação das saídas com os resultados esperados e a validação das pós-condições. Assim, os testes assumem um papel tão importante quanto o código-fonte e devem ser cuidadosamente desenvolvidos, revisados e mantidos.

2.1.2 Erro, defeito e falha

Conceitos importantes de se destacar ao tratar sobre testes de Software é a diferença entre erro, defeito e falha.

O erro é "uma ação humana que resulta em um defeito" (BSTQB, 2023). Ele pode ser causado por vários fatores, sejam emocionais, físicos ou por natureza externa.

Já o defeito, é visto como “uma imperfeição ou deficiência em um produto que não atende aos seus requisitos, especificações ou prejudica seu uso pretendido” (BSTQB, 2023). Além disso, alguns defeitos podem nunca causar algum tipo de falha, enquanto outros sempre que forem executados terão um resultado negativo, ou seja, resultarão em falhas.

Por fim, a falha pode ser entendida como “um evento no qual um componente ou sistema não atende aos seus requisitos dentro dos limites especificados durante sua execução” (BSTQB, 2023), sendo ela, a manifestação do defeito.

Em suma, os humanos cometem erros que geram defeitos no sistema e que podem ser visualizados através das falhas, que são consequências dessas anomalias.

2.1.3 Verificação e validação

A verificação é o processo de avaliar se uma implementação está em conformidade com sua especificação. De forma geral, trata-se de analisar se o sistema ou os requisitos definidos estão alinhados com as necessidades reais (PEZZè; YOUNG, 2008).

Já a validação avalia em que grau um sistema atende aos requisitos, no sentido de satisfazer as necessidades dos usuários, sendo atender os requisitos diferente de estar conforme uma especificação de requisito (PEZZè; YOUNG, 2008).

2.2 Automação de testes

A automação de testes é uma técnica que utiliza ferramentas que automatizam o processo de verificação parcial ou completa de um software, para fazer a validação de um determinado sistema (IEEE, 1990). Essa é uma atividade de grande importância devido à sua capacidade de aumentar a eficiência operacional, já que permite executar os testes de forma mais rápida, com isso, ela pode trazer maior precisão aos resultados, ampliando a cobertura de testes, especialmente em cenários mais complexos (ALI; HAMZA; RASHID, 2023).Além disso, outra vantagem é a otimização dos testes de regressão, possibilitando a execução simultânea de múltiplos testes.

No mercado, há várias ferramentas e frameworks, cada uma com suas próprias especificidades que se adequam para cada tipo de projeto. Assim é preciso fazer uma boa avaliação para escolher uma ferramenta adequada para o tipo de aplicação desenvolvida. Os critérios de avaliação para a escolha de uma boa ferramenta de automatização de testes, são, o grau de complexidade, custo da ferramenta, aprendizagem da aplicação e reutilização do código automatizado do teste (SILVA; DALLILO, 2019).

2.2.1 Cypress

O Cypress é uma ferramenta de testes frontend, com foco em aplicações web (Cypress.io, 2025). Ele é muito utilizado para automatizar testes E2E e validar a interface

de usuário (UI) em navegadores, já que ele atua diretamente no navegador e no mesmo ciclo de execução da aplicação.

A ferramenta possibilita a execução de fluxos completos de testes, com acompanhamento em tempo real por meio da visualização interativa dos passos executados, além de oferecer suporte à geração automática de capturas de tela e vídeos, recursos que são especialmente úteis durante a depuração de falhas.

Além disso, um diferencial do Cypress é sua sintaxe simples e intuitiva, com suporte às linguagens JavaScript e TypeScript, o que facilita sua adoção por desenvolvedores frontend e promove maior integração entre as equipes de desenvolvimento e testes.

2.2.2 Cucumber

O Cucumber é uma ferramenta de suporte à automação de testes baseada no conceito de BDD (Behavior-Driven Development). Seu principal objetivo é promover uma colaboração mais eficaz entre testadores e desenvolvedores por meio de uma linguagem acessível e simples a todos os envolvidos no processo de desenvolvimento do sistema.

Uma das principais características do Cucumber é o uso da linguagem Gherkin, que permite descrever o comportamento do sistema em um formato estruturado e compreensível, por meio de sentenças como “Dado que...”, “Quando...” e “Então...”. Esse formato facilita a comunicação entre áreas técnicas e não técnicas, garantindo que todos tenham a mesma compreensão dos requisitos (Cucumber Open Source Project, 2025).

Após a escrita dos testes em Gherkin, cada passo do cenário é mapeado para métodos implementados em linguagens de programação, como JavaScript. Esses métodos são responsáveis por executar as ações correspondentes aos passos descritos, com isso, os cenários comportamentais passam a desempenhar um duplo papel, funcionam simultaneamente como especificações do sistema e como testes automatizados de validação.

2.3 Testes E2E

Os testes de ponta a ponta avaliam se vários componentes integrados em um sistema funcionam juntos e corretamente desde a interface do usuário (IU) até o back-end, simulando interações reais do usuário e verificando a funcionalidade do aplicativo. No teste E2E, a aplicação é testada como um todo, tomando como base a perspectiva do usuário final.

Para o propósito do trabalho, o foco será nos testes E2E voltados para a validação da interface do usuário. A GUI é a forma mais convencional de interação do usuário com o sistema, sendo dedicada grande parte do desenvolvimento para tal. Com isso, não se deve negligenciar os testes de interface, eles são usados para validar os elementos da interface gráfica, testar os fluxos funcionais que interagem com esses elementos e conferir se os dados processados no backend são exibidos nas páginas iniciais (PRADHAN, 2012).

2.3.1 Fragilidade nos testes de GUI

Realizar testes da GUI não é uma tarefa simples e ela encontra diversos desafios no caminho, uma delas é a fragilidade dos testes. Com isso, um caso de teste da GUI pode ser visto como frágil quando ele:

1. Precisa de modificação quando a aplicação se desenvolve (COPPOLA; MORISIO; TORCHIANO, 2018);

2. Tal necessidade não se deve a uma modificação das funcionalidades testadas, mas a mudanças no arranjo ou definição da interface (COPPOLA; MORISIO; TORCHIANO, 2018).

Algumas vezes, as entradas e saídas de um caso de teste são gráficas e podem depender da posição do objeto exibido na tela. Os dados podem estar lá na GUI, mas os testes podem não encontrá-los, pois não são exibidos na posição especificada. Além disso, alterações na posição de botões, menus e outros layouts podem falhar nos casos de teste já gerados (PRADHAN, 2012).

Se uma fragilidade é identificada em um caso de teste, isso pode causar uma falha no teste na próxima versão da aplicação. Isso faz com que o testador ou desenvolvedor precise dedicar um esforço adicional para garantir que não houve regressão nas funcionalidades e, se necessário, adaptar o teste às mudanças (COPPOLA; MORISIO; TORCHIANO, 2018). Entretanto, a geração e execução de casos de teste de GUI geralmente consomem muito tempo e recursos, o que pode não estar sempre prontamente disponível, tornando esse processo um desafio.

2.4 Evolução e manutenção de testes

A manutenção de testes é o processo de mudar o sistema depois de ele já estar em uso, seja pela adição de uma nova funcionalidade ou correção de bug (SOMMERVILLE, 2011). Essa é uma etapa constante no processo de desenvolvimento de software, e à medida que o software evolui, os testes automatizados também precisam acompanhar essas modificações para continuar sendo eficazes.

Em especial, os testes voltados à validação da interface gráfica já que são altamente suscetíveis a falhas causadas por alterações visuais, mesmo quando não há mudanças significativas na lógica de negócio. Isso torna a manutenção dos testes uma das etapas mais trabalhosas e custosas do processo de verificação de software, podendo consumir uma parcela significativa do orçamento destinado à qualidade (HARROLD; ORSO, 2008).

A distinção entre desenvolvimento e manutenção vem se tornando cada vez menos relevante, uma vez que o software está em constante evolução (SOMMERVILLE, 2011). Nesse sentido, pensar nos testes como entidades também sujeitas à evolução contínua é fundamental para garantir que o sistema permaneça confiável ao longo do tempo.

2.4.1 Testes de regressão

Esses testes são executados após modificações no código, como a implementação de novas funcionalidades ou a correção de bugs, com o objetivo de assegurar que as mudanças não causem novos problemas em partes do sistema que já operavam corretamente (RIOS; FILHO, 2013). Ao tratar sobre a validação da UI, os testes de regressão tem um importante papel, eles são aplicados para fazer a verificação de novas modificações e se elas introduziram algum tipo de bug no sistema. Por serem significativamente afetados pela mudança regular no layout, o teste de GUI consome recursos e tempo significativos, dificultando os testes de regressão (PRADHAN, 2012).

É possível identificar dois tipos distintos de teste de regressão, classificados com base na modificação ou não da especificação do sistema: regressão progressiva e regressão corretiva (LEUNG; WHITE, 1989). O teste de regressão progressivo ocorre quando há modificações na especificação do software, geralmente devido à adição de novos requisitos ou funcionalidades. Essas mudanças demandam uma atualização do plano de testes para refletir os novos comportamentos esperados do sistema, tornando necessário validar o programa modificado em relação à nova especificação.

Por outro lado, o teste de regressão corretivo é empregado quando a especificação permanece inalterada, mas há mudanças pontuais no código ou nas decisões de design com o objetivo de corrigir falhas existentes. Nesse cenário, grande parte dos testes antigos ainda são válidos, pois os requisitos não mudaram. No entanto, como as estruturas de controle ou o fluxo de dados do programa podem ter sido alterados, alguns testes anteriores podem deixar de cobrir adequadamente as funcionalidades impactadas pelas correções (LEUNG; WHITE, 1989).

2.5 Ferramentas utilizadas no ambiente de testes automatizados

Para configurar o ambiente de testes, assim como executar os testes do sistema usado no estudo de caso, foram utilizadas algumas ferramentas, sendo elas:

- NodeJs e JavaScript: O NodeJs é um ambiente de execução da linguagem de programação JavaScript de código aberto e multiplataforma, muito utilizado em aplicações web (Node.js Contributors, 2025).

- Git: O Git é um sistema de controle de versão distribuído, criado com o objetivo de registrar as mudanças realizadas no código-fonte de um projeto ao longo do tempo. Ele permite que desenvolvedores acompanhem o histórico de alterações, colaborem em equipe e revertam modificações quando necessário (Git, 2025).

- Docker: O Docker é uma plataforma projetada para simplificar a criação, execução e distribuição de aplicações por meio da utilização de containers, que são unidades portáteis e padronizadas que reúnem tudo o que uma aplicação precisa para

funcionar, incluindo o código, bibliotecas, dependências e configurações (Docker, 2025).

- Docker Compose: Já o Docker Compose é uma ferramenta para definir e executar aplicações multi-contêineres, facilitando o gerenciamento de aplicações complexas e promovendo maior reprodutibilidade nos ambientes de desenvolvimento, teste e produção (Docker, 2025).

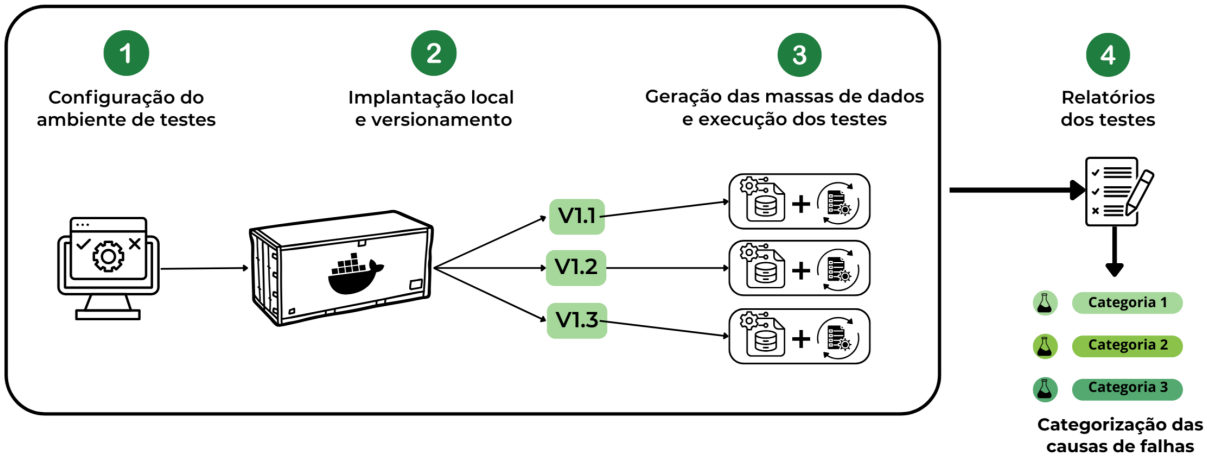
3 METODOLOGIA

Este capítulo tem como propósito apresentar a metodologia que foi adotada para gerar os relatórios de falha dos testes de acordo com cada versão do sistema, bem como a categorização de cada causa de falha. Para atingir esse objetivo, foi necessário seguir algumas etapas, sendo elas: preparar o ambiente de testes, implantar o sistema localmente, executar os testes e gerar os relatórios para análise.

