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

JOSÉ VINÍCIUS SILVA ALVES

# IMPLEMENTAÇÃO E AUTOMAÇÃO DE TESTES: CASO PRÁTICO DE PROJETO EM AMBIENTE DE PRODUÇÃO

CAMPINA GRANDE - PB
2025

**JOSÉ VINÍCIUS SILVA ALVES**

# IMPLEMENTAÇÃO E AUTOMAÇÃO DE TESTES: CASO PRÁTICO DE PROJETO EM AMBIENTE DE PRODUÇÃO

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

**Área de concentração**: Engenharia de Software

**Orientador**: Prof. Dr. Fábio Luiz Leite Júnior

**CAMPINA GRANDE - PB**

**2025**

A474i Alves, José Vinícius Silva.
Implementação e automação de testes [manuscrito] : caso prático de projeto em ambiente de produção / José Vinícius Silva Alves. - 2025.
51 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. Dr. Fábio Luiz Leite Júnior, Departamento de Computação - CCT".

1. Automação de testes. 2. Testes de software. 3. Microsserviços. I. Título

21. ed. CDD 005.427

Elaborada por Lêda Cristina Diniz Andrade - CRB - 15/1032 BC

JOSE VINICIUS SILVA ALVES

# IMPLEMENTAÇÃO E AUTOMAÇÃO DE TESTES: CASO PRÁTICO DE PROJETO EM AMBIENTE DE PRODUÇÃO

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

Aprovada em: 12/06/2025.

BANCA EXAMINADORA

Documento assinado eletronicamente por:

- **Janderson Jason Barbosa Aguiar** (***.765.854-**), em **25/06/2025 21:34:45** com chave **5bc932da522511f082172618257239a1**.
- **Fábio Luiz Leite Júnior** (***.848.564-**), em **25/06/2025 19:05:54** com chave **906946f2521011f0bf071a7cc27eb1f9**.
- **Luciana de Queiroz Leal Gomes** (***.309.554-**), em **26/06/2025 08:51:14** com chave **dc830a78528311f097a61a7cc27eb1f9**.

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:** 26/06/2025
**Código de Autenticação:** 984c4c

Dedico a Deus por me guiar nessa trajetória e aos meus pais por todo apoio, esforço e incentivo.

## AGRADECIMENTOS

Gostaria, em primeiro lugar, de agradecer a Deus, pela presença constante em minha vida, por guiar meus passos e abençoar cada etapa da minha jornada. Sua luz foi fundamental para que eu pudesse superar todos os desafios ao longo do caminho.

Agradeço à minha mãe Magna e ao meu pai Jailson, por todo o esforço, dedicação e sacrifícios realizados para que eu chegasse até aqui. Sou profundamente grato por estarem sempre ao meu lado, tanto nos momentos de alegria quanto nas adversidades, oferecendo apoio incondicional e palavras de encorajamento.

Estendo meus agradecimentos ao meu irmão João Pedro e a todos os demais familiares que, de alguma forma, contribuíram e me apoiaram durante essa trajetória.

Aos amigos que estiveram comigo ao longo dessa jornada acadêmica, agradeço pelo apoio e companheirismo.

Por fim, agradeço ao campus I da Universidade Estadual da Paraíba (UEPB) pela oportunidade de estudo, e a todos os professores, em especial ao professor Fábio Leite, que contribuíram com seu conhecimento, dedicação e incentivo, tornando essa experiência ainda mais enriquecedora e significativa.

**RESUMO**

O cenário atual de desenvolvimento de software tem enfrentado crescentes demandas por qualidade e confiabilidade, especialmente diante da transformação digital e da necessidade de entregas rápidas. Nesse cenário, os testes de software são fundamentais, transformando-se em um processo contínuo e integrado ao ciclo de desenvolvimento, em vez de consistirem apenas em uma etapa final. Este trabalho apresenta a implementação e automação de testes funcionais no Sistema de Controle e Acompanhamento da Malha Fiscal (SCAMF), durante a migração da sua arquitetura monolítica para microsserviços, com o propósito de elevar e garantir a qualidade do sistema. A abordagem proposta adotou um conjunto de ferramentas como Cypress e Cucumber na automação dos testes, Jenkins para integração contínua e Jira para gerenciamento e rastreabilidade dos resultados. Os resultados demonstram que a automação permitiu a detecção precoce de falhas, redução de tempo na execução dos testes e maior eficiência na correção de comportamentos indesejados. Além disso, a integração com o Jira proporcionou transparência e colaboração entre a equipe no processo de realização dos testes no sistema, consolidando-se como uma estratégia eficaz para garantir a qualidade do software em ambientes de produção.

**Palavras-chave:** automação de testes; testes de software; microsserviços.

**ABSTRACT**

The current software development landscape has been facing increasing demands for quality and reliability, especially in the context of digital transformation and the need for rapid deliveries. In this scenario, software testing is fundamental, evolving into a continuous process integrated into the development cycle, rather than consisting solely of a final stage. This work presents the implementation and automation of functional tests in the Fiscal Network Control and Monitoring System (SCAMF) during its migration from a monolithic architecture to a microservices-based architecture, ensuring system quality. The proposed approach utilized a set of tools, including Cypress and Cucumber for test automation, Jenkins for continuous integration, and Jira for managing and tracking results. The results demonstrate that automation enabled early detection of failures, reduced test execution time, and improved efficiency in correcting undesirable behaviors. Furthermore, the integration with Jira provided transparency and collaboration among the team during the system testing process, establishing itself as an effective strategy to ensure software quality in production environments.

**Keywords:** test automation; software testing; microservices.

# LISTA DE ILUSTRAÇÕES

## SUMÁRIO

Página

# 1 INTRODUÇÃO

Nas últimas décadas, observou-se um crescimento gigantesco na descoberta de novas tecnologias e no desenvolvimento tecnológico, impulsionado por uma necessidade crescente por soluções digitais mais robustas, sofisticadas e personalizadas em diferentes contextos (Schwab, 2016). Os avanços tecnológicos têm se manifestado em diversos setores, como comunicação, mobilidade urbana, saúde, educação, finanças e administração pública (Manyika et al., 2016). Com isso, a transformação digital passou a ser um requisito fundamental para organizações, sejam elas públicas ou privadas, que buscam acompanhar as rápidas mudanças sociais e econômicas. Nesse contexto, o desenvolvimento de software desempenha um papel estratégico ao viabilizar soluções inovadoras, eficientes e escaláveis, capazes de atender a diferentes contextos com agilidade e precisão.

Diante da necessidade de acompanhar esse cenário de transformação acelerada, emergiram novas abordagens de desenvolvimento de software, fundamentadas em ciclos incrementais de especificação, implementação e entrega (Pressman; Maxim, 2021). Essas abordagens mostraram-se mais adequadas em projetos com requisitos dinâmicos e sujeitos a mudanças constantes, consolidando o que atualmente se conhece como métodos ágeis (Sommerville, 2019). Tais métodos estão alinhados aos princípios do Manifesto Ágil (Fowler; Highsmith, 2001).

Com o desenvolvimento baseado em métodos ágeis, tornou-se necessário revisar as abordagens de garantia de qualidade, atribuindo aos testes de software um papel fundamental no desenvolvimento. Em contextos ágeis, marcados por entregas constantes e mudanças frequentes nos requisitos, os testes deixam de ser uma etapa final e passam a fazer parte de todo o fluxo de trabalho, sendo executados de forma contínua (Crispin; Gregory, 2009)

Independentemente da metodologia de desenvolvimento escolhida, os testes de software constituem uma etapa fundamental para garantir a qualidade, confiabilidade e segurança dos sistemas. Como destacado por Myers, Badgett e Sandler (2012), o processo de teste tem como objetivo principal a identificação de erros, e a identificação e remoção desses erros aumentam a confiabilidade no sistema.

A ocorrência de erros é um fator comum no desenvolvimento de software, podendo manifestar-se em diversas fases do ciclo de vida do sistema. Como destacam Bernardo e Kon (2008), até mesmo erros que foram identificados e corrigidos podem ressurgir após modificações significativas no código, gerando impactos consideráveis tanto para as equipes de desenvolvimento quanto para os clientes. A propensão ao surgimento de erros está diretamente ligada à condição humana no processo de desenvolvimento, segundo Sommerville (2019), defeitos e falhas em sistemas de software geralmente se originam de equívocos humanos, nos quais os desenvolvedores perdem o controle sobre as complexas

relações entre variáveis e componentes, resultando em comportamentos inesperados do sistema.

O processo de teste de software consiste na execução do sistema sob condições controladas, utilizando dados de entrada simulados para avaliar tanto a conformidade funcional, de acordo com os requisitos especificados, quanto os atributos não funcionais (Sommerville, 2019). Na prática, os testes são organizados em diferentes níveis de verificação, sendo os principais categorizados como testes de unidade, integração, sistema e regressão.

Na indústria de software, os testes são frequentemente realizados de maneira manual, nos quais um testador executa casos de teste e compara os resultados obtidos com o comportamento esperado. Segundo Bernardo e Kon (2008), a execução de testes manuais pode ser ágil e eficiente para um caso de teste, mas a aplicação repetitiva de uma bateria extensa de casos de teste através de métodos manuais revela-se uma atividade pesada e propensa a fadiga, e quando essa metodologia é aplicada exclusivamente ao final de módulos ou do sistema completo, frequentemente resulta em identificação tardia de erros, atraso na entrega e dificuldade em assegurar a evolução e manutenção do software. Diante desses problemas e da necessidade de desenvolvimento de software cada vez mais ágil, surge a abordagem da automação de testes.