A figura 1 abaixo representa uma visão geral desse processo adotado, o qual foi dividido em 4 etapas principais:

1. Configuração do ambiente de testes para ser possível executar as suítes de testes;
2. Implantação do sistema localmente através da conteinerização para versionar a interface da aplicação e os testes;
3. Geração das massas de dados que vão ser consumidas na execução dos testes;
4. Análise dos relatórios e criação das categorias de causa de falhas dos testes.

Figura 1 – Visão geral da abordagem



Fonte: Elaborado pela autora, 2025.

3.1 Configuração do ambiente de testes

Nesta etapa, são realizadas as configurações para que seja possível executar as suítes de testes, incluindo a instalação das bibliotecas e dependências necessárias. Com este processo, uma infraestrutura que simula o real ambiente de uso da aplicação é gerada, sendo possível identificar falhas e inconsistências tanto visuais, quanto funcionais na interface, além disso, é gerado evidências visuais que facilitam o processo de geração dos relatórios.

3.2 Implantação local e versionamento

Nessa fase, o ambiente de testes é executado localmente por meio do processo de conteinerização e a sua versão específica é empacotada e inicializada, sendo a versão adequada do repositório de testes também preparada. Essa etapa é essencial, pois garante a construção da versão do Frontend necessária para a realização dos testes, sendo o alinhamento entre o ambiente de testes e o repositório de testes fundamental para garantir a integridade da análise.

Para realizar essa etapa, as versões dos microsserviços bem como das APIs, com os quais o frontend se comunica, precisam ser atualizadas com o auxílio de uma ferramenta de controle de versionamento. E, após essa atualização, uma ferramenta de conteinerização é utilizada para levantar a imagem da aplicação frontend, garantindo que ela seja executada em um ambiente compatível com os serviços de backend.

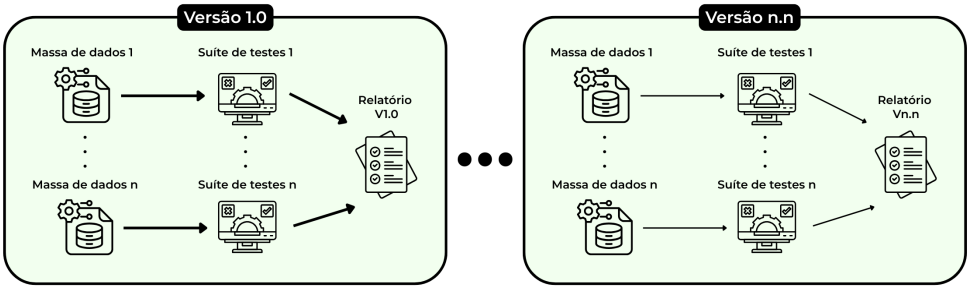
3.3 Geração das massas de dados e execução dos testes

Nesta etapa, são criadas as massas de dados necessárias para a execução dos testes, elas simulam dados que estariam presentes em cenários reais de uso da aplicação. A geração da massa de dados para os testes de Frontend é usada para compor a validação das funcionalidades da interface, já que ela contempla diferentes situações de uso. Somado a isso, esses dados permitem a interação com a interface como um usuário real, cadastrando informações, filtrando dados, entre outras funcionalidades.

Com as massas de dados definidas e preparadas, somado a realização dos passos anteriormente definidos, os testes se encontram prontos para serem executados de forma automatizada, utilizando os scripts previamente desenvolvidos. Dessa forma, cada teste é executado individualmente, sendo coletado e armazenado alguns dados para uma posterior verificação.

Para analisar o comportamento dos testes conforme a evolução do sistema, o processo observado na figura 2 abaixo foi seguido. Nele, uma massa de dados foi gerada para cada suíte de testes. Em seguida, as suítes foram executadas e então as informações sobre cada teste foram coletadas para compor o relatório, e, esse processo foi repetido para cada versão do ambiente de testes.

Figura 2 – Geração das massas de dados e execução dos testes



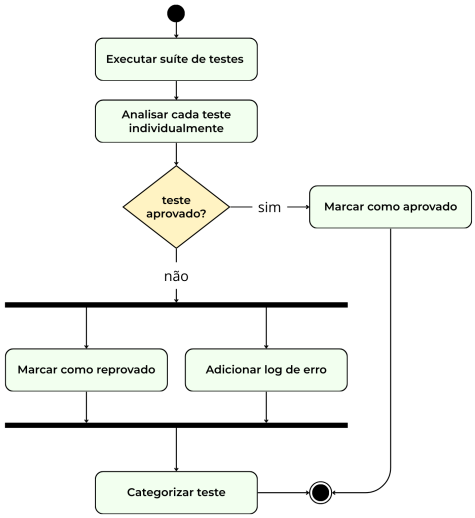
Fonte: Elaborado pela autora, 2025.

3.4 Relatórios dos testes

Por fim, após a execução dos testes, as informações de cada teste foram organizadas e armazenadas em uma planilha para serem analisadas Dentre essas informações, foram separados dados como o cenário de teste, sua versão correspondente, seu log de erro e resultado (aprovado ou reprovado).

Em seguida, por meio dos relatórios e análise de cada teste individualmente, foram criadas categorias das causas de falhas mais comuns encontradas ao executar os testes. Essas categorias são fundamentais para a validação do comportamento dos testes, dado que permitem uma visão clara da evolução da aplicação ao longo das versões, assim como das falhas mais recorrentes e seus motivos.

Figura 3 – Geração das massas de dados e execução dos testes



Fonte: Elaborado pelo autor, 2025.

A figura 3 representa esse processo, no qual, após a análise individual de cada teste, verifica-se se ele foi aprovado ou reprovado. Caso tenha sido reprovado, seu log de erro e seu resultado são extraídos, caso tenha sido aprovado, essa informação também é somada

a planilha. Com isso, esses dados são adicionados à planilha e, então, cada teste falho é categorizado em uma determinada categoria de causa de falha.

Finalmente, com o relatório dos testes de cada versão, contendo seus resultados, logs de erro e categoria das causas de falhas já pronto, outras planilhas com os resultados gerais de todas as versões analisadas são criadas. Essas planilhas são derivadas dos resultados obtidos a partir do relatório dos testes de cada versão, sendo elas as seguintes: tipo das falhas, categorias das causas de falhas, relação entre a falha e causa, análise de regressão entre as versões e análise geral das versões.

4 O SISTEMA REGPET

Este capítulo tem como foco apresentar as principais características do projeto RegRet, sistema utilizado para aplicar a metodologia previamente abordada. A princípio, será exibido as principais funcionalidades do sistema, bem como seu propósito. Em seguida, será tratado sobre os testes de Frontend, seu processo de criação e suas características.

4.1 O RegPet

O RegPet é uma aplicação web desenvolvida em uma parceria entre o Governo do Estado da Paraíba, a Gerência Operacional de Políticas da Causa Animal e o Núcleo de Tecnologias Estratégicas em Saúde (NUTES), sendo ele, uma ferramenta de regulação de procedimentos da causa animal.

A causa animal está relacionada com um conjunto de atividades que visam o bem estar, proteção e cuidado animal. Essa iniciativa busca assegurar que os animais sejam tratados de maneira ética e digna, diminuindo práticas como maus-tratos, abandono e negligência.

O sistema tem como objetivo incentivar os municípios paraibanos na realização do controle populacional de cães e gatos, além de promover a saúde e o bem-estar animal por meio da realização de procedimentos cirúrgicos, em especial a castração.

4.2 Principais funcionalidades

As principais funcionalidades encontradas no sistema, são as seguintes:

- Login e acesso: O sistema disponibiliza um mecanismo de autenticação de usuários, permitindo o redirecionamento para uma interface personalizada conforme o perfil de cada usuário, assegurando que o acesso às funcionalidades seja direcionado a cada tipo de perfil, dependendo do seu tipo de permissão.

- Gerenciamento de usuários: O administrador do sistema possui autonomia para cadastrar novos usuários, atualizar dados existentes, modificar níveis de permissão e remover usuários, garantindo o controle sobre o acesso e a gestão da base de usuários da plataforma.

- Dashboard: Dashboard com estatísticas e dados relacionados a quantidade de procedimentos realizados, espécies atendidas, municípios alcançados, etc.

- Filtragem: O sistema permite realizar a filtragem de várias informações que vão se adaptando a depender do tipo de usuário.

- Geração de Relatórios: Alguns usuários conseguem gerar relatórios personalizados sobre os procedimentos, assim como PDFs para impressão.

- Criação de solicitações de procedimento: É possível criar solicitações de procedimentos, adicionando dados dos animais, informações sobre sua saúde e o procedimento que deseja ser realizado.

- Validação das solicitações: Após a análise da solicitação criada, verificando e validando os dados inseridos, é possível validar ou invalidar a solicitação dependendo do resultado dessa verificação.

- Reservar procedimento: Com uma solicitação válida, o procedimento pode ser reservado para ser realizado em um dos locais de execução cadastrados.

- Agendar procedimento: Possuindo uma solicitação reservada, está na fase de agendá-la. Para tal, é informado a data de agendamento, horário e alguma observação se for pertinente. Caso a solicitação esteja incorreta ou não seja possível fazer seu agendamento, ela pode ser rejeitada.

- Reagendar procedimento: Se o procedimento não foi viabilizado na data marcada, há como reagendá-lo para ser feito em outra data, para tal, é informado a nova data, horário e observações.

- Informar realização: Com o procedimento feito, é informado a realização adicionando a data, nome do veterinário responsável e desfecho da cirurgia, que possui alguns status, como alta, óbito, intercorrência, etc. Além disso, também são adicionados documentos que comprovem que o procedimento foi de fato realizado, como foto do animal com o procedimento cirúrgico realizado, ficha de cadastramento e termo de autorização cirúrgica.

- Informar intercorrência: Caso alguma intercorrência pós-cirúrgica tenha acontecido, é informado que houve intercorrência adicionando a data de retorno do animal, o veterinário responsável, descrição e a documentação de comprovação.

- Cadastro de locais de execução: Cadastro de clínicas/hospitais em que os procedimentos podem ser realizados e de castramóveis, veículos que percorrem os municípios do estado para realizar cirurgias de castração em cães e gatos.

- Adição de vagas nos locais de execução: Distribuição da quantidade de vagas para cães e gatos em cada local de execução.

4.3 Fronted do RegPet

Nesta seção será apresentado o frontend do sistema, os usuário que o compõem e suas funções. O frontend do sistema foi desenvolvido para se adaptar a cada tipo de usuário e suas respectivas funções, com o objetivo de oferecer uma interface intuitiva e funcional

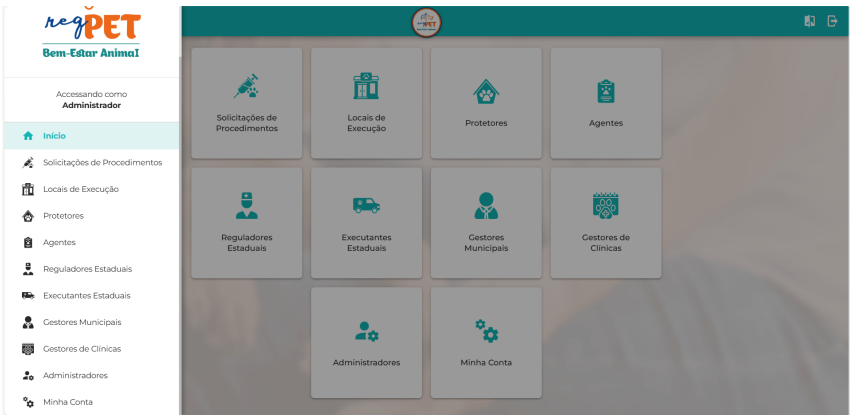
para cada um. A seguir, são detalhados a organização dos usuários, a estrutura visual do sistema e exemplos de telas que o compõem.