A automação de testes manifesta-se transformando a garantia de qualidade em um processo contínuo e escalável. Diferentemente dos testes manuais, que dependem da intervenção humana para cada execução, os testes automatizados são implementados como *scripts* reutilizáveis que podem ser executados sob demanda (Crispin; Gregory, 2009). Além de promover a garantia de qualidade, a automação dos testes funciona como facilitador essencial para implementação de mudanças frequentes com maior segurança, minimizando significativamente a ocorrência de regressões no sistema (Bernardo; Kon, 2008).

## 1.1 Motivação

O estudo apresentado neste trabalho tem como foco o Sistema de Controle e Acompanhamento da Malha Fiscal (SCAMF), desenvolvido para auxiliar o trabalho dos fiscais no processo de fiscalização tributária das empresas. O sistema concentra e organiza as inconsistências tributárias, apresentando-as por meio de interfaces acessíveis e intuitivas, tanto para a equipe de fiscalização quanto para as empresas contribuintes do Estado da Paraíba. Inicialmente desenvolvido em uma arquitetura monolítica, o SCAMF vem passando por um processo de reestruturação para a adoção de uma arquitetura baseada em microsserviços, decisão proveniente de mudanças nos requisitos. Nesse cenário, surgiu a necessidade de implementação de testes automatizados como estratégia para garantir a qualidade e a confiabilidade das funcionalidades desenvolvidas.

## 1.2 Objetivo

Diante do contexto abordado, o presente trabalho tem como objetivo geral implementar a integração e automação de testes funcionais, tendo como propósito elevar a qualidade do sistema mediante a detecção precoce de falhas presentes nas funcionalidades desenvolvidas. Para isso, propõem-se os seguintes objetivos específicos:

- (1) Desenvolver *scripts* de testes automatizados abrangendo as funcionalidades.
- (2) Criar um fluxo de integração e automação de testes replicável.
- (3) relatar o impacto da integração e automação dos testes funcionais no processo de desenvolvimento de software.

## 1.3 Estrutura do Trabalho

Este trabalho está organizado em seis capítulos: o capítulo 1 refere-se a introdução que contextualiza o tema e apresenta a motivação e os objetivos do trabalho; o capítulo 2 refere-se a fundamentação teórica que consolida os conceitos que embasam o trabalho; o capítulo 3 refere-se a metodologia que detalha a abordagem adotada; o capítulo 4 descreve a implementação da integração e automação dos testes para o sistema SCAMF; o capítulo 5 refere-se a discussão dos resultados obtidos com a automação; e, por fim, o capítulo 6 refere-se à conclusão que sintetiza o trabalho.

# 2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, delimita-se a base teórica que sustenta as discussões desenvolvidas ao longo deste trabalho. São abordados os conceitos relacionados à qualidade e à arquitetura de software, bem como os princípios dos testes de software, com ênfase nos testes funcionais e na automação de testes, os quais são essenciais para o escopo do trabalho.

## 2.1 Qualidade de Software

A qualidade de software é um conceito complexo que engloba tanto aspectos técnicos quanto gerenciais. Pressman e Maxim (2021) definem qualidade de software como uma gestão de qualidade estruturada e efetiva, orientada para a criação de um produto que seja útil e gere valor real. Essa abordagem incorpora tanto mecanismos de controle organizacional quanto práticas de engenharia de software que possibilitam a análise consistente de problemas e a proposição de soluções. O software deve não apenas atender aos requisitos funcionais explicitamente definidos, mas também satisfazer expectativas implícitas, como usabilidade e confiabilidade. Além disso, precisa garantir a ausência de erros, gerando valor tanto para as organizações desenvolvedoras, por meio da redução de custos com manutenção, suporte técnico e retrabalho, quanto para os usuários finais, através da agilidade em processos de negócio.

Duarte e Falbo (2000) complementam essa visão ao enfatizar que a qualidade deve ser entendida como um conjunto de características que precisam atingir determinados níveis para satisfazer necessidades explícitas e implícitas dos usuários. Além disso, ressaltam que a qualidade em um software deve ser construída através de processos de desenvolvimento bem estruturados.

Nessa relação entre qualidade do produto e qualidade do processo, Sommerville (2019) traz a perspectiva de que, diferentemente da indústria manufatureira, o desenvolvimento de software é uma atividade criativa, não mecânica, o que torna as habilidades e conhecimentos individuais dos desenvolvedores elementos cruciais. Além disso, questões externas, como o grau de inovação de uma aplicação ou pressões comerciais por lançamentos antecipados, podem impactar diretamente a qualidade do produto final, independentemente da metodologia adotada.

## 2.2 Testes de Software

Segundo Delamaro, Maldonado e Jino (2007), O teste de software é uma atividade dinâmica que se concentra em executar um programa ou modelo com determinados dados de entrada e observar se seu comportamento corresponde ao que se espera. Quando os resultados obtidos diferem dos especificados, considera-se que um erro ou defeito foi encontrado. Nesse sentido, Myers, Badgett e Sandler (2012) destacam que o software deve agir de forma consistente e previsível, evitando comportamentos inesperados que possam

surpreender os usuários. Para alcançar esse nível de qualidade e confiabilidade, os testes de software constituem uma etapa fundamental.

Conforme Sommerville (2019), os testes de software integram um processo mais abrangente de verificação e validação. Nesse contexto, a verificação tem como objetivo principal assegurar que o sistema atenda estritamente aos requisitos funcionais e não-funcionais especificados, enquanto a validação busca garantir que o software realmente atende às necessidades operacionais e expectativas do cliente e dos usuários.

De acordo com Delamaro, Maldonado e Jino (2007), o processo de teste de software é organizado em diferentes fases, cada uma com objetivos específicos para garantir a qualidade durante o desenvolvimento e manutenção do sistema. Essas fases compreendem os testes de unidade, integração, sistema e regressão.

- Teste de Unidade: concentra-se em testar os componentes menores do software, como funções, métodos ou classes individuais. Nesta etapa, busca-se identificar erros relacionados a implementação incorreta de algoritmos, manipulação inadequada de estruturas de dados e falhas pontuais de codificação (Delamaro; Maldonado; Jino, 2007).

- Teste de Integração: verifica se a interação entre os módulos integrados ocorre conforme especificado, identificando como módulos individuais se comportam em conjunto (Pressman; Maxim, 2021).

- Teste de Sistema: o software é avaliado em sua totalidade, com a intenção de validar se todas as funcionalidades atendem aos requisitos especificados e verificar os atributos não funcionais essenciais, como desempenho, segurança e robustez em condições operacionais realistas (Delamaro; Maldonado; Jino, 2007).

- Teste de Regressão: consiste em repetir um conjunto de testes previamente realizados com o objetivo de verificar se modificações feitas no software não causaram efeitos indesejados em funcionalidades que antes funcionavam corretamente (Pressman; Maxim, 2021).

Além dos testes citados, existe o teste de aceitação. É nele que o software é avaliado por usuários finais ou clientes para garantir que atenda aos requisitos de negócio e esteja pronto para implantação (Sommerville, 2019).

### 2.2.1 Teste caixa-preta e caixa-branca

Segundo Myers, Badgett e Sandler (2012), as duas estratégias mais comuns de elaboração de testes consistem nos testes de caixa-preta e de caixa-branca.

O teste de caixa-preta caracteriza-se por não se preocupar com a estrutura interna ou lógica de implementação do software (Pressman; Maxim, 2021). Nessa abordagem, o

foco está na verificação dos requisitos funcionais a partir das entradas e saídas do sistema, analisando se as saídas produzidas estão de acordo com os resultados esperados definidos pelos requisitos funcionais. Os dados de teste são baseados nas especificações do software, tornando essa abordagem interessante para validar a integração entre componentes e sistemas distintos (Myers; Badgett; Sandler, 2012).

Diferentemente do teste de caixa-preta, o teste de caixa-branca se baseia na estrutura interna do software para construir os casos de teste. O foco é desenvolver os testes de maneira que garantam a execução de todo o caminho lógico implementado no software (Sommerville, 2019). No entanto, devido à quantidade exorbitante de caminhos lógicos que um software pode ter, torna-se impraticável gerar casos de teste para cobrir o software como um todo (Pressman; Maxim, 2021).

### 2.2.2 Automação de Testes

De acordo com Bernardo e Kon (2008), embora a execução de testes manuais possa ser ágil e eficiente para poucos casos de teste, sua repetição frequente e em grandes volumes se torna uma tarefa custosa, sujeita ao cansaço humano. Diante desse cenário, surge o processo de automação de testes.

A automação de testes consiste no processo de desenvolvimento de testes automatizados. Os testes automatizados são *scripts* ou programas especializados que simulam a interação com o software, executam suas funcionalidades e realizam comparações automáticas entre os resultados obtidos e os comportamentos esperados (Bernardo; Kon, 2008). Como se trata de um processo automatizado, é possível manter um grande conjunto de testes, capaz de ser executado de maneira consistente, eficiente e com o mínimo de intervenção humana. Desse modo, a cada modificação ou inclusão de funcionalidades no sistema, o conjunto de testes pode ser executado, permitindo a identificação imediata de possíveis inconsistências ou regressões introduzidas pelas alterações recentes no código (Sommerville, 2019).

Além disso, como destacam Crispin e Gregory (2009), a automação de testes serve como documentação do software, garantindo que o comportamento do sistema esteja de acordo com o especificado. Quando ocorrem mudanças nos requisitos ou funcionalidades, os testes automatizados devem ser ajustados para refletir as novas especificações. Essa característica é fundamental quando se usa metodologias ágeis, onde as mudanças acontecem com certa frequência e a documentação tende a se tornar obsoleta.