4.3.1 Organização dos usuários

O RegPet possui diferentes perfis de acesso, cada um com permissões específicas definidas de acordo com as responsabilidades atribuídas a cada tipo de usuário. Entre os perfis existentes, estão:

- Administrador: Possui acesso total ao sistema, podendo gerenciar todos os usuários, visualizar solicitações de procedimentos, cadastrar locais de execução e editar a própria conta. Uma visão geral de como a interface está estruturada pode ser vista na figura 4 abaixo:

Figura 4 – Interface do administrador



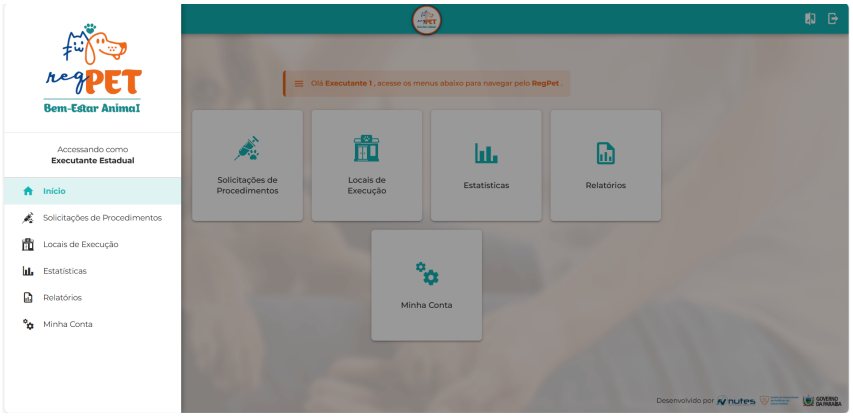
Fonte: Elaborado pela autora, 2025.

- Protetor: Envia as solicitações de procedimentos e faz o acompanhamento dessa solicitação pela ferramenta, ademais, ele também pode editar a própria conta.

- Regulador Estadual: Analisa e regula(valida e reserva) todas as solicitações que serão executadas pelo Governo. Também pode exportar relatórios e editar a própria conta.

- Executante Estadual: Disponibiliza as vagas do governo nos locais de execução, garante a execução dos procedimentos e finaliza as solicitações. Somado a isso, ele pode visualizar as estatísticas, exportar relatórios e editar a própria conta. Uma visão geral de como a interface está estruturada pode ser vista na figura 5 abaixo:

Figura 5 – Interface do executante estadual

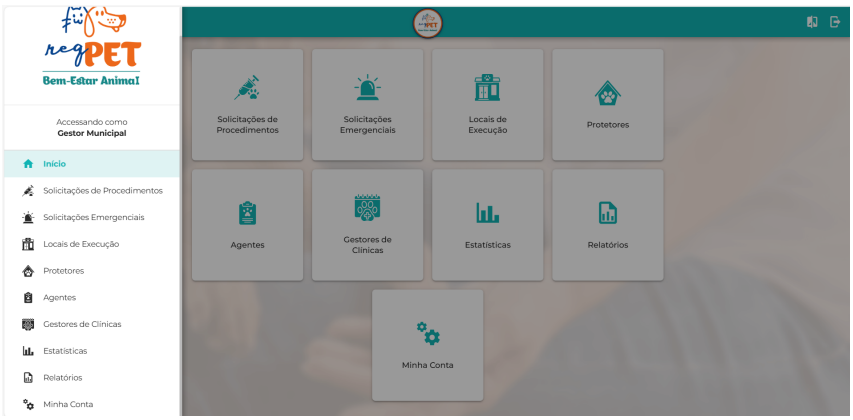


Fonte: Elaborado pela autora, 2025.

- Agente Municipal: Usuário que exerce as funções tanto de regulador municipal quanto de executante municipal. Como regulador, ele analisa e regula as solicitações apenas do seu município, além de exportar relatórios e editar a própria conta. Já como executante, disponibiliza as vagas do município e garante a realização e finalização da solicitação. Ele também pode visualizar as estatísticas, exportar relatórios e editar a própria conta.

- Gestor Municipal: Visualiza solicitações de procedimentos, locais de execução e estatísticas, cria solicitações emergenciais, cadastra protetores, agentes e gestores de clínicas, exporta relatórios e edita a própria conta. Uma visão geral de como a interface está estruturada pode ser vista na figura 6 abaixo:

Figura 6 – Interface do gestor municipal



Fonte: Elaborado pela autora, 2025.

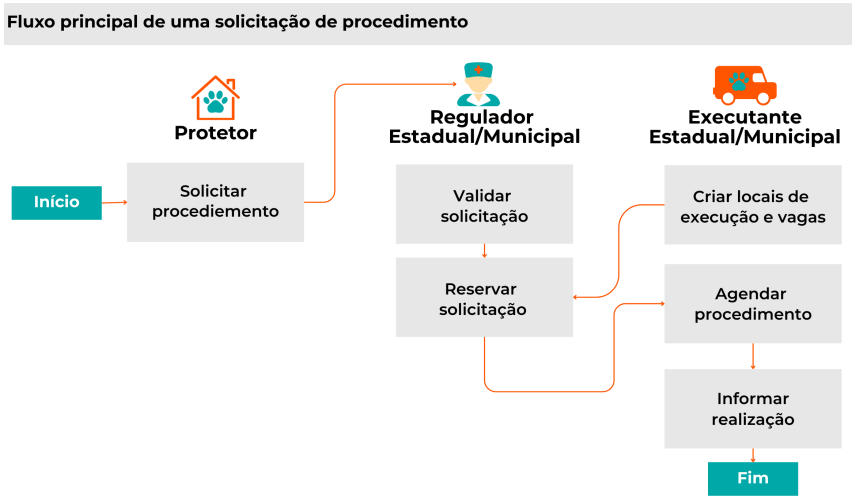
- Gestor de Clínicas: Gerencia a clínica, as vagas e solicitações da clínica que é responsável e finaliza a solicitação.

4.3.2 Funcionamento do sistema

Como pode ser visto na figura 7, para alcançar o objetivo do sistema, cada procedimento a ser realizado deve, primeiramente, ser solicitado por meio da plataforma, com o preenchimento do cadastro do protetor responsável e o envio das imagens e documentos exigidos. Após fazer a solicitação, os funcionários da Gerência Operacional de Políticas da Causa Animal farão a gestão e regulação, através do regulador, encaminhando para o processo de execução do procedimento.

Com isso, os municípios atuarão no processo de execução através do cadastro e disponibilização de locais para execução desses procedimentos (hospitais veterinários, clínicas e outros). Por fim, será necessário o envio de documentos e imagens comprobatórias, demonstrando que o animal foi devidamente atendido pelo programa.

Figura 7 – Fluxograma de uma solicitação de procedimento



Fonte: Elaborado pela autora, 2025.

4.4 Testes de E2E no RegPet

Nesta seção será apresentado todo o processo relacionado aos testes E2E realizados no sistema, desde seu planejamento, design, execução até os resultados e rastreamento de bugs.

Na figura 8 pode ser visto todo o processo utilizado para testar o frontend do RegPet. A primeira etapa é a de planejamento, na qual as novas funcionalidades são mapeadas em possíveis casos de teste. Em seguida, durante a fase de definição dos casos de teste, os cenários de teste são desenvolvidos e devidamente estruturados, cada um com seus critérios de aceitação.

Posteriormente, na fase de execução, os testes são executados diretamente no sistema. Os resultados obtidos são então avaliados, permitindo uma análise dos testes e do sistema. Por fim, caso sejam identificados erros, há o registro dos bugs e o acompanhamento até a

correção final.

Figura 8 – Processo de testes

Fonte: Elaborado pela autora, 2025.

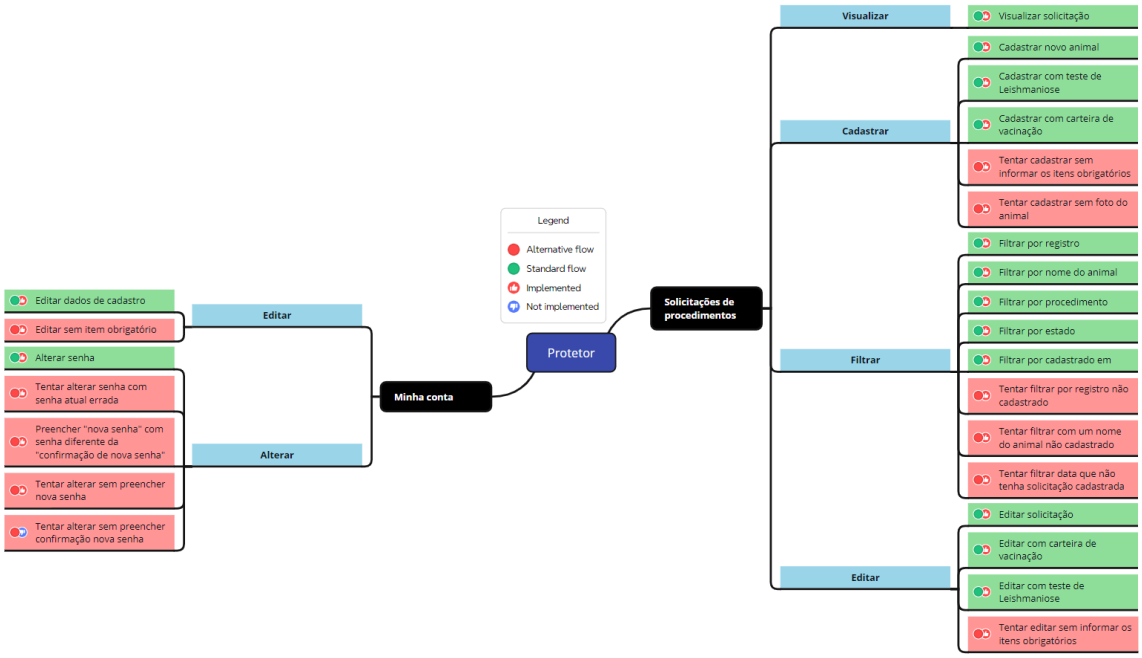
4.4.1 Planejamento dos testes

A fase inicial dos testes começa com o planejamento, para isso, é definido o objetivo do teste, ou seja, o que precisa ser validado e quais são as funcionalidades e usuários envolvidos nessa etapa. No contexto dos testes do RegPet, todas as principais funcionalidades disponibilizadas no sistema foram verificadas através de testes E2E automatizados, com o objetivo de verificar se todas as ações disponibilizadas estavam funcionando de acordo com os requisitos antes de serem implantadas no ambiente de produção.

Para possuir um rastreamento dos testes, foram utilizados mapas mentais para rastrear os possíveis casos de testes, através de uma ferramenta chamada Xmind. Para que todas as funcionalidades pudessem ser cobertas e testadas como um usuário real, os casos de testes foram organizados de acordo com cada usuário do sistema. Dessa forma, foram criados testes para funcionalidades presentes em cada “aba” ou página de cada perfil.

Com relação a classificação dos casos de testes, eles foram divididos em fluxos padrões e fluxos alternativos. Em que o fluxo padrão é aquele em que a funcionalidade sendo testada segue um caminho comum e conhecido, já o alternativo, é um conjunto de ações que levam a um caminho contrário ao fluxo padrão, seja por escolha do usuário ou por erros e exceções.

Figura 9 – Mapa mental do protetor



Fonte: Elaborado pela autora, 2025.