### 2.2.3 Ferramentas de Integração e Automação de Testes

De acordo com Alferidah e Ahmed (2020), a criação de ferramentas para automação de testes apresenta vários obstáculos significativos, como a exigência de alto desempenho, a simplicidade de uso e a efetividade na identificação de problemas. Elas podem ser ca-

racterizadas conforme seu método de funcionamento, estágio do ciclo de desenvolvimento ou propósito específico.

Cada tipo de teste de software requer sua respectiva ferramenta, sendo que algumas ferramentas são versáteis o suficiente para atender a múltiplos tipos de teste. Atualmente, essas tecnologias estão facilmente acessíveis no mercado, adaptáveis a diversos ambientes de criação de software (Alferidah; Ahmed, 2020). Essas ferramentas aceleram a execução de testes e melhoram a precisão e a confiabilidade dos resultados. Quando associadas às ferramentas de integração contínua, tornam-se componentes essenciais do ciclo de desenvolvimento, principalmente em projetos que utilizam metodologias ágeis (Crispin; Gregory, 2009).

Nesse sentido, para o processo de integração, gerenciamento e automação de testes descrito neste trabalho, foram escolhidas as ferramentas Cypress, Cucumber, Jenkins e Jira.

O Cypress é um framework de teste front-end, voltado principalmente para lidar com os desafios de testar aplicações web modernas. Ele permite testar tudo que é executado em um navegador, fornecendo informações sobre o conjunto de testes e a qualidade do sistema. Diferentemente de outras ferramentas, o Cypress suporta diferentes tipos de testes, como o teste de ponta-a-ponta, teste de acessibilidade, teste de componentes e cobertura de interface (Cypress.io, 2025).

O diferencial marcante do Cypress, em relação à maioria das outras ferramentas semelhantes, está em sua arquitetura. A execução do Cypress ocorre no mesmo loop de eventos do navegador, o que elimina a necessidade de drivers externos. Isso proporciona testes mais rápidos, consistentes e com maior controle sobre o comportamento da aplicação (Cypress.io, 2025).

Além das características citadas, é possível adotar padrões, como o Page Object Model (POM), no Cypress. O POM cria uma abstração eficiente ao separar a estrutura das páginas web dos casos de teste, reduzindo o acoplamento e facilitando a reutilização de código, esses fatores colaboram para a redução dos custos de manutenção a longo prazo (Mobaraya; Ali, 2019).

O Cucumber consiste em uma ferramenta para testes automatizados, baseada nos princípios do Desenvolvimento Orientado ao Comportamento (Behavior Driven Development - BDD). O diferencial do Cucumber é sua capacidade de interpretar especificações funcionais escritas em linguagem natural e transformá-las em casos de teste executáveis. Essa característica permite validar se o comportamento real do software está em conformidade com o especificado em sua documentação (Cucumber.io, 2025b).

O conjunto de regras gramaticais que possibilitam a escrita em linguagem natural é denominado de Gherkin. O Gherkin emprega palavras-chave padronizadas para facilitar o entendimento e possibilitar a execução das especificações, documentando o comportamento do sistema de maneira indireta (Cucumber.io, 2025b).

O BDD, por sua vez, elimina as barreiras de comunicação entre as áreas técnicas e de negócio através da colaboração entre as diferentes áreas para o entendimento do problema em questão, desenvolvimento em iterações rápidas com validação contínua e da documentação que reflete o comportamento do sistema (Cucumber.io, 2025a).

O Jenkins é um servidor de automação de código aberto e independente, amplamente utilizado para proporcionar a automatização dos mais diversos tipos de tarefas. Ele permite a automação de todo o fluxo de desenvolvimento, desde a construção até a implantação contínua, incluindo a execução automatizada de testes (Jenkins.io, 2025b).

A existência de uma variedade de plugins permite a integração com diversas ferramentas de desenvolvimento. Um exemplo disso é o plugin AIO Tests, que possibilita reportar os resultados dos testes automatizados executados para sistemas de gerenciamento de projetos como o Jira (Jenkins.io, 2025a).

O Jira é uma plataforma de gestão de projetos, amplamente utilizada por equipes que buscam planejar, monitorar e entregar softwares de alta qualidade. Sua capacidade de centralizar informações em todas as etapas do ciclo de desenvolvimento permite que as equipes tenham visibilidade completa sobre suas atividades, facilitando a tomada de decisões rápidas e o alinhamento com os objetivos estratégicos do negócio (Atlassian, 2025).

Uma das principais vantagens do Jira é o suporte a metodologias ágeis, como Scrum e Kanban, oferecendo recursos personalizáveis para planejamento, execução e medição de desempenho. Atualmente, o Jira é muito utilizado pelas empresas devido à sua flexibilidade, capacidade de adaptação a diferentes tipos de projetos e integração com diversos aplicativos e extensões, como o AIO tests (Atlassian, 2025).

## 2.3 Arquitetura de Software

Segundo Garlan (2008), a arquitetura de software representa a estrutura fundamental de um sistema, abrangendo todos os seus componentes, as propriedades visíveis externamente e as relações estabelecidas entre eles. Complementando essa visão, Jaiswal (2019) ressalta que a definição arquitetural envolve decisões estratégicas cruciais sobre a organização do sistema, onde cada escolha realizada influencia diretamente atributos como qualidade, desempenho, manutenabilidade e o sucesso do produto de software.

A arquitetura de software liga os objetivos organizacionais e a implementação concreta do sistema, garantindo que a solução desenvolvida atenda às necessidades de negócio (Bass; Clements; Kazman, 2022). Nesse contexto, das diversas abordagens existentes, destacam-se a arquitetura monolítica e a arquitetura baseada em microsserviços, cada uma com características distintas que as tornam adequadas para diferentes cenários de desenvolvimento.

### 2.3.1 Arquitetura Monolítica

Segundo Powell e Smalley (2024), a arquitetura monolítica consiste em um modelo clássico de desenvolvimento de software, no qual as funcionalidades de negócio são executadas a partir de uma única base de código.

A característica de possuir uma única base de código, na qual todos os componentes estão altamente acoplados, torna o sistema complexo, dificultando o processo de testes e de realização de alterações. Esse conjunto de particularidades pode levar à construção de sistemas frágeis, que quebram sempre que alguma novidade é implantada (Richards, 2015).

### 2.3.2 Arquitetura de Microsserviços

Segundo Lewis e Fowler (2014), embora não exista uma definição única e precisa para a arquitetura de microsserviços, ela representa uma abordagem contemporânea para o desenvolvimento de software, na qual as aplicações são especificamente estruturadas como conjuntos de serviços independentes e implantáveis de forma autônoma, diferenciando-se das demais abordagens existentes.

De acordo com Richards (2015), as questões de forte acoplamento que tornam as aplicações monolíticas complexas, para atualizar e testar, são resolvidas utilizando a arquitetura de microsserviços no software, através da separação do código da aplicação em diferentes serviços. Com essa separação, é possível desenvolver, testar e implantar o código, sem ligação direta com outros componentes.

Dentre as principais características possibilitadas pela a arquitetura de microsserviços estão a heterogeneidade tecnológica, a resiliência e a escalabilidade. Elas são abordadas por Newman (2015) da seguinte maneira:

- Heterogeneidade Tecnológica: cada serviço pode adotar tecnologias distintas dos demais, permitindo selecionar as ferramentas mais adequadas para cada contexto específico, diferentemente das abordagens monolíticas que comumente exigem padronização tecnológica em todo o sistema.

- Resiliência: mesmo com a ocorrência da falha completa de um serviço, o sistema consegue continuar operando com funcionalidade reduzida, isso demonstra maior robustez e capacidade de recuperação em comparação com a arquitetura monolítica. Em sistemas que usam a arquitetura monolítica, a falha de um componente pode causar a inutilização da aplicação por completo.

- Escalabilidade: em sistemas monolíticos, o escalonamento do sistema ocorre de forma integral, mesmo quando apenas componentes específicos demandam maior capacidade. Na arquitetura baseada em serviços, pode-se escalonar apenas o serviço em específico.

## 2.4 Síntese

A fundamentação teórica construída nesse capítulo serve como base para a metodologia e as discussões desenvolvidas neste trabalho. A qualidade de software é compreendida como uma gestão estruturada e eficaz, visando entregar um produto útil que agregue valor aos seus usuários (Pressman; Maxim, 2021). Esse entendimento evidencia a importância de processos de verificação e validação, como os testes de software, que são essenciais para a garantia da qualidade (Delamaro; Maldonado; Jino, 2007).

Nesse contexto, a automação de testes mostrou-se interessante, especialmente em projetos ágeis (Crispin; Gregory, 2009). A possibilidade de execução repetitiva e consistente dos testes reduz os custos e aumenta a confiabilidade (Bernardo; Kon, 2008). As ferramentas Cypress, Cucumber e Jenkins alinham-se a essas questões, permitindo a integração com sistemas de gestão como o Jira. Essa combinação de tecnologias é relevante para sistemas em transição arquitetural, como o SCAMF, que está migrando de uma arquitetura monolítica para microsserviços.

Esses elementos teóricos não apenas justificam as escolhas metodológicas adotadas no trabalho, mas servem de referência para a análise dos resultados obtidos, como detalhado nos capítulos seguintes.

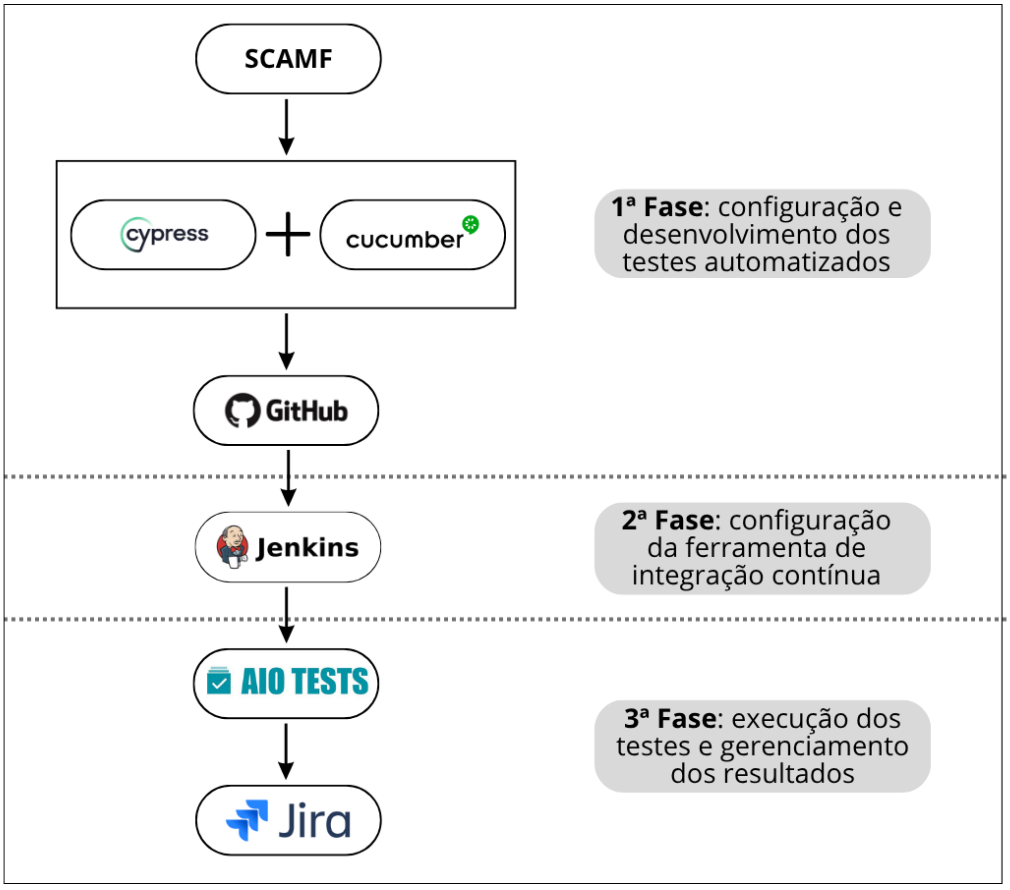
# 3 METODOLOGIA

Este capítulo apresenta a abordagem metodológica desenvolvida para implementar o fluxo de integração contínua e automação dos testes, visando garantir uma melhor qualidade, confiabilidade e rastreabilidade no processo de desenvolvimento de software.

De acordo com Wholin (2021), o estudo de caso em engenharia de software consiste em uma investigação empírica que utiliza diversas fontes de evidência para analisar um fenômeno contemporâneo em seu contexto real. Nesse sentido, este trabalho adota uma abordagem empírica, de natureza descritiva, fundamentada na coleta e análise dos resultados obtidos por meio da implementação de testes automatizados voltados à qualidade do sistema.

A Figura 1 ilustra o fluxo geral da metodologia proposta, destacando a relação entre as três fases e suas respectivas ferramentas. A 1ª fase é a de configuração e desenvolvimento dos testes automatizados utilizando as ferramentas Cypress e Cucumber, a 2ª fase consiste na configuração da ferramenta de integração, que nesse caso é o Jenkins, e a 3ª fase aborda o gerenciamento dos resultados dos testes executados usando o AIO Tests e o Jira.

Figura 1 – Visão geral da abordagem




*Fonte: Elaborado pelo autor, 2025*

## 3.1 Configuração e Desenvolvimento dos Testes Automatizados

O processo de configuração e desenvolvimento dos testes automatizados inicia com a análise das funcionalidades do sistema, onde são identificados fluxos críticos e cenários repetitivos candidatos à automação. Com base nos requisitos do sistema, os casos de teste são especificados utilizando as regras gramaticais do Gherkin, proporcionado pelo Cucumber, permitindo a escrita do comportamento esperado através de linguagem natural.

Para que os cenários escritos sejam executáveis, utilizou-se a ferramenta de testes Cypress, baseada na linguagem Javascript. A estruturação e desenvolvimento dos códigos de teste se baseiam no padrão Page Object Model (POM), organizando o código em classes que representam as páginas da aplicação, encapsulando os elementos de interface e as ações correspondentes, possibilitando a reutilização de código para diferentes cenários.

Os testes foram desenvolvidos com foco na verificação e validação dos fluxos e comportamentos especificados, replicando as ações que um usuário real realizaria durante a utilização do sistema. Cada cenário descrito é mapeado para um teste executável no Cypress, garantindo que o comportamento observado na aplicação corresponda às especificações definidas.

## 3.2 Configuração da Ferramenta de Integração Contínua

O Jenkins foi selecionado como a ferramenta para a implementação da integração contínua. Para a configuração do ambiente, inicialmente criou-se uma tarefa, ou *job*, do tipo estilo livre na interface administrativa do Jenkins. Esse *job* foi especialmente parametrizado para atender aos requisitos do projeto, com ligação direta ao repositório do GitHub onde estão versionados os testes automatizados desenvolvidos com Cypress e Cucumber. O ponto-chave da configuração adotada está na capacidade do *job* ser acionado de maneira remota.

A capacidade de acionamento remoto do *job* permite que, quando instalado em uma máquina virtual, o Jenkins pode ser acessado externamente através da liberação da porta de comunicação e configuração adequada de um domínio. Dessa forma, o Jenkins possibilita que o *job* seja disparado através de requisições web, informando os parâmetros necessários para execução.

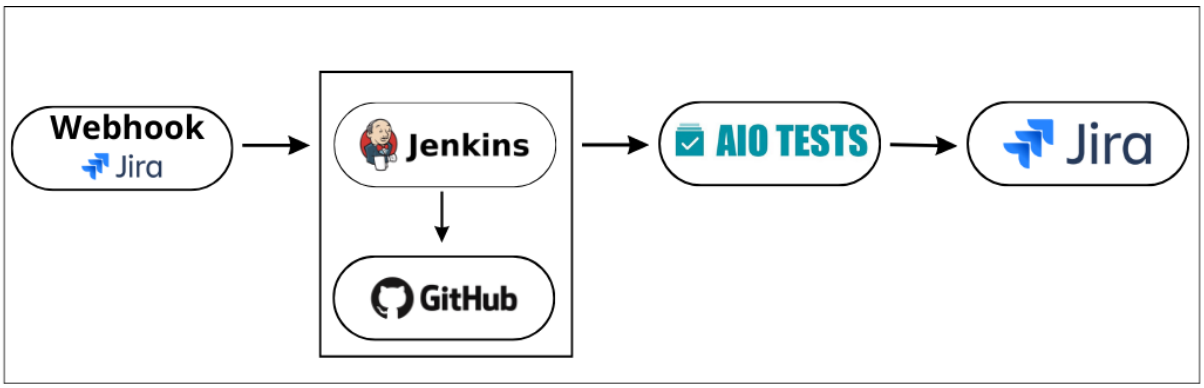
## 3.3 Execução dos Testes e Gerenciamento dos Resultados

A abordagem implementada permite a execução de testes específicos ou conjuntos de testes, conforme as necessidades do projeto, integrando-se ao ambiente de gerenciamento de projetos Jira para acompanhamento dos resultados obtidos. Conforme apresentado na Figura 2, o fluxo inicia com um disparo remoto via webhook configurado no Jira, que envia uma requisição HTTPS ao Jenkins contendo o parâmetro que identifica precisamente

quais testes devem ser executados. O parâmetro utilizado consiste em tags do Cucumber que identificam um ou mais cenários de testes presentes no Apêndice B. Ao receber a requisição, o Jenkins aciona imediatamente o *job* correspondente, garantindo que apenas os testes correspondentes a tag informada sejam executados. Quando acionado, o *job* realiza a clonagem do repositório GitHub onde encontram-se os testes automatizados. Em seguida, faz a instalação de todas as dependências necessárias, configurando o ambiente de execução de maneira correta.

Com o ambiente devidamente configurado, os testes são executados de acordo com o parâmetro enviado na requisição. Ao finalizar a execução, os resultados são gerados e, através do plugin AIO Tests, são enviados ao Jira. Essa integração permite que toda a equipe visualize de forma centralizada o status de cada teste e examine screenshots capturados em caso de falhas.

Figura 2 – Fluxo de execução da abordagem




*Fonte: Elaborado pelo autor, 2025*

Estudos como o de Melo (2024) e o de Guimarães (2016) abordam a execução do *job* referente aos testes automatizados de maneira manual e programada, respectivamente. O principal diferencial da abordagem apresentada nesse trabalho está na capacidade de executar o *job* de forma imediata, por meio de uma requisição, e executar os testes automatizados de forma seletiva, economizando tempo e recursos computacionais. A rastreabilidade completa dos resultados, centralizada no Jira, facilita a análise de erros e o acompanhamento da qualidade ao longo do tempo. Todo o processo, a partir do acionamento do *job*, ocorre de maneira automatizada, eliminando a necessidade de intervenção manual e reduzindo possíveis erros humanos.

# 4 INTEGRAÇÃO E AUTOMAÇÃO DE TESTES FUNCIONAIS NO SCAMF

Este capítulo apresenta a implementação prática da abordagem proposta, no capítulo anterior, no sistema SCAMF. Primeiramente inicia-se com uma visão geral do SCAMF, abordando suas principais funcionalidades e arquitetura, fornecendo o contexto necessário para compreensão da aplicação. Em seguida, demonstra-se como a abordagem de automação e integração contínua foi aplicada ao SCAMF e os resultados obtidos a partir dela.