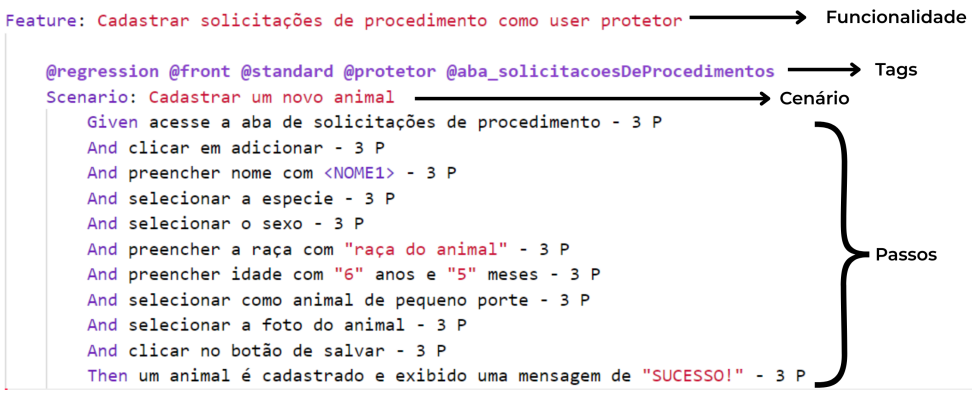
Na figura 9 está um exemplo de um mapa mental para o usuário protetor, nele, os retângulos pretos representam as abas/páginas disponíveis neste usuário, os retângulos azuis são as funcionalidades apresentadas em cada página, por fim, no último nível do diagrama, estão os casos de testes. Os casos de teste que possuem cor verde representam o fluxo padrão e os em cor vermelha, os fluxos alternativos. Com o objetivo de ter um maior controle dos casos de teste implementados ou não, tags com essa finalidade também foram adicionadas.

4.4.2 Definição dos testes

Na fase de definição dos testes, os requisitos e novas funcionalidades desenvolvidas são transformadas em casos de teste. A fim de alcançar tal objetivo, uma série de passos devem ser seguidos, como elaborar os cenários de teste, coletar IDs, executar os testes e avaliar os resultados.

Dessa forma, o primeiro passo é preparar os cenários de teste utilizando a linguagem Gherkin, definindo cada passo necessário para alcançar o critério de aceitação estabelecido. Na figura 10 está a representação de um cenário de teste utilizando o Gherkin, nela, é escrito a funcionalidade sendo testada, o cenário escolhido para o teste, os passos para atingir o objetivo final e as tags que ajudam os testadores a terem uma melhor rastreabilidade dos testes.

Figura 10 – Cenário de teste



Fonte: Elaborado pela autora, 2025.

Com os cenários criados, os campos e elementos necessários para a interação são coletados e as funções que representam as ações dos usuários nessas funcionalidades também são criadas. Os IDs podem ser coletados através do devtools do navegador ou por meio dos seletores disponíveis no Cypress. Na figura 11, pode ser visto como os identificadores dos elementos são organizados para uma determinada feature:

Figura 11 – Arquivo com os indentificadores

Fonte: Elaborado pela autora, 2025.

Com relação às funções dos testes, a figura 12 abaixo representa o principal arquivo de teste, o ".spec", em que é feito o mapeamento da linguagem Gherkin para os testes. A construção dele segue o mesmo princípio do cenário de teste, possuindo o pré-requisito de execução, os passos para atingir o objetivo do teste e o critério de aceitação. Os critérios de aceitação podem ser entendido como "o critério que um sistema ou componente deve satisfazer para ser aceito por um usuário, cliente ou outra entidade autorizada"(IEEE, 1990). Sendo ele de fundamental importância para garantir que os casos de testess estão alinhados com os requisitos da funcionalidade sendo testada.

Figura 12 – Arquivo `.spec`

Fonte: Elaborado pela autora, 2025.

Como pode ser observado, foi adotado a técnica BDD para a especificação dos casos de teste. Essa abordagem traz benefícios principalmente no cenário de testes E2E, já que é possível especificar como o sistema deve se comportar do ponto de vista do usuário de forma simples e clara.

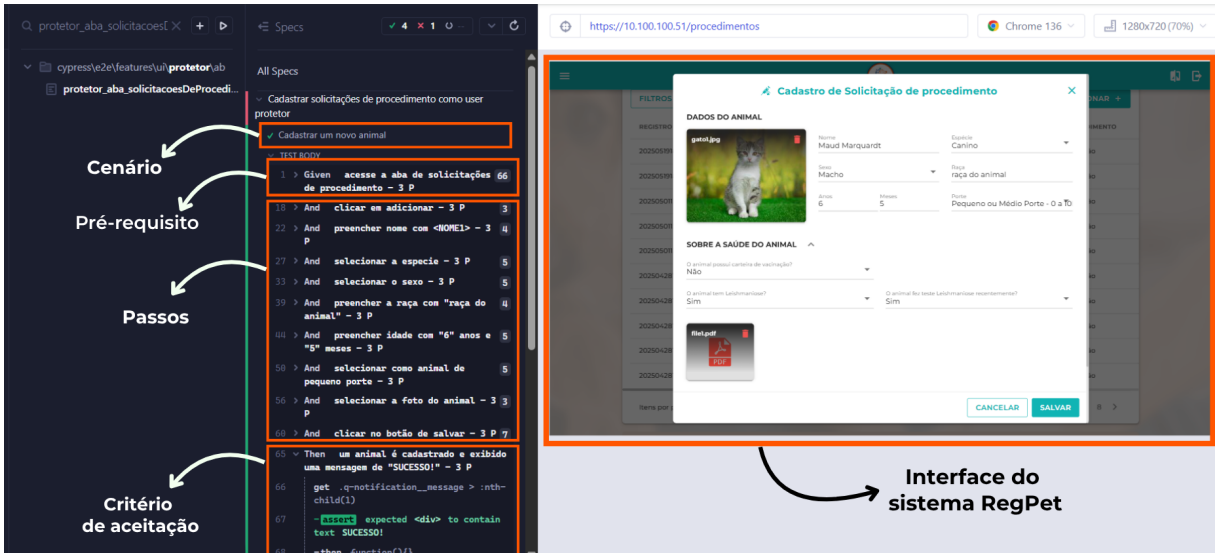
4.4.3 Execução dos testes

Com o código pronto, os testes podem ser executados utilizando o framework de testes Cypress, JavaScript e Node Js. A execução começa com o comando "npm run open", em que é aberta uma interface gráfica que permite selecionar e executar os testes manualmente. Ao escolher um teste, durante a sua execução é possível visualizar a interface do sistema e as interações que vão acontecendo ao longo do teste.

Também é possível realizar essa execução em segundo plano pela linha de comando, entretanto, não é possível visualizar as interações com a interface como na primeira forma descrita.

Na figura 13, há um exemplo de um teste realizado utilizando o modo interativo,é possível visualizar cada etapa do teste e as interações com o sistema diretamente no navegador.

Figura 13 – Modo interativo do Cypress



Fonte: Elaborado pela autora, 2025.

Durante a execução, o Cypress automatiza todos os passos que foram descritos no arquivo ".spec", ou seja, ele acessa a URL do sistema por meio de um navegador especificado, interage com os elementos que foram definidos no arquivo mostrado na subseção anterior, avalia as validações estabelecidas e também pode gerar imagens e vídeos da execução dos testes.

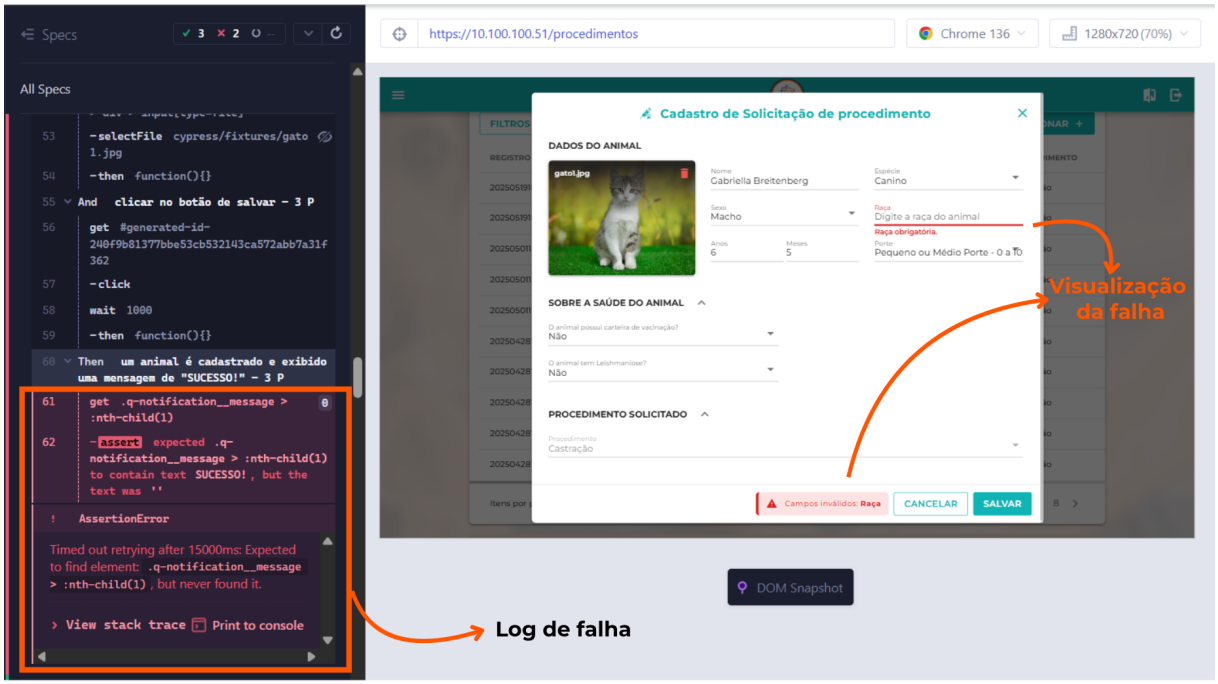
4.4.4 Análise dos resultados

Após a execução dos testes, o Cypress apresenta uma série de resultados sobre cada cenário testado. Com ele, pode ser visto quantos cenários foram aprovados, os passos executados para chegar ao sucesso do teste e o comportamento do sistema.

Para os testes falhos, também pode ser visto quais e quantos testes foram reprovados e quais etapas não atenderam os critérios de aceitação. Somado a isso, para se ter uma melhor visão da situação de erro, é disponibilizado pelo Cypress logs de erro, prints de tela do momento da falha e, se configurado, gravações em vídeo da execução.

A figura 14 mostra um teste falho, em que é exibido o log de falha e também o seu real motivo de ter falhado, que pode ser visto pela a interface disponibilizada pelo Cypress.

Figura 14 – Modo interativo do Cypress



Fonte: Elaborado pela autora, 2025.

Esses tipos de recursos visuais são muito úteis para que o testador consiga ter uma melhor compreensão do que ocorreu no instante da falha, como problemas ao carregar o sistema, elementos que não foram adicionados ou comportamentos inesperados da interface. Somado a isso, os relatórios permitem verificar:

- A quantidade de testes executados, aprovados e reprovados.
- Identificar padrões de falhas, como problemas recorrentes em determinados componentes.
- Monitorar a evolução da qualidade do software ao longo do tempo.

4.4.5 Gerenciamento de bugs

Caso algum bug tenha sido detectado, o gerenciamento dos bugs foi feito por meio de um Software de rastreamento de bugs chamado Bugzilla. E para realizar o cadastro dos bugs, é seguido um template, em que são definidos:

- Pré-requisitos: São as condições iniciais para conseguir reproduzir o bug, em que são apresentadas informações como o tipo de perfil, suas credenciais de login, o ambiente no qual o bug aconteceu e a sua versão.

- Descrição: Resumo do comportamento encontrado, explicando de forma geral a funcionalidade afetada pelo bug e o impedimento que ela gera para o fluxo de atividades no sistema.

- Passos para reprodução: Sequência de ações que precisam ser feitas para reproduzir o bug, ajudando outros testadores e desenvolvedores a terem uma melhor compreensão de como atingir o bug.

- Resultado observado: Real resultado obtido ao seguir os passos anteriores, sendo anexado mensagens de erro e logs de erro.

- Resultado esperado: Desempenho correto das funcionalidades, ou seja, o correto comportamento que era esperado ao executar uma ação.

- Frequência: Indica a recorrência em que o bug acontece, já que há bugs que podem ocorrer sempre, ocasionalmente, uma única vez e raramente.

- Evidências: Imagens, vídeos, mensagens de erro e logs de erro que demonstram o erro de forma visual, auxiliando os desenvolvedores a terem uma visão mais clara do bug.

- Severidade: Indicador do impacto que o bug tem no funcionamento do sistema, no qual o seu nível pode ser crítico, alto, médio e baixo. Indicar a severidade ajuda a equipe a priorizar os bugs que tem urgência de correção ou não.

Na figura 15, pode ser visto como as informações descritas acima são organizadas no Bugzilla:

Figura 15 – Bug no Bugzilla

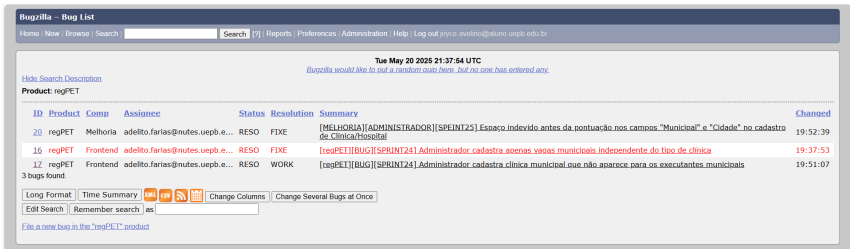
```
PRÉ-REQUISITOS:
    - Perfil: Administrador
    - Autenticar com: admin@regpet.com / admin123
    - Ambiente: https://10.100.100.51/

DESCRIÇÃO: Ao cadastrar uma clínica municipal com o usuário Administrador, a clínica
cadastrada não aparece na visão do executante municipal daquele município onde a clínica
se encontra.
-OBS.: O mesmo comportamento ocorre quando o Administrador cadastra uma clínica estadual,
e o executante estadual também não consegue visualizar. As Clínicas só ficam visíveis para
os executantes quando eles mesmo cadastram.

PASSOS PARA REPRODUÇÃO:
     1. Logar no sistema com um usuário com perfil Administrador
     2. Clicar na aba "Locais de Execução"
     3. Clicar no botão "+ Adicionar"
     4. Selecionar Clínica/Hospital
     5. Preencher os campos necessários adicionando um município válido e colocando a
clínica como municipal
     6. Clicar no botão "Salvar"
     7. Logar como um agente daquele município que possua a capacidade de executante
municipal
     8. Clicar na aba "Locais de Execução" e verificar que a clínica não aparece

RESULTADO ESPERADO: ✅ O sistema deve mostrar a nova clínica cadastrada para o Agente
Executante Municipal do município da clínica.

RESULTADO OBSERVADO: ❌ O sistema cadastra a clínica como municipal, mas não mostra a
mesma ao Executante de seu município.

FREQUÊNCIA:

(X) Sempre
( ) Quase nunca
( ) Ocasionalmente
( ) Uma única vez
```

Fonte: Elaborado pela autora, 2025.

Após um bug ser cadastrado, os desenvolvedores relacionados com o tipo de bug cadastrado(bug de frontend ou backend) são notificados por e-mail. Com isso, eles conseguem ter acesso às informações do bug cadastrado, bem como à lista de todos os bugs já cadastrados até o presente momento. Pelo Bugzilla, o desenvolvedor e testador também conseguem se comunicar, assim como visualizar a mudança do status do bug, ou seja, se ele foi confirmado, resolvido, verificado, validado, etc.

Na figura 16 abaixo está uma lista dos bugs já cadastrados, onde pode ser observado o ID do bug, o nome do sistema, o tipo de bug(frontend, backend, melhoria), o status e resolução, quem cadastrou o bug, uma descrição dele e quando foi modificado.

Figura 16 – Lista de bugs no Bugzilla



Fonte: Elaborado pela autora, 2025.

5 ESTUDO DE CASO: ANÁLISE DE FALHAS EM TESTES E2E NA EVOLUÇÃO DO REGPET

Neste capítulo será abordado como a metodologia desenvolvida foi aplicada ao sistema do RegPet, com o objetivo de analisar as falhas dos testes no contexto da evolução da aplicação. Com isso, o processo foi dividido em alguns passos, sendo eles:

1. Configuração do ambiente de testes
2. Implantação local e versionamento
3. Geração das massas de dados e execução dos testes
4. Relatório dos testes

Tais etapas são detalhadamente descritas nas seções a seguir.

5.1 Configuração do ambiente de testes

A configuração do ambiente de testes representou a etapa inicial da aplicação da metodologia proposta. Para a realização dos testes no sistema RegPet, foi utilizado o framework Cypress, com a linguagem de programação JavaScript e o interpretador de código Node.js. A instalação dos pacotes necessários para execução dos testes foi através do comando “npm i”, sendo a versão do Cypress a “13.0.0”, ou seja, o gerenciador do pacote pode instalar qualquer versão compatível com a 13.0.0, já a versão do NodeJs foi a maior ou igual à “6.9.0”.

Com relação a forma de execução dos testes, foi escolhido o modo interativo para se ter uma melhor compreensão do comportamento dos testes ao serem aplicados no sistema. Com isso, as configurações do script foram definidas no arquivo “Package.json”, como por exemplo: “cypress open –e2e –browser chrome –config video=false”. Esse script indica que ao executar o comando “npm run open”, os testes do tipo E2E serão executados no modo interativo, utilizando o navegador Google Chrome, com a gravação de vídeos desativada durante os testes.

De modo geral, não foram realizadas alterações significativas nos arquivos de teste, com exceção de um arquivo denominado “defaultEvironment.js”, o qual contém as variáveis de ambiente padrão utilizadas pelo sistema. O arquivo original pode ser visto na figura 17 abaixo:

Figura 17 – Variáveis de ambiente

Fonte: Elaborado pela autora, 2025.

Nesse arquivo, estão definidas a URL da API Gateway utilizada no ambiente de desenvolvimento, responsável por centralizar o tráfego de requisições para os serviços de backend, e a URL correspondente ao ambiente de testes da aplicação. No entanto, como o ambiente de testes foi configurado localmente, foi necessário ajustar esse endereço para que apontasse para um endpoint local, resultando na configuração presente na figura 18 a seguir:

Figura 18 – Variáveis de ambiente

Fonte: Elaborado pela autora, 2025.

Dessa forma, ao executar um teste, ele será redirecionado para o endereço correto sem necessitar de uma VPN, como é o caso do endereço definido anteriormente.

5.2 Implantação local e versionamento

Com o repositório de testes configurado, a próxima etapa é versionar o ambiente de testes, bem como o próprio repositório dos testes. Esse processo tem início com a seleção da versão desejada. Para o trabalho, foram selecionadas três versões sequenciais, com o objetivo de ter uma melhor compreensão da evolução dos testes ao longo dessas versões.

Assim, por meio do comando “git checkout v1.1”, foi possível retornar o repositório de testes ao estado correspondente à versão v1.1, recuperando todas as configurações e arquivos presentes naquele ponto específico do histórico de versionamento.

Após isso, utilizando o comando “docker compose up -d –build”, todos os serviços são inicializados com o auxílio da ferramenta de conteinerização Docker e de sua extensão Docker Compose. Por meio delas, todo o ambiente da API, incluindo seus módulos, é empacotado e iniciado de forma automatizada.

Por fim, o comando “git submodule update -N –recursive –init” garante que os submodules sejam inicializados e atualizados para a versão específica registrada no commit atual do repositório principal, ou seja, a versão corresponde a v1.1 do ambiente de teste. Isso é feito recursivamente em todos os níveis de submodules, usando apenas o que está localmente disponível, sem buscar dados remotamente.

Depois disso, o ambiente de testes e repositório de testes estão sincronizados, com isso, o frontend necessário para realizar os testes está com a sua correta versão, garantindo uma melhor integridade dos testes, bem como seus resultados. Todo esse processo é repetido para as próximas versões do ambiente de testes também estudadas, sendo elas a v1.2 e v1.3.

5.3 Geração das massas de dados e execução dos testes

Com o ambiente devidamente configurado, a etapa de geração de massas e execução dos testes pode ser realizada. Para gerar as massas de dados que vão alimentar os casos de testes, foram utilizadas um conjunto de técnicas que envolveram o uso de bibliotecas como o Faker.js e o 4Devs, assim como funções criadas internamente pela equipe de testes que seguiram a seguinte distribuição:

- 4Devs: RG, CEP;
- FakerJs: Nome, email, senha, ramal, número, títulos, etc;
- Funções internas: telefones, CPF, CNPJ.

Com base nos dados fictícios gerados, foi desenvolvido um comando customizado no Cypress com o objetivo de criar massas de dados dinâmicas para diferentes perfis de usuário, adaptando-se conforme a necessidade de cada cenário de teste. Esses dados foram armazenados em um arquivo no formato JSON, o qual contém um array de objetos. Cada objeto neste array representa um usuário gerado, com todos os atributos necessários para simular o comportamento real da aplicação durante os testes, conforme a figura 19 a seguir:

Figura 19 – Massas de dados do administrador

Fonte: Elaborado pela autora, 2025.

A quantidade de massas de dados geradas pode ser configurada conforme a necessidade de cada cenário de teste, sendo possível passá-la como argumento para a função responsável pela geração das massas. Para utilizar os dados gerados, os testes fazem uso de um método chamado "gera_random", o qual retorna um número aleatório utilizado para acessar dinamicamente um item do array correspondente ao perfil em questão no arquivo JSON.

No entanto, vale destacar que os dados utilizados não são removidos do arquivo após o uso. Isso significa que, ao longo da execução de múltiplos testes, os mesmos dados podem ser reutilizados, o que pode ocasionar conflitos por duplicidade, especialmente em cenários que envolvam operações de cadastro ou validações de unicidade, como CPF, e-mail ou CNPJ.

Por fim, com as massas de dados devidamente geradas para cada perfil de usuário, as suítes de testes foram executadas individualmente e de forma manual, permitindo a verificação detalhada do comportamento da aplicação em diferentes cenários.

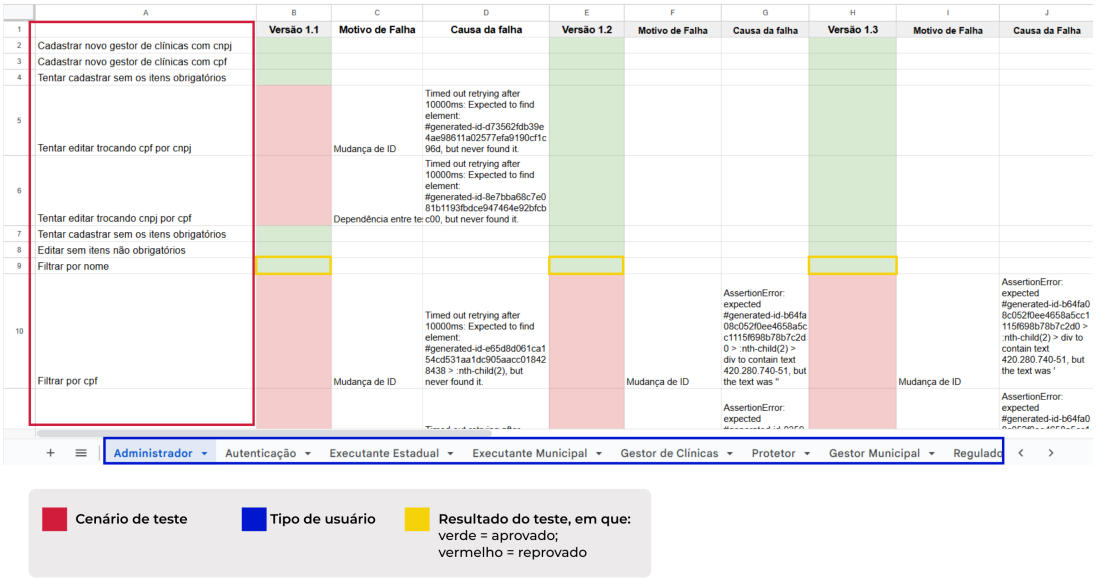
5.4 Relatório dos testes

Com o objetivo de manter uma organização eficiente durante o processo de execução dos testes, os resultados de cada versão da aplicação foram coletados sequencialmente, iniciando pela versão v1.1, seguida pela v1.2 e, por fim, pela v1.3. Além disso, foi elaborada uma tabela específica para cada perfil de usuário do sistema, estruturada de acordo com as versões testadas e os respectivos cenários aplicáveis a cada perfil, facilitando a análise e comparação dos resultados obtidos. Dessa forma, ao executar e analisar cada teste individualmente, as seguintes informações de cada um foram coletadas:

- Cenário de teste;
- Versão analisada;
- Log de erro;
- Motivo da falha;
- Resultado do teste.