## 4.1 SCAMF

O Sistema de Controle e Acompanhamento da Malha Fiscal (SCAMF) é uma aplicação web, desenvolvida com o intuito de otimizar o processo de fiscalização tributária realizado pela Secretaria de Estado da Fazenda da Paraíba (SEFAZ-PB), complementando o conjunto de tecnologias utilizadas até então e evitando processos que exigem um grau elevado de conhecimento tecnológico, como a necessidade de acessar diretamente os bancos de dados e extrair os dados necessários para os relatórios fiscais utilizados nas análises.

O sistema consolida as inconsistências tributárias em interfaces, permitindo que os fiscais concentrem seus esforços na análise, exportação e inserção de dados, em vez de se ocuparem com operações e técnicas de consulta.

### 4.1.1 Funcionalidades do SCAMF

O SCAMF conta, atualmente, com diversas funcionalidades em produção, projetadas para otimizar o fluxo de trabalho dos auditores fiscais. Dentre elas, destacam-se:

- Consulta e Exportação de Relatórios de Inconsistências: permite aos fiscais acessar e exportar, em Excel, os relatórios detalhados das inconsistências identificadas. Essa funcionalidade agiliza o encaminhamento formal das inconsistências aos contribuintes, servindo como base para o processo de autuação.

- Homologação de Valores de Malha Fiscal: oferece uma interface para registro e validação dos valores corrigidos de cada item fiscalizado. Os auditores podem inserir os valores homologados avaliando os itens das inconsistências.

- Geração de Demonstrativos Fiscais: facilita a consulta e exportação de demonstrativos consolidados com os valores de impostos definitivamente homologados.

- Finalização de Processos de Auditoria: onde ocorre a conclusão formal dos trabalhos de auditoria, com registro completo da obrigação principal apurada. Essa funciona-

lidade garante que todos os dados necessários para encerramento do caso estejam devidamente registrados no sistema.

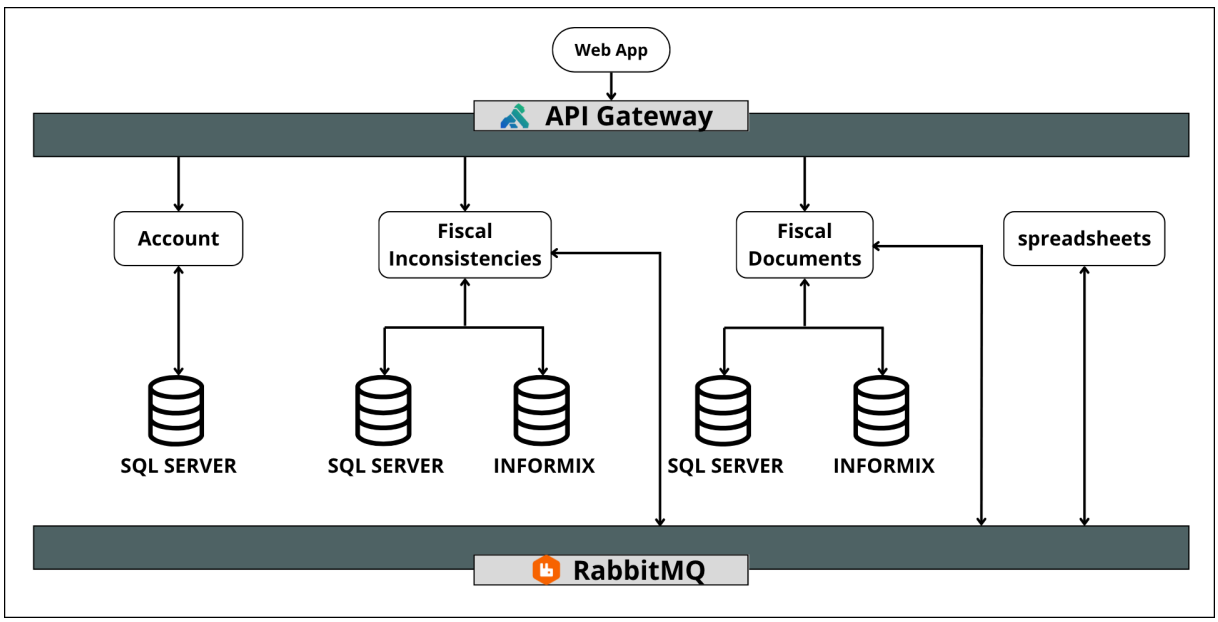
### 4.1.2 Arquitetura do SCAMF

O SCAMF foi originalmente desenvolvido seguindo uma arquitetura monolítica tradicional, com todas as funcionalidades implementadas em um conjunto de código. Nesse modelo, o back-end apresentava alto acoplamento entre seus componentes, o que inicialmente não era um grande problema, pois atendia aos requisitos do projeto, já que o sistema estava destinado a um grupo restrito de contribuintes.

Contudo, no final do ano de 2024, o requisito envolvendo a quantidade de contribuintes teve uma alteração, a SEFAZ-PB decidiu expandir o escopo do sistema para atender todos os contribuintes elegíveis do estado da Paraíba. Com isso, foi visto que a arquitetura monolítica utilizada possuía limitações significativas em termos de escalabilidade, para lidar com o aumento exponencial de usuários, flexibilidade, para implementar novas funcionalidades, e de capacidade de manutenção e atualizações independentes.

Diante desses desafios, está sendo realizada uma migração para a arquitetura baseada em microsserviços, representada na Figura 3. A nova arquitetura organiza as funcionalidades em serviços especializados e independentes.

Figura 3 – Visão geral da proposta de microsserviços para o SCAMF




*Fonte: Elaborado pelo autor, 2025*

Conforme ilustrado na Figura 3. A arquitetura do SCAMF foi divida nos seguintes serviços:

- API Gateway: ponto único de entrada para todas as requisições, responsável por

rotear chamadas aos microsserviços apropriados e implementar autenticação e autorização centralizadas.

- Account: gerencia a autenticação dos usuários, além de criar, armazenar e validar as chaves de acesso dos contribuintes.
- Fiscal Inconsistencies: gerencia todo o fluxo de auditoria fiscal, incluindo as ações nas malhas fiscais.
- Fiscal Documents: armazena e organiza os documentos fiscais.
- Spreadsheets: gera e exporta relatórios em formato Excel, garantindo a integridade dos dados exportados.

A persistência de dados é garantida por bancos relacionais SQL Server e Informix, ambos utilizando SQL como linguagem de consulta e operações. Essa abordagem permite o isolamento dos dados entre serviços, enquanto mantém consistência transacional onde necessário.

Atualmente, a transição para a arquitetura de microsserviços está sendo realizada de forma incremental, priorizando a continuidade operacional e a minimização dos riscos. Cada microsserviço é desenvolvido, testado e implantado individualmente. Quando um serviço é considerado estável e validado, ele é integrado ao ambiente de produção.

Durante o período de transição, o sistema opera em um modelo híbrido. Os microsserviços já implementados funcionam em paralelo com as partes ainda não migradas do monolito original. Esse modelo híbrido garante que todas as funcionalidades permaneçam disponíveis para os usuários, sem interrupções no serviço. A integração entre os novos serviços e o sistema legado é cuidadosamente gerenciada para manter a consistência dos dados e a qualidade do serviço.

Diante das mudanças na arquitetura, tornou-se necessário assegurar que as funcionalidades implementadas nos novos serviços apresentem comportamento equivalente ao da arquitetura monolítica. Assim, para manter o padrão de qualidade e confiabilidade do SCAMF, foi preciso desenvolver novos testes, manter os já existentes e implementar uma integração que tornasse os resultados visíveis para toda a equipe de desenvolvimento.

## 4.2 Desenvolvimento da integração e dos Testes Automatizados no SCAMF

### 4.2.1 Configuração do Cypress e Cucumber

O primeiro passo para a construção dos testes automatizados consistiu na criação e configuração do ambiente de testes. Para isso, foi necessária a instalação das ferramentas Cypress e Cucumber.

Antes da instalação das dependências, foi necessário garantir que o ambiente de desenvolvimento contasse com o Node.js e o npm devidamente instalados, pois são pré-requisitos para a execução do Cypress e para o gerenciamento das dependências.

Ademais, o processo de configuração ocorreu de acordo com o apresentado no Apêndice A. Com toda essa configuração de estrutura e dependências realizadas, foi possível dar início ao desenvolvimento dos cenários de teste e do código dos testes automatizados.

### 4.2.2 Cenários de Teste

Com o Cucumber devidamente configurado, é possível criar os cenários de testes utilizando o Gherkin. O Gherkin facilita a escrita do cenário, permitindo escrevê-los em linguagem natural, com o uso de palavras reservadas, como: Given, When e Then.

A Figura 4 mostra o cenário de teste de login do usuário do tipo fiscal no SCAMF, escrito usando o Gherkin e suas palavras reservadas, destacadas na cor azul.

Figura 4 – Cenário de teste de login do fiscal

loginFiscal.feature

cypress > e2e > features > fiscal > login > loginFiscal.feature

You, há 4 dias | 1 author (You)

```
1  @LoginFisc
2  Feature: Login functionality for fiscal users
3
4      Background:
5          Given the fiscal is on the login page
6
7      @TLF-1
8      Scenario: Fiscal logs in with valid credentials
9          When the fiscal enters username "<fiscal_username>" and password "<fiscal_password>"
10         Then the fiscal should be redirected to the OSs screen
```

*Fonte: Elaborado pelo autor, 2025*

Ainda na Figura 4 é possível observar que a tag “@TLF-1” foi atribuída ao cenário e a tag “LoginFisc”foi atribuída a “Feature” como um todo. Dessa forma é possível executar apenas o cenário em específico ou a “Feature” inteira, informando a respectiva tag no comando de execução.

O cenário descrito na Figura 4 foi elaborado a partir do caso de teste apresentado na Figura 5. Esse caso de teste foi construído com base nos requisitos relacionados à funcionalidade de login do sistema, com o objetivo de verificar se a implementação no serviço Account está integrada corretamente e se comporta conforme o esperado.

Figura 5 – Caso de teste de login do fiscal

| • **TLF01 - Logar com username correto e senha válida** |
|---|
| **Dado** que o fiscal esteja na tela de login |
| **Quando** ele informar username correto e senha válida |
| **Então** o fiscal é autenticado no sistema |

*Fonte: Elaborado pelo autor, 2025*

Seguindo a estrutura de pastas definida anteriormente, os cenários são armazenados em "./Cypress/e2e/features". Cada arquivo possui a extensão ".feature". O arquivo referente aos cenários de login do fiscal foi denominado como "loginFiscal.feature", por exemplo.

### 4.2.3 Execução do Cenário de Teste

Um cenário é composto por um ou mais passos, conforme demonstrado na Figura 4. Os passos são transformados em código executável através dos "step definitions", que consistem em funções JavaScript utilizando o Cypress para simular a interação com o sistema . A escrita utilizada nos passos do cenário deve ser a mesma utilizada para as funções, visto que a associação é feita através do padrão de texto utilizado nos cenários, onde o Cucumber identifica e executa a função correspondente declarada nos arquivos de step definitions.

Figura 6 – Step definitions para o cenário de teste de login do fiscal

cypress > e2e > step_definitions > fiscal > login > JS loginFiscal.js > ...

You, há 2 meses | 1 author (You)

```
1  import { Given, When, Then } from '@badeball/cypress-cucumber-preprocessor';
2  import LoginFiscalPage from '../../../../pages/fiscal/login/loginFiscal.page.js';
3
4  const loginFiscalPage = new LoginFiscalPage();
5
6  Given('the fiscal is on the login page', () => {
7      loginFiscalPage.fiscalLoginPage();
8  });
9
10 When('the fiscal enters username {string} and password {string}', (username, password) => {
11     const resolvedUsername = username === '<fiscal_username>' ? Cypress.env('fisc_username') : username;
12     const resolvedPassword = password === '<fiscal_password>' ? Cypress.env('fisc_password') : password;
13
14     loginFiscalPage.loginCommand(resolvedUsername, resolvedPassword);
15     loginFiscalPage.clickLoginButton();
16 });
17
18 Then('the fiscal should be redirected to the OSs screen', () => {
19     loginFiscalPage.validateRedirectToOSScreen();
20 });
```

*Fonte: Elaborado pelo autor, 2025*

Como demonstrado na Figura 6, cada passo do cenário é implementado por funções que replicam as ações que um usuário real realizaria no sistema. Essas funções estão organizadas seguindo o padrão Page Object Model (POM), que promove a separação de responsabilidades e proporciona a reutilização de código nos testes. Com isso, o código é organizado da seguinte maneira:

- Step Definitions (./Cypress/e2e/step_definitions): Contêm as funções que ligam os passos dos cenários e as ações no sistema.
- Pages (./Cypress/pages): Implementam as funções, chamadas no step definitions, que representam as ações do usuário, encapsulando a lógica de interação com cada página do sistema.
- Page Elements (./Cypress/page_elements): Armazenam os seletores dos elementos da interface, como os botões, por exemplo.

As Figuras 7 e 8, mostram, respectivamente, como foram definidas as funções e os seletores que dão suporte ao step definitions da Figura 6.

Figura 7 – Implementação de funções que realizam ações voltadas ao login

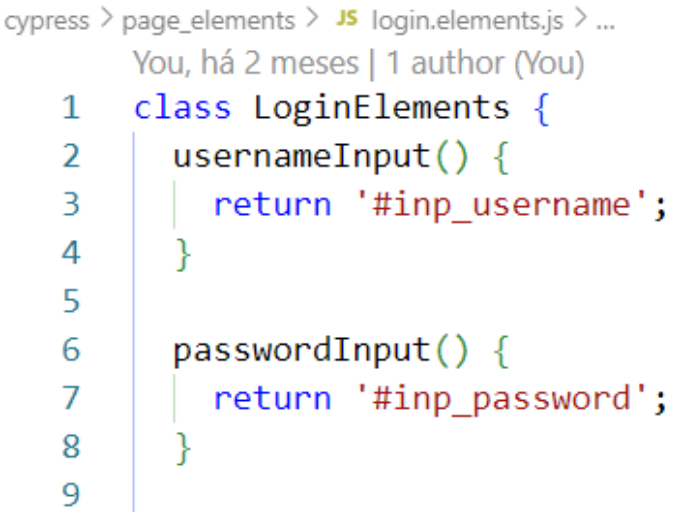
cypress > pages > fiscal > login > JS loginFiscal.page.js > ...

You, há 2 meses | 1 author (You)

```
1  import LoginElements from '../../../page_elements/login.elements.js';
2
3  const baseUrl = Cypress.config('baseUrl');
4  const loginElements = new LoginElements();
5
   You, há 2 meses | 1 author (You)
6  class LoginFiscalPage {
7    fiscalLoginPage() {
8      cy.visit('login/');
9    }
10
11   loginCommand(username, password) {
12     cy.login(username, password);
13   }
14
15   enterUsername(username) {
16     cy.get(loginElements.usernameInput()).type(username);
17   }
```

*Fonte: Elaborado pelo autor, 2025*

Figura 8 – Seletores da interface do SCAMF




*Fonte: Elaborado pelo autor, 2025*

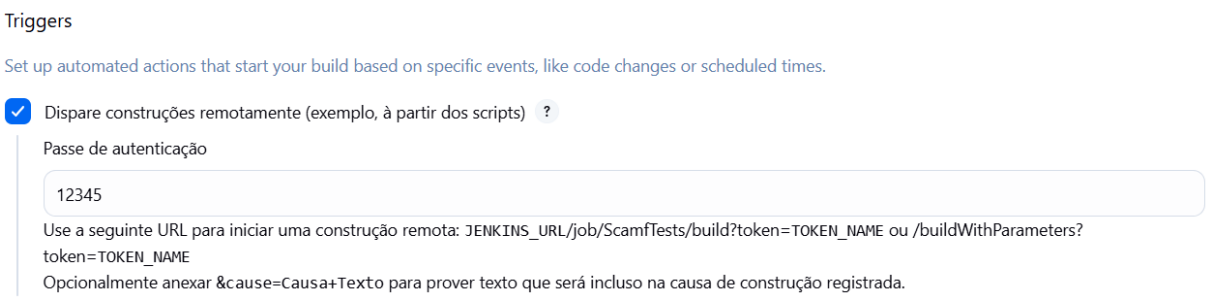
### 4.2.4 Configuração da Integração Contínua com Jenkins

Esta etapa consiste na configuração de uma tarefa, ou *job*, no Jenkins. O pré-requisito dessa etapa é que os testes automatizados desenvolvidos para o sistema estejam versionados em um repositório do GitHub.

O intuito é que o *job* seja configurado de maneira que possibilite o seu acionamento remoto. Quando acionado, ele deve clonar o repositório em que os testes estão versionados, executar os testes referentes à tag enviada como parâmetro e enviar os resultados para o Jira. Para isso, o *job* foi configurado como um projeto de estilo livre com construção parametrizada. O parâmetro a ser recebido foi definido como "TEST_TAG".

Em seguida, conforme mostrado na Figura 9, foi habilitada a opção de disparo de construção remota, de acordo com o passe de autenticação definido, permitindo que o *job* seja acionado via webhook.

Figura 9 – Configuração de construção remota

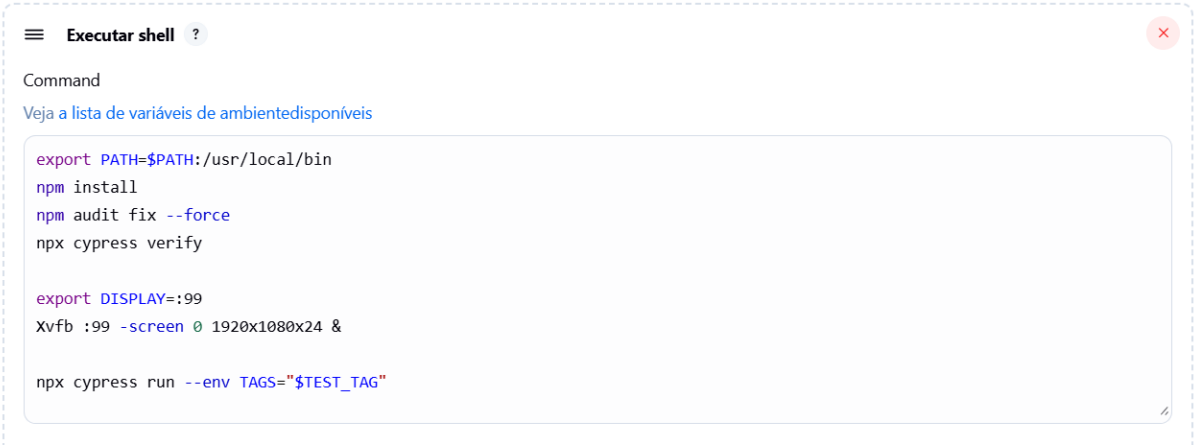



*Fonte: Elaborado pelo autor, 2025*

Na seção de "Passos de construção", conforme apresentado na Figura 10, foi adicionado

um comando responsável por realizar a instalação das dependências e executar os testes com base na tag recebida como parâmetro.