A figura 20 abaixo representa uma tabela construída para o perfil do administrador, em que é possível ter uma melhor compreensão dessas informações coletadas.

Figura 20 – Resultados do administrador



Fonte: Elaborado pela autora, 2025.

Embora a figura não exiba todos os cenários de teste relacionados ao perfil de administrador, é importante destacar que cada um deles foi devidamente analisado. Para todos os cenários, foram documentados o resultado obtido, os logs de erro (causa da falha) e o motivo específico da não conformidade. Esse mesmo procedimento foi repetido para os demais perfis de usuário do sistema, garantindo uma análise abrangente e padronizada dos testes realizados. Esses reultados podem ser vistos no link anexado ao rodapé.[1]

[1] planilha com resultados

6 RESULTADOS

Nesta seção, serão apresentados os resultados da metodologia aplicada nas três versões do RegPet. Com isso, como resultado, foram coletadas métricas sobre o comportamento das falhas ao longo dessas versões, com o intuito de ter um melhor entendimento das falhas no contexto de evolução do sistema. Na figura 21 abaixo está uma visão geral de quantos testes passaram e falharam em cada versão estudada, assim como o total de testes que existem em cada versão.

Figura 21 – Visão geral dos resultados dos testes

Versão	#Passou	#Falhou	Total
V1.1	288	117	405
V1.2	292	143	435
v1.3	251	186	437

Fonte: Elaborado pela autora, 2025.

Com base nos objetivos específicos definidos, foram estabelecidas as seguintes questões de pesquisa para respondidas com base na metodologia proposta e aplicada no contexto do RegPet:

- Objetivo 1: Mapear as causas das falhas em cada versão
  - QP1.1: Quais são as causas das falhas nos testes?
  - QP1.2: Qual a distribuição das causas das falhas por versão?
- Objetivo 2: Mapear as falhas em cada versão
  - QP2.1: Quais são os tipos das falhas nos testes?
  - QP2.2: Qual a distribuição das falhas por versão?
- Objetivo 3: Analisar a relação entre causa e falha entre as versões
  - QP3.1: Qual a relação entre as categorias de falhas e os logs de erro?
- Objetivo 4: Analisar regressões entre as versões
  - QP4.1: Qual a quantidade de testes que passavam na versão anterior e começaram a falhar na próxima?

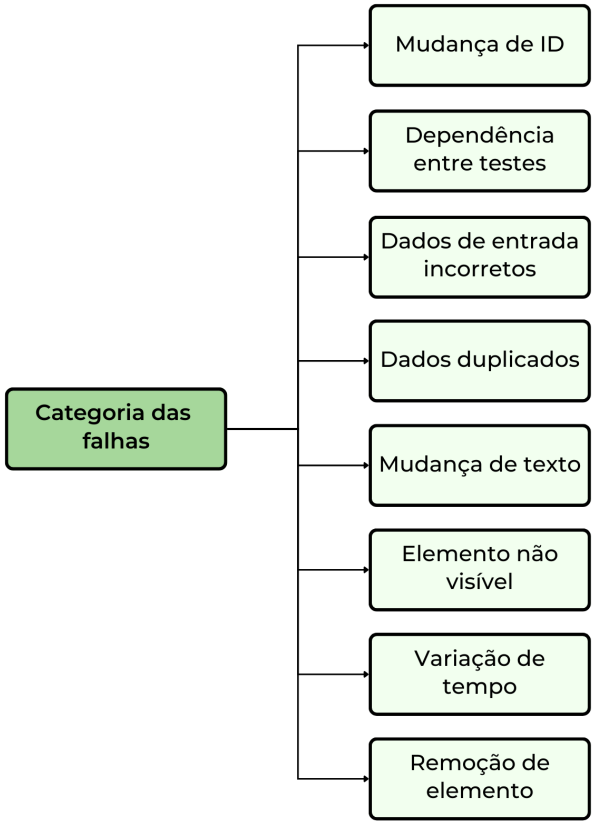
6.1 Objetivo 1: Mapear as causas das falhas em cada versão

Para mapear as causas de falhas por versão, elas foram primeiramente categorizadas e, então, distribuidas de acordo com cada versão do sistema, como pode ser observado nas subsecções a seguir.

6.1.1 Quais são as causas das falhas nos testes?

A partir da aplicação da metodologia e geração dos relatórios, foi possível construir categorias dos testes que classificam as falhas mais recorrentes identificadas durante os testes. Essa categorização contribuiu para uma melhor compreensão das principais causas de falhas e facilitou a identificação de padrões de comportamento do sistema em diferentes versões.

Figura 22 – Categorização das falhas



Fonte: Elaborado pela autora, 2025.

A figura 22 mostra a categorização das falhas, com a descrição detalhada de cada uma delas descritas a seguir:

- C1 - Mudança de ID: Os elementos da interface utilizados durante a execução dos testes automatizados são identificados por meio de um identificador único (ID). No entanto, caso alguma alteração na estrutura da interface ocorra, como a substituição

ou reconfiguração de um componente, os IDs atribuídos aos elementos podem ser modificados. Essa mudança compromete a integridade do teste automatizado, que deixa de reconhecer o elemento, resultando na sua falha.

- C2 - Dependência entre testes: Alguns testes exigem que determinados dados estejam previamente cadastrados ou que testes anteriores sejam executados com sucesso, pois há uma relação de dependência entre eles. Caso essas condições não sejam atendidas, os testes subsequentes tendem a falhar, não por erro funcional da aplicação, mas por falhas na própria construção do cenário de teste.

- C3 - Dados de entrada incorretos: Dados de entrada inválidos ou incompatíveis com as regras de negócio da aplicação que são utilizados na criação do cenário de teste, esses erros afetam diretamente a execução do cenário e comprometem sua validação.

- C4 - Dados duplicados: No RegPet, há diversos cenários que envolvem campos que exigem valores únicos, como CPF,CNPJ e e-mail, entretanto como os dados utilizados das massas de dados não são removidos automaticamente do JSON após seu uso, pode haver reutilização não intencional dos mesmos valores, causando erros por duplicidade.

- C5 - Mudança de texto: A alteração de textos exibidos na interface, como rótulos, mensagens de sucesso e erro, ou títulos de botões, pode fazer com que testes que dependem de valores textuais fixos não reconheçam os elementos esperados, gerando falhas mesmo quando a funcionalidade permanece inalterada.

- C6 - Elemento não visível: Quando um teste tenta interagir com um elemento que, apesar de estar presente no DOM, não está visível ou acessível no momento da execução. Isso pode acontecer devido a problemas de carregamento, estados de exibição condicionais ou animações em andamento, impedindo a manipulação do elemento e resultando em falha.

- C7 - Variação de tempo: Em alguns casos, a aplicação precisa de alguns instantes para realizar certas operações, já que nem sempre os elementos da interface estão imediatamente disponíveis após a navegação ou o envio de uma requisição. Então, quando a automação tenta interagir com esses elementos antes que eles sejam carregados completamente, ocorre um erro de sincronização, reportado como timeout.

- C8 - Remoção de elemento: Quando um componente ou funcionalidade é removido da aplicação sem a devida atualização dos testes correspondentes, os testes continuam tentando acessar elementos inexistentes, o que inevitavelmente leva à falha.

6.1.2 Qual a distribuição das causas das falhas por versão?

Na figura 23 a seguir é possível visualizar como está a distribuição das categorias de falhas em cada versão.

Figura 23 – Categorias das falhas por versão

Versão	Categoria das falhas								Total
	C1	C2	C3	C4	C5	C6	C7	C8	
V1.1	5	90	1	10	1	1	9	0	117
V1.2	4	97	1	5	1	1	12	22	143
V1.3	6	50	1	10	4	0	71	44	186
Total	15	237	3	25	6	2	92	66	446

Fonte: Elaborado pela autora, 2025.

Com base na tabela apresentada, é observado que a distribuição das categorias de falhas variou ao longo das versões do sistema, na versão v1.1, a categoria com maior incidência foi a de dependência entre testes (C2), com 90 falhas, demonstrando que muitos testes estavam atrelados a dados ou execuções prévias. Também houve destaque para as categorias de dados duplicados (C4), com 10 falhas, e variação de tempo (C7), com 9 falhas. Nesta versão, a categoria de remoção de elemento (C8) não registrou nenhuma ocorrência.

Na versão v1.2, as falhas por dependência entre testes (C2) se mantiveram como as mais recorrentes, com 97 ocorrências, mas percebe-se um crescimento significativo nas falhas por remoção de elemento (C8), que passou a registrar 22 casos. A categoria de variação de tempo (C7) também aumentou para 12 falhas, enquanto as demais categorias permaneceram com poucos registros.

Já na versão v1.3, houve uma mudança no padrão: a categoria variação de tempo (C7) passou a ser a mais incidente, com 71 falhas, indicando maior instabilidade na disponibilidade dos elementos da interface. A remoção de elementos (C8) também cresceu, somando 44 ocorrências, enquanto as falhas por dependência entre testes (C2) diminuíram significativamente para 50 casos.

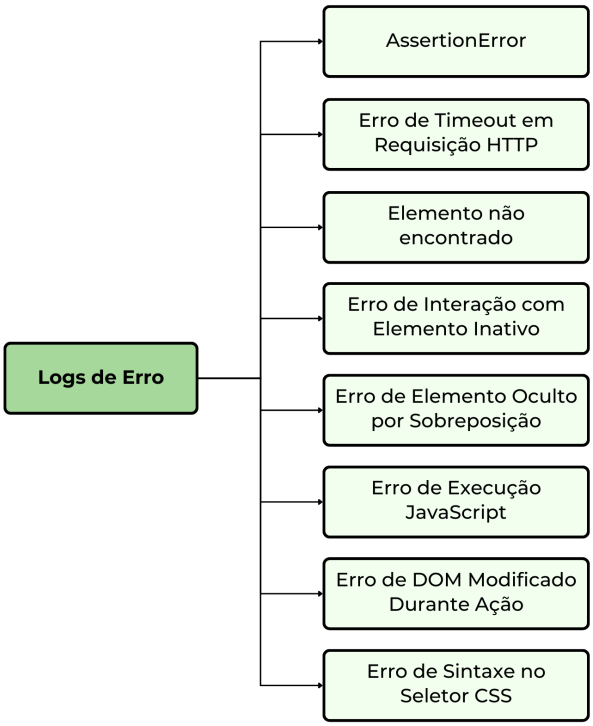
6.2 Objetivo 2: Mapear as falhas em cada versão

Para mapear as falhas em cada versão, foram coletados os tipos de falhas(logs de erro) mais comuns ao executar os testes, então, elas foram distribuidas de acordo com cada versão do sistema, como pode ser observado nas subsecções a seguir.

6.2.1 Quais são os tipos das falhas nos testes?

Além da categorização das falhas identificadas, os logs de erro que foram gerados durante a execução dos testes também foram coletados e analisados, isso com o intuito de ter informações mais detalhadas sobre o comportamento da aplicação no momento das falhas.

Figura 24 – Logs de erro



Fonte: Elaborado pela autora, 2025.

A Figura 24 apresenta tipos de logs de erro registrados durante a execução dos testes, sendo a descrição detalhada de cada um deles apresentada a seguir:

- T1 - AssertionError: Esse tipo de erro ocorre quando uma asserção esperada no teste não é satisfeita. Por exemplo, ao esperar que determinado elemento contenha um texto específico, mas o valor encontrado é diferente ou está ausente.

- T2 - Erro de Timeout em Requisição HTTP: Ocorre quando a resposta de uma requisição para a API ou backend demora mais do que o tempo limite configurado, o que pode ser um indicativo de possíveis problemas de desempenho na aplicação ou instabilidade no ambiente de testes.

- T3 - Elemento não encontrado: Indica que o elemento buscado pelo seletor definido no teste não está presente no DOM no momento da execução, isso pode ocorrer por

problemas de sincronização, mudanças na interface ou carregamento incompleto da página, entre outros problemas.

- T4 - Erro de Interação com Elemento Inativo: Quando o Cypress tenta interagir com um elemento que está presente na interface, mas se encontra desativado, sem conseguir realizar ações como clique ou digitação.
- T5 - Erro de Elemento Oculto por Sobreposição: Acontece quando há algum elemento na interface que está tecnicamente visível no DOM, entretanto coberto por algum outro elemento, como um modal, o que acaba impedindo a interação direta com esse elemento.
- T6 - Erro de Execução JavaScript: Indica que houve uma falha no código JavaScript da própria aplicação durante a execução do teste, geralmente relacionada a exceções não tratadas, como "TypeError" ou "ReferenceError".
- T7 - Erro de DOM Modificado Durante Ação: Esse erro é gerado quando o DOM sofre alterações inesperadas no exato momento em que o teste tenta realizar uma ação, como clicar ou digitar, resultando em falha por instabilidade da estrutura da página.
- T8 - Erro de Sintaxe no Seletor CSS: Está relacionado com erros no formato do seletor utilizado no teste, o que impede o Cypress de localizar corretamente o elemento desejado. Pode ocorrer devido a erros de digitação ou mudanças na estrutura HTML/CSS da aplicação.

6.2.2 Qual a distribuição dos logs de erro por versão?

Na figura 25 abaixo pode ser observado como está a distribuição dos logs de erro em cada versão.

Figura 25 – Logs de erro por versão

Versão	Tipos de falhas								Total
	T1	T2	T3	T4	T5	T6	T7	T8	
V1.1	45	9	55	0	2	6	0	0	117
V1.2	42	17	77	0	0	6	1	0	143
V1.3	27	13	76	57	8	3	1	1	186
Total	114	39	208	57	10	15	2	1	446

Fonte: Elaborado pela autora, 2025.

Na versão v1.1, os tipos de falha mais frequentes foram o erro de elemento não encontrado (T3), com 55 ocorrências, e o AssertionError (T1), com 45 registros. Esses dados

indicam que, nesse estágio inicial, os testes automatizados falharam com maior frequência por não localizar elementos na interface ou por não atenderem às condições esperadas nas asserções. Outros tipos de erros, como T2 (Timeout em requisição HTTP), T5 (Elemento oculto por sobreposição) e T6 (Erro de execução JavaScript), ocorreram em menor escala, e não houve registro de falhas dos tipos T4 (Interação com elemento inativo), T7 (DOM modificado) ou T8 (Erro de seletor CSS).

Já na versão v1.2, os logs de erro do tipo T3 (Elemento não encontrado) continuaram como os mais recorrentes, aumentando para 77 ocorrências. Os erros T1 (AssertionError) também permaneceram elevados, com 42 casos. Também houve um crescimento nos erros T2 (Timeout em requisição HTTP), que subiram de 9 para 17, e o primeiro registro de erro do tipo T7 (DOM modificado durante ação), com uma ocorrência. Apesar do aumento no total de falhas, os erros T4, T5, e T8 ainda não apareceram nesta versão.

Por fim, a versão v1.3, teve uma mudança na distribuição dos tipos de erro. O erro T3 (Elemento não encontrado) se manteve como o mais frequente, com 76 ocorrências, mas houve um aumento expressivo dos erros do tipo T4 (Interação com elemento inativo), com 57 casos, um tipo de falha que não havia sido registrado nas versões anteriores. Os erros T5 (Elemento oculto por sobreposição) também aumentaram, atingindo 8 ocorrências. Por outro lado, os AssertionErrors (T1) diminuíram para 27. As falhas do tipo T2, T6, T7 e T8 se mantiveram com níveis baixos.

6.3 Objetivo 3: Analisar a relação entre causa e falha entre as versões

Após coletar os logs de erros e categorizar as causas de falhas mais recorrentes, foi analisada a relação entre a causa e falha de erro entre as versões, a fim de compreeender se elas teriam alguma relação e se algum tipo de log de erro estaria mais relacionado com um tipo de categoria de falha específico. Essa análise pode ser vista na subsecção a seguir.

6.3.1 Qual a relação entre as categorias de falhas e os logs de erro?

Para compreender a relação entre as categorias de falhas e os logs de erro, foi documentada a quantidade de testes que falharam em cada categoria e que estavam associados a cada tipo de log de erro, considerando separadamente cada uma das versões analisadas.

A figura 26 abaixo representa essa relação para a versão v1.1 da aplicação.

Figura 26 – Relação entre as categorias de falhas e os logs de erro - V1.1

Tipo das falhas	Categoria das falhas								Total
	C1	C2	C3	C4	C5	C6	C7	C8	
T1		36	1	6	1	1			45
T2		3					6		9
T3	5	48		2					55
T4									0
T5				2					2
T6		3					3		6
T7									0
T8									0
Total	5	90	1	10	1	1	9	0	117

Fonte: Elaborado pela autora, 2025.

O log de erro T1 (AssertionError), que ocorre quando uma asserção esperada no teste não é satisfeita, está em grande parte relacionado à categoria C2 (dependência entre testes), com 36 ocorrências. Isso mostra que muitas asserções falharam por conta de pré-condições não atendidas, como a necessidade de dados prévios ou execução de testes anteriores. T1 também se relaciona às categorias C4 (dados duplicados, 6 ocorrências), C1 (mudança de ID, 1 ocorrência), C3 (dados de entrada incorretos, 1 ocorrência), C5 (mudança de texto, 1 ocorrência) e C6 (elemento não visível, 1 ocorrência), demonstrando que as falhas de asserção também são influenciadas por alterações na interface, visibilidade ou nos dados utilizados, mas sem grandes impactos.

O log de erro T2 (erro de timeout em requisição HTTP), que surge quando a resposta da API demora além do limite configurado, possui 6 ocorrências na categoria C7 (variação de tempo), indicando problemas de sincronização ou instabilidade na aplicação. Outras 3 ocorrências estão associadas à categoria C2 (dependência entre testes), sugerindo que alguns timeouts podem ter origem na ausência de dados esperados.

O log de erro T3 (elemento não encontrado), que representa falhas ao localizar elementos no DOM durante os testes, é predominante na categoria C2 (48 ocorrências). Isso demonstra que a ausência do elemento muitas vezes decorre de falhas no cenário de teste ou de dependências não satisfeitas. Além disso, T3 está relacionado com C1 (mudança de ID, 5 ocorrências) e C4 (dados duplicados, 2 ocorrências), indicando que alterações na interface e dados repetidos também contribuem para a falha na localização de elementos.

O log de erro T5 (erro de elemento oculto por sobreposição) aparece duas vezes e está vinculado à categoria C4 (dados duplicados), sugerindo que o problema pode ser causado por entradas repetidas que geram estados inesperados da interface, como a sobreposição de elementos.

O log de erro T6 (erro de execução JavaScript), que ocorre por exceções como TypeError ou ReferenceError, aparece 3 vezes e está completamente relacionado à categoria C2 (dependência entre testes), o que indica que erros de execução da aplicação também

podem ocorrer por falta de preparo do cenário de teste.

Já os logs T4 (erro de interação com elemento inativo), T7 (erro de DOM modificado durante ação) e T8 (erro de sintaxe no seletor CSS) não foram registrados nesta versão, indicando ausência de falhas associadas a esses tipos de erro.

De forma semelhante, a figura 27 a seguir representa essa relação entre os logs de erro e as categorias das falhas para a versão v1.2 do sistema.

Figura 27 – Relação entre as categorias de falhas e os logs de erro - V1.2

Tipo das falhas	Categoria das falhas								Total
	C1	C2	C3	C4	C5	C6	C7	C8	
T1	2	34	1	3	1		1		42
T2		9					8		17
T3	1	51		2		1		22	77
T4									0
T5									0
T6		3					3		6
T7	1								1
T8									0
Total	4	97	1	5	1	1	12	22	143

Fonte: Elaborado pela autora, 2025.

Nela, ao todo, foram identificadas 143 falhas, sendo que a categoria C2, que representa a dependência entre testes, permanece como a mais recorrente, com 97 ocorrências. Além, disso, os tipos de falha mais frequentes continuam sendo o T3, que indica "Elemento não encontrado", com 77 ocorrências, e o T1, correspondente ao erro "AssertionError", com 42 falhas. O T3 está fortemente associado à categoria C2, com 51 ocorrências, indicando que a ausência de elementos muitas vezes está relacionada à execução inadequada do cenário de teste anterior. Além disso, o T3 também se relaciona significativamente com a categoria C8, que indica a "Remoção de elementos da aplicação", representando 22 falhas, o que mostra que alguns testes não foram atualizados após alterações na interface do sistema, resultando em tentativas de acessar componentes inexistentes.

Já o T1 aparece principalmente ligado à categoria C2 (34 vezes), reforçando o impacto da dependência entre testes também sobre as asserções feitas no código. Há, ainda, ocorrências menores relacionadas a outras categorias, como C1 (mudança de ID), C3 (dados incorretos), C4 (dados duplicados), C5 (alteração de texto exibido) e C7 (variação de tempo), cada uma contribuindo com uma pequena quantidade de falhas.

O tipo T2, referente a "Timeout em requisição HTTP", registrou 17 ocorrências, divididas entre C2 (9 falhas) e C7 (8 falhas). Sugerindo que além da dependência entre testes, problemas de sincronização ou lentidão na aplicação estão contribuindo para a

falha nas requisições. O tipo T6, correspondente a “Erros de execução JavaScript”, teve 6 registros, também ligados às categorias C2 e C7, o que reforça a presença de falhas relacionadas ao carregamento e ao tempo de resposta da aplicação.

Por fim, o tipo T7, que indica que o “DOM foi modificado durante uma ação”, apareceu uma única vez, relacionado à categoria C1 (mudança de ID). Já os tipos T4, T5 e T8 não apresentaram nenhuma falha registrada nesta versão. Por fim, na figura 28 a seguir está essa relação entre os logs de erro e as categorias das falhas para a versão v1.3 do sistema.

Figura 28 – Relação entre as categorias de falhas e os logs de erro - V1.3

Tipo das falhas	Categoria das falhas								Total
	C1	C2	C3	C4	C5	C6	C7	C8	
T1	3	16		4	4				27
T2							11	2	13
T3	2	31		1				42	76
T4		2					55		57
T5				5			3		8
T6		1					2		3
T7	1								1
T8			1						1
Total	6	50	1	10	4	0	71	44	186

Fonte: Elaborado pela autora, 2025.

A quantidade total de erros aumentou para 186 ocorrências na versão v1.3, em comparação com as versões anteriores. Como pode ser visto, as categorias de falhas C7 (variação de tempo) e C8 (remoção de elementos da aplicação) ganharam ainda mais relevância, com 71 e 44 falhas, respectivamente. Contudo, a categoria C2 (dependência entre testes) continua sendo significativa, com 50 ocorrências, embora tenha diminuído em relação à versão anterior.

O tipo de falha T3 (Elemento não encontrado) permanece como o mais frequente, com 76 registros, sendo fortemente associado à categoria C8 (42 ocorrências) e também à C2 (31), o que indica que as falhas mais comuns ainda estão relacionadas tanto à ausência de elementos removidos da interface quanto à execução de testes em ordem inadequada.

Nessa versão houve um aumento expressivo de falhas do tipo T4 (Elemento não visível), que registrou 57 ocorrências, sendo 55 delas relacionadas à categoria C7. Esse crescimento mostra que muitos testes estão falhando devido ao tempo insuficiente para que os elementos apareçam na tela ou por problemas de carregamento assíncrono.

O tipo T1 (AssertionError) teve uma queda no número total, com 27 ocorrências, mas continua presente de forma relevante. Ele está distribuído entre as categorias C2 (16), C1 (3), C4 (4) e C5 (4), indicando falhas ligadas tanto à dependência de testes quanto

a mudanças no conteúdo e nos identificadores de elementos, impactando as asserções realizadas nos testes.

Já o tipo T2 (Timeout em requisição HTTP) aparece com 13 falhas, sendo a maioria relacionado à categoria C7 (11), o que também remete à instabilidade no tempo de resposta da aplicação. O T5 (Falha ao preencher campo) e T6 (Erro de execução JS) registraram 8 e 3 falhas, respectivamente, com causas atribuídas às categorias C4, C7 e C2. Os tipos T7 e T8, por fim, tiveram apenas 1 ocorrência cada, sendo que T8 foi associado à categoria C3 (dados incorretos) e T7 à C1 (mudança de ID).