Figura 10 – Comando de execução dos testes no Jenkins




*Fonte: Elaborado pelo autor, 2025*

Em "Ações de pós-construção", foi instalado o plugin do AIO Tests para o Jira no Jenkins e devidamente configurado, informando o token gerado no Jira e as demais informações necessárias. Com a próxima etapa, que corresponde à configuração do Jira, já será possível executar o *job* e reportar os resultados ao Jira.

No caso do SCAMF, o Jenkins foi configurado em uma máquina virtual, onde, além de realizado todo o processo de configuração do *job*, ocorreu a liberação da porta de comunicação, utilizando um domínio pertencente à SEFAZ-PB, no qual o jenkins se tornou visível para web. Dessa forma, o acionamento remoto do *job* funciona perfeitamente a todo momento.

### 4.2.5 Configuração do Jira

No SCAMF, a plataforma de gerenciamento de projetos utilizada é o Jira. Tendo em vista isso, ele foi configurado para acionar e exibir os resultados dos testes automatizados de forma acessível a todos os membros da equipe.

A implementação teve início com a criação de um campo personalizado denominado "Test Tag" no Jira, conforme ilustrado na Figura 11. Esse campo permite que qualquer integrante da equipe informe a tag referente aos cenários dos testes que ele deseja executar.

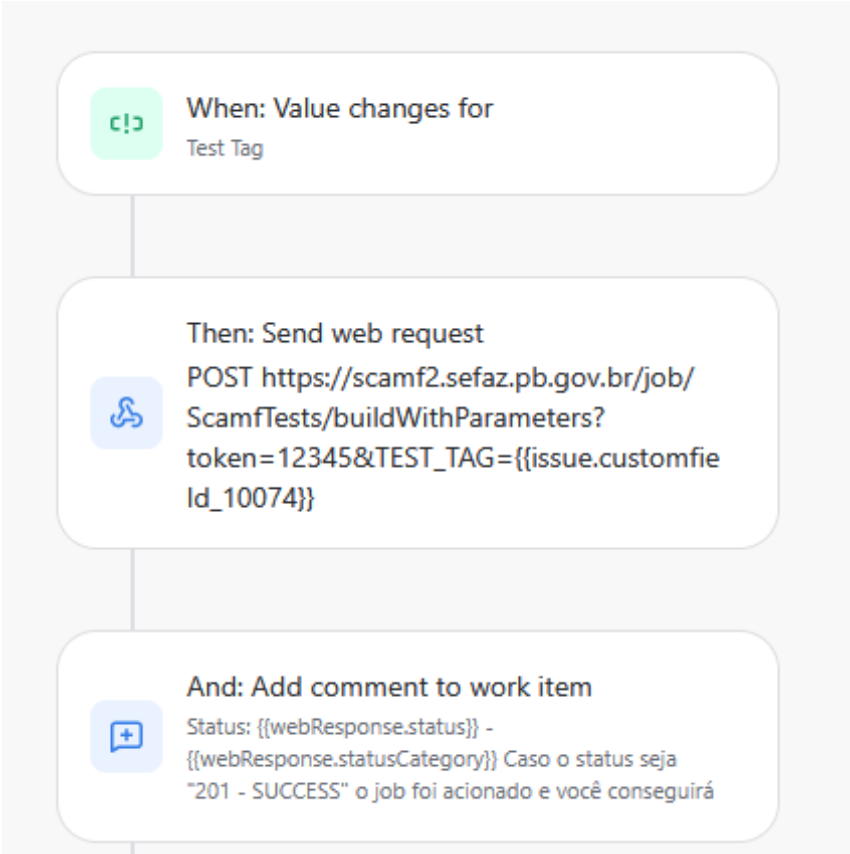
Figura 11 – Campo Test Tag associado a uma história do Jira


*Fonte: Elaborado pelo autor, 2025*

Em seguida, foi instalado e configurado o plugin AIO Tests no Jira. Com isso, foi possível gerar o token que permite a comunicação entre o Jira e o Jenkins, possibilitando o reporte dos resultados dos testes.

Para que o *job* consiga ser acionado, foi necessária a criação de uma automação no Jira, apresentada na Figura 12. A regra de automação parte do princípio que quando um valor é inserido ou modificado no campo "Test Tag", um webhook é imediatamente acionado, enviando uma requisição HTTPS para o Jenkins com a tag especificada. Essa requisição ocasiona a execução do *job*, realizando todo o processo de execução dos testes, de acordo com a tag informada, e reportando o resultado dos testes para o Jira.

Figura 12 – Regra de automação do Jira




*Fonte: Elaborado pelo autor, 2025*

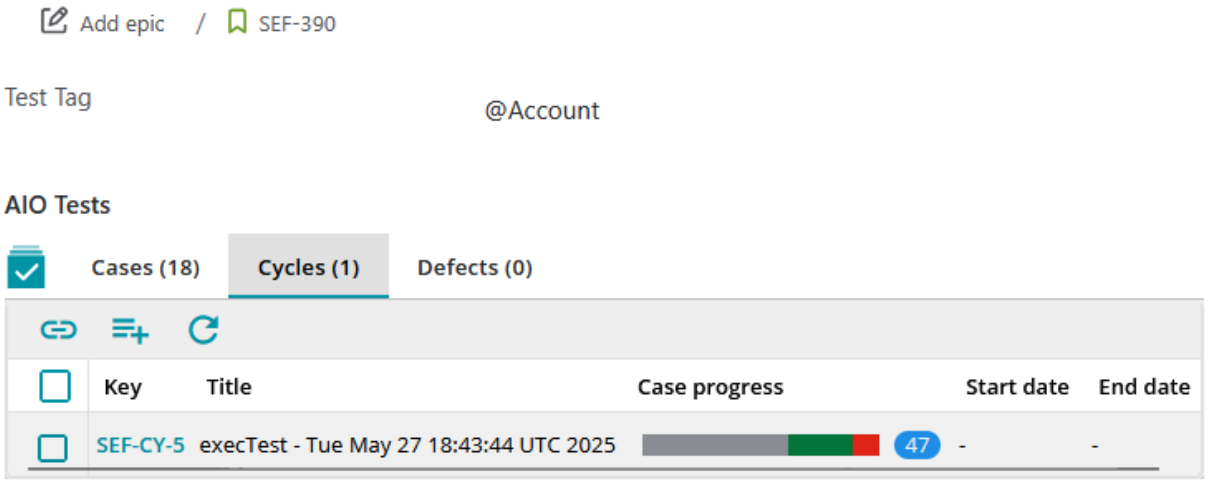
Com a finalização dessa etapa, o fluxo de integração dos testes automatizados no SCAMF foi devidamente configurado e disponibilizado para a equipe utilizá-lo.

### 4.2.6 Resultados da Integração

A implementação da integração entre Jira, Jenkins e os testes automatizados auxiliou significativamente no processo de desenvolvimento do SCAMF, com a identificação precoce de erros impedindo que eles se manifestassem no ambiente de produção, agregando valor ao processo de garantia da qualidade do sistema. Antes dessa integração, os testes eram realizados manualmente, com foco principal na verificação e validação pontual de novas funcionalidades. Essa prática requeria um tempo considerável, que variava de acordo com a complexidade da funcionalidade, podendo ocupar horas de trabalho da equipe para ser executada e analisada, dificultando a identificação do comportamento do sistema diante das novas funcionalidades.

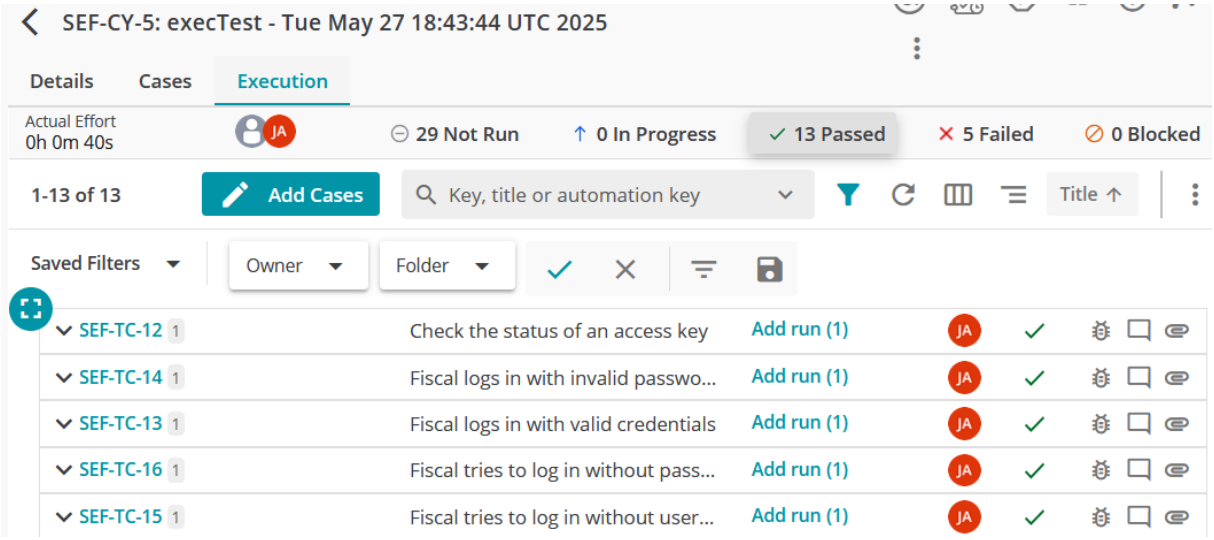
A capacidade de executar testes de forma seletiva e automatizada a partir do Jira, eliminou a necessidade de clonar e executar os testes localmente, possibilitando que qualquer integrante da equipe realize a execução dos testes desejados. Conforme demonstram as Figuras 13, 14 e 15, o integrante da equipe informa a tag desejada no campo "Test Tag" do Jira para que todo o ciclo de teste seja acionado automaticamente. A Figura 13 mostra a inserção da tag "@Account" no Jira e o ciclo "SEF-CY-5", e as Figuras 14 e 15 apresentam, respectivamente, alguns dos testes que passaram e falharam, e que foram devidamente reportados para o AIO Tests no Jira.