6.4 Objetivo 4: Analisar regressão entre as versões

Com os dados coletados, foi possível ter uma visão geral do comportamento do sistema, podendo ser feita uma análise de possíveis regressões no sistema, o que pode ser visto na subsecção a seguir.

6.4.1 Qual a quantidade de testes que passavam na versão anterior e começaram a falhar na próxima?

A figura 29 resume o comportamento dos testes em todas as versões, com o objetivo de identificar possíveis regressões.

Figura 29 – Resultado dos testes em todas as versões

	Quantidade de testes
Passou sempre	222
Passou e falhou	78
Falhou sempre	107
Falhou e passou	19
Passou e falhou de forma variada	11
Total	437

Fonte: Elaborado pela autora, 2025.

A primeira coluna descreve o comportamento dos testes e a segunda apresenta a quantidade correspondente, sendo o seu significado descrito a seguir:

- Passou sempre: Representa os testes que tiveram sucesso em todas as execuções ao longo das versões analisadas.
- Passou e falhou: Refere-se a testes que inicialmente passaram, mas falharam em versões posteriores.
- Falhou sempre: São testes que apresentaram falhas em todas as versões testadas.

- Falhou e passou: Refere-se a casos em que os testes falharam inicialmente e passaram em versões posteriores.
- Passou e falhou de forma variada: Indica testes que oscilaram entre sucesso e falha sem um padrão fixo.

Com base na tabela apresentada, foram executados 437 testes no total, dos quais 222 passaram em todas as execuções. No entanto, 78 testes que haviam passado anteriormente falharam em versões subsequentes, além disso, 19 testes apresentaram comportamento inverso, falhando inicialmente e passando depois. Outros 11 testes passaram e falharam de forma variada ao longo das execuções, esses dados somam 108 testes com comportamento instável, o que representa aproximadamente 24,7% do total.

Esses comportamentos podem indicar uma certa regressão nas funcionalidades do sistema, entretanto, com base nos dados coletados, não é possível afirmar com certeza que houve regressão, pois grande parte das falhas observadas esteve relacionada à dependência entre testes e à variação no tempo de execução, o que não indica necessariamente que houve falha nas funcionalidades da aplicação, mas sim aspectos externos que podem ter influenciado nos resultados dos testes.

6.5 Discussões

Com a análise das três versões do sistema RegPet, foi possível ter uma melhor compreensão sobre o comportamento das falhas dos testes ao longo do tempo, principalmente no contexto de evolução contínua da aplicação. Como foi visto, as categorias de falhas mais recorrentes nas três versões do sistema, foram:

1. C2 - Dependência entre testes: 237 ocorrências;
2. C7 - Variação de tempo: 92 ocorrências;
3. C8 - Remoção de elemento: 66 ocorrências.

A grande ocorrência de falhas do tipo C2 mostra que há uma fragilidade estrutural na construção dos próprios testes, o que pode comprometer sua confiabilidade como instrumentos de validação. Construir cenários de testes com essa dependência torna os testes suscetíveis ao estado do ambiente, como se há dados já cadastrados ou não, ou à ordem de execução, o que limita sua eficácia em contextos de execução paralela e dificultando a detecção precisa de falhas reais no sistema.

A alta incidência da categoria C7 (Variação de tempo) evidencia que muitos testes falham devido à instabilidade temporal na interface do sistema. Essa categoria está relacionada à falhas relacionadas ao carregamento assíncrono de elementos, à demora nas respostas da aplicação ou à ausência de sincronização adequada entre o teste e a interface.

Já a categoria C8 (Remoção de elemento) representa as falhas causadas por tentativas de acessar componentes que já foram removidos da aplicação em versões mais recentes. Entre as versões da aplicação houve um crescimento dessa categoria o que indica que o sistema passou por mudanças estruturais na interface, mas os testes não foram devidamente atualizados para refletir essas alterações. O que mostra um desafio comum ao realizar testes em ambientes de desenvolvimento ágil e evolução contínua, que é a manutenção da suíte de testes automatizados.

Com relação a regressão, embora o comportamento instável de cerca de 25% dos testes possa levantar a hipótese de regressão funcional, a análise das causas mostra que não há evidência concreta de que funcionalidades tenham sido corrompidas. Uma vez que a maioria das falhas decorre de fatores técnicos e estruturais nos testes, como sincronização inadequada e uso de elementos que foram removidos.

Portanto, é necessário cautela ao interpretar falhas em testes automatizados como indicativos diretos de regressão. A confiabilidade dos testes depende não apenas da execução automatizada, mas também da sua qualidade como artefatos de verificação, incluindo clareza, independência, manutenção contínua e alinhamento com as funcionalidades reais da aplicação.

7 CONCLUSÃO

A constante evolução das aplicações web, embora essencial para atender às demandas dos usuários, enfrenta vários desafios, entre os quais se destaca a automação de testes E2E para validação da interface e funcionalidades do sistema. Este trabalho teve como objetivo compreender através de um estudo de caso, como as mudanças impactam na estabilidade dos testes E2E.

A partir da aplicação de uma metodologia, que envolveu a implantação local de diferentes versões do sistema, a geração de massas de dados e a análise dos resultados obtidos, foi possível não apenas identificar falhas recorrentes, mas também relacionar essas falhas com as categorias de falhas mais comuns dos testes. Com os dados levantados foi possível categorizar os tipos de falhas, examinar seus logs e compreender em que medida elas resultaram de mudanças legítimas na interface ou de fragilidades estruturais nos próprios testes.

Com o trabalho, pode ser observado que uma parcela significativa das falhas não decorre da perda de funcionalidades, mas sim da forma como os testes são sensíveis a alterações superficiais, como mudanças em identificadores de elementos, reorganização visual ou variações no conteúdo textual da interface.Essas falhas, embora não representem necessariamente defeitos no sistema, geram ruído no processo de verificação e aumentam o tempo de manutenção das suítes de teste.

Além disso, também foi visto como a boa construção dos próprios testes é uma etapa fundamental para garantir que falsos positivos não ocorram, com isso, adotar técnicas que promovam testes mais robustos e menos sensíveis a mudanças superficiais na interface é essencial para garantir a confiabilidade da automação.

No campo prático, os resultados da pesquisa podem contribuir para o aperfeiçoamento do processo de testes no RegPet, proporcionando maior previsibilidade sobre o que deve ser melhorado para evitar que as falhas nos testes continuem a acontecer e permitindo uma melhor alocação de esforços na correção de falhas realmente relevantes. Já no campo acadêmico, o trabalho reforça a necessidade de se aprofundar os estudos sobre a fragilidade dos testes de GUI e sua relação com a manutenção evolutiva dos sistemas, um tema ainda pouco explorado na literatura.

Como possibilidades de trabalhos futuros, é recomendado a ampliação da análise para um conjunto mais diverso de aplicações web, a investigação de técnicas de detecção automática de falhas falsas (falsos positivos) e também seria relevante explorar soluções para essas falhas mais comuns.Em suma, este estudo demonstra a importância da construção efetiva e independente de cada teste, assim como a necessidade dos testes automatizados de frontend serem capazes de evoluir com a aplicação.

REFERÊNCIAS

ALI, H. M.; HAMZA, M. Y.; RASHID, T. A. A comprehensive study on automated testing with the software lifecycle. *The Journal of Duhok University*, v. 26, p. 613–620, 12 2023. ISSN 18127568. Disponível em: ⟨https://journal.uod.ac/index.php/uodjournal/article/view/3098⟩.

ALIAN, P. et al. A feature-based approach to generating comprehensive end-to-end tests. In: *arXiv preprint arXiv:2408.01894*. [S.l.: s.n.], 2024.

BSTQB. *Certified Tester Foundation Level Syllabus CTFL 4.0*. [S.l.], 2023. Acesso em: 1 jun. 2025. Disponível em: ⟨https://bstqb.online/files/syllabus_ctfl_4.0br.pdf⟩.

COPPOLA, R.; MORISIO, M.; TORCHIANO, M. Maintenance of android widget-based gui testing: A taxonomy of test case modification causes. In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. [S.l.]: IEEE, 2018. p. 151–158. ISBN 978-1-5386-6352-3.

Cucumber Open Source Project. *Cucumber Documentation*. 2025. Acesso em: 1 jun. 2025. Disponível em: ⟨https://cucumber.io/docs/⟩.

Cypress.io. *Why Cypress?* 2025. ⟨https://docs.cypress.io/app/get-started/why-cypress⟩. Acesso em: 1 jun. 2025.

Docker. *Docker overview*. 2025. ⟨https://docs.docker.com/get-started/docker-overview/⟩. Acesso em: 2 jun. 2025.

Git. *Getting Started - About Version Control*. 2025. ⟨https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control⟩. Acesso em: 2 jun. 2025.

HARROLD, M. J.; ORSO, A. Retesting software during development and maintenance. In: *2008 Frontiers of Software Maintenance*. [S.l.]: IEEE, 2008. p. 99–108. ISBN 978-1-4244-2654-6.

IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. 1990. 1–84 p. Acesso em: 2 jun. 2025. Disponível em: ⟨https://www.informatik.htw-dresden.de/~hauptman/SEI/IEEE_Standard_Glossary_of_Software_Engineering_Terminology%20.pdf⟩.

JORGENSEN, P. C. *Software Testing: A Craftsman's Approach*. 4th. ed. Boca Raton, FL, USA: CRC Press, 2013. Disponível em: ⟨https://malenezi.github.io/malenezi/SE401/Books/Software-Testing-A-Craftsman-s-Approach-Fourth-Edition-Paul-C-Jorgensen.pdf⟩. Disponível em: ⟨https://malenezi.github.io/malenezi/SE401/Books/Software-Testing-A-Craftsman-s-Approach-Fourth-Edition-Paul-C-Jorgensen.pdf⟩.

LEUNG, H.; WHITE, L. Insights into regression testing (software testing). In: *Proceedings. Conference on Software Maintenance - 1989*. [S.l.]: IEEE Comput. Soc. Press, 1989. p. 60–69. ISBN 0-8186-1965-1.

NETO, A.; DIAS, C. Introdução a teste de software. *Engenharia de Software Magazine*, n. 1, p. 22, 2007. Disponível em: ⟨https://www.researchgate.net/...⟩ Disponível em: ⟨https://www.researchgate.net/profile/Arilo-Neto/publication/

266356473_Introducao_a_Teste_de_Software/links/5554ee6408ae6fd2d821ba3a/Introducao-a-Teste-de-Software.pdf⟩.

Node.js Contributors. *Introduction to Node.js*. [S.l.], 2025. Acesso em: 2 jun. 2025. Disponível em: ⟨https://nodejs.org/en/learn/getting-started/introduction-to-nodejs⟩.

OLIVEIRA, R.; DELAMARO, M.; NUNES, F. Oráculos de teste para domínios gui: Uma revisão sistemática. In: *Anais do III Workshop Brasileiro de Teste de Software Sistemático e Automatizado (SAST)*. [S.l.: s.n.], 2009.

PEZZè, M.; YOUNG, M. *Teste e análise de software: processos, princípios e técnicas*. Porto Alegre: Bookman, 2008. ISBN 978-85-7780-262-3.

PRADHAN, L. *User Interface Test Automation and its Challenges in an Industrial Scenario*. Dissertação (Mestrado) — Mälardalen University, 01 2012.

RegPet. *RegPet*. 2025. ⟨https://regpet.pb.gov.br⟩. Acesso em: 2 jun. 2025.

RIOS, E.; FILHO, T. M. *Teste de Software*. Rio de Janeiro: Alta Books, 2013. ISBN 9788576087755.

SILVA, M. L. M. da; DALLILO, F. D. Uma visão geral sobre automação de testes. *Revista Científica Multidisciplinar Núcleo do Conhecimento*, p. 117–130, 12 2019. ISSN 24480959.

SOMMERVILLE, I. *Engenharia de Software*. 9. ed. São Paulo: Pearson Prentice Hall, 2011. Acesso em: 2 jun. 2025. ISBN 9788576056942. Disponível em: ⟨https://www.facom.ufu.br/~william/Disciplinas%202018-2/BSI-GSI030-EngenhariaSoftware/Livro/engenhariaSoftwareSommerville.pdf⟩.

TSAI, W. et al. End-to-end integration testing design. In: *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*. [S.l.]: IEEE, 2006. p. 166–171. ISBN 0-7695-1372-7.