Figura 13 – Tag @Account no campo Test Tag e o ciclo SEF-CY-5



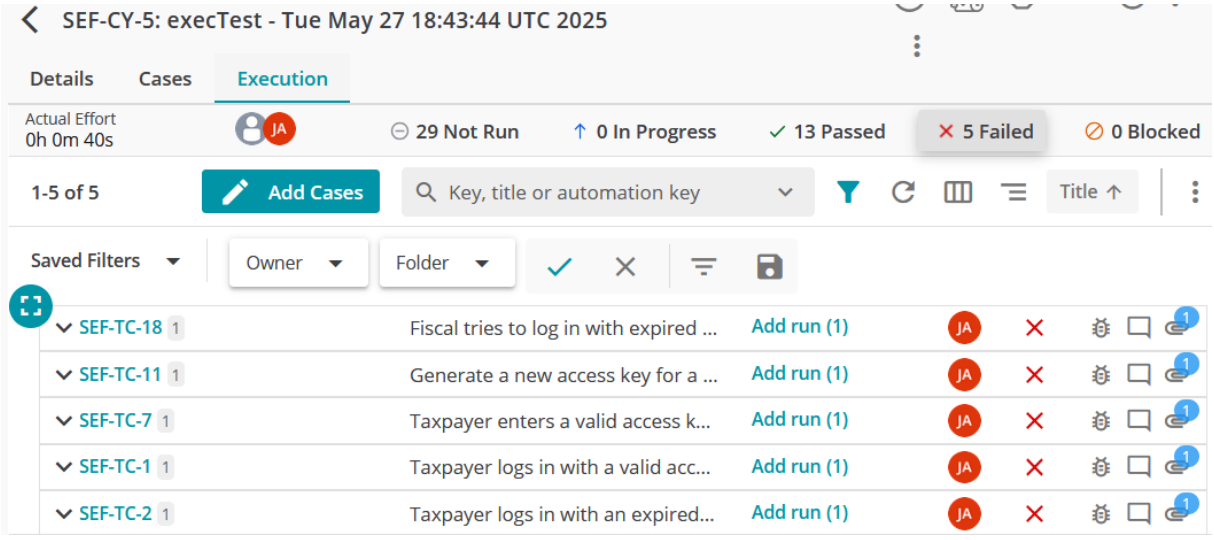

*Fonte: Elaborado pelo autor, 2025*

Figura 14 – Testes que passaram na execução de @Account




*Fonte: Elaborado pelo autor, 2025*

Figura 15 – Testes que não passaram na execução de @Account




*Fonte: Elaborado pelo autor, 2025*

Antes do início do processo de migração, os testes automatizados começaram a ser realizados. Ainda que de forma inicial, foram desenvolvidos, no contexto das funcionalidades que viriam a compor o serviço de Account, 8 testes automatizados. Com o sistema ainda na arquitetura monolítica, foram desenvolvidos, ao todo, 25 testes automatizados.

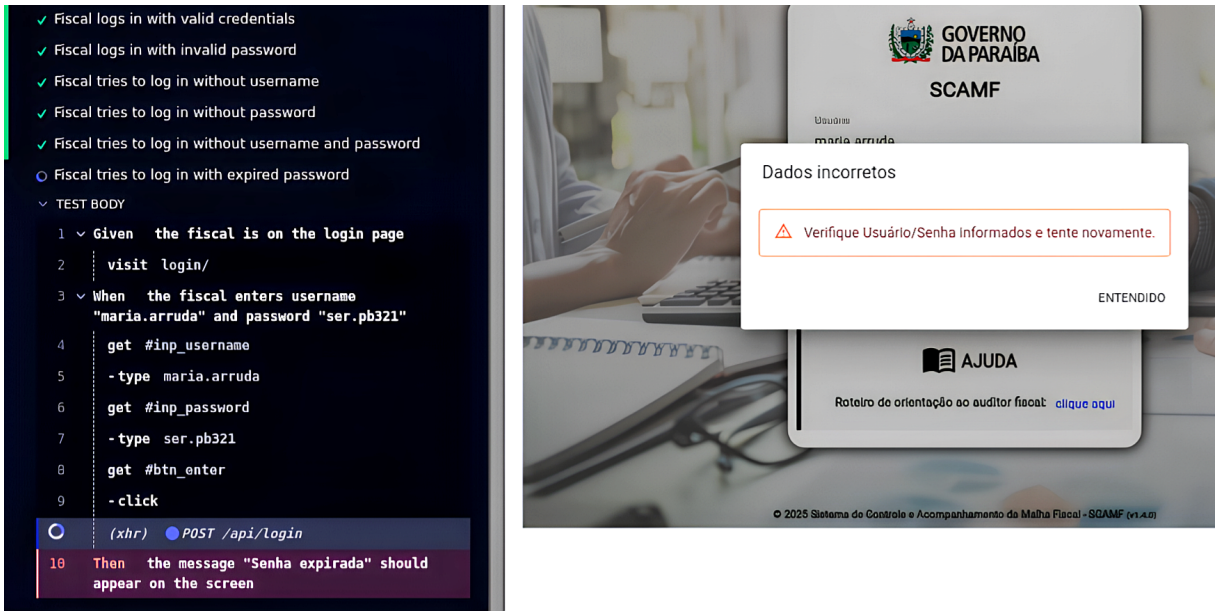
Atualmente, durante o processo de migração arquitetural, o SCAMF já conta com 47 testes automatizados, como indica a Figura 13, sendo que 18 desses testes são referentes ao serviço Account. Cada um dos cenários de teste do account pode ser observado no Apêndice B.

Com a migração para a arquitetura de microsserviços, cada novo serviço integrado ao sistema passou a ter seu conjunto de testes automatizados, que são executados juntamente com os testes já existentes para o sistema. A tag "@Account", utilizada na Figura 13, refere-se justamente aos testes elaborados para o serviço Account. Essa abordagem permitiu identificar de maneira mais rápida incompatibilidades e comportamentos inesperados no sistema. Como o caso da Figura 15, que apresenta os testes que falharam, dentre eles está o "SEF-TC-18" que se refere ao cenário de tentativa de login com senha expirada. Na arquitetura monolítica original do SCAMF, o "SEF-TC-18" foi executado com sucesso, mas quando ocorreu a integração com o serviço Account, ele apresentou falha.

Quando ocorre a falha de um teste, junto ao reporte informando que ele falhou, é disponibilizada uma captura de tela que mostra em qual passo do cenário ocorreu a falha. A Figura 16 consiste na captura de tela que documenta a execução do cenário de teste "SEF-TC-18", evidenciando que o esperado seria a exibição da mensagem "Senha expirada". No entanto, a mensagem exibida foi "Verifique Usuário/Senha informados e tente novamente".

Ao analisar a informação da Figura 16, constatou-se que havia um problema na função responsável por realizar a verificação das credenciais de acesso no serviço Account. O problema foi corrigido mediante a atualização da condicional que distingue as credenciais inválidas das expiradas.

Figura 16 – Cenário de teste referente ao SEF-TC-18



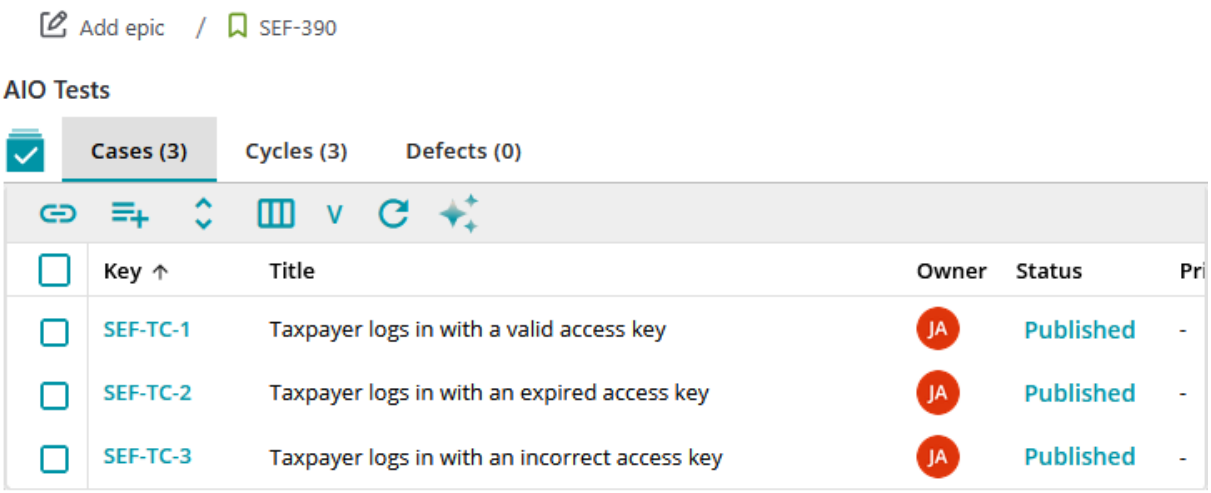

*Fonte: Elaborado pelo autor, 2025*

A identificação precoce de comportamentos indesejados no software, como o apresentado na Figura 16, contribui para garantir a qualidade do produto e a confiabilidade de

suas funcionalidades. Ao corrigir essas inconsistências ainda na fase de desenvolvimento, minimiza-se a ocorrência de falhas operacionais que poderiam impactar diretamente os usuários finais, contribuindo para a satisfação do cliente e para a integridade do sistema.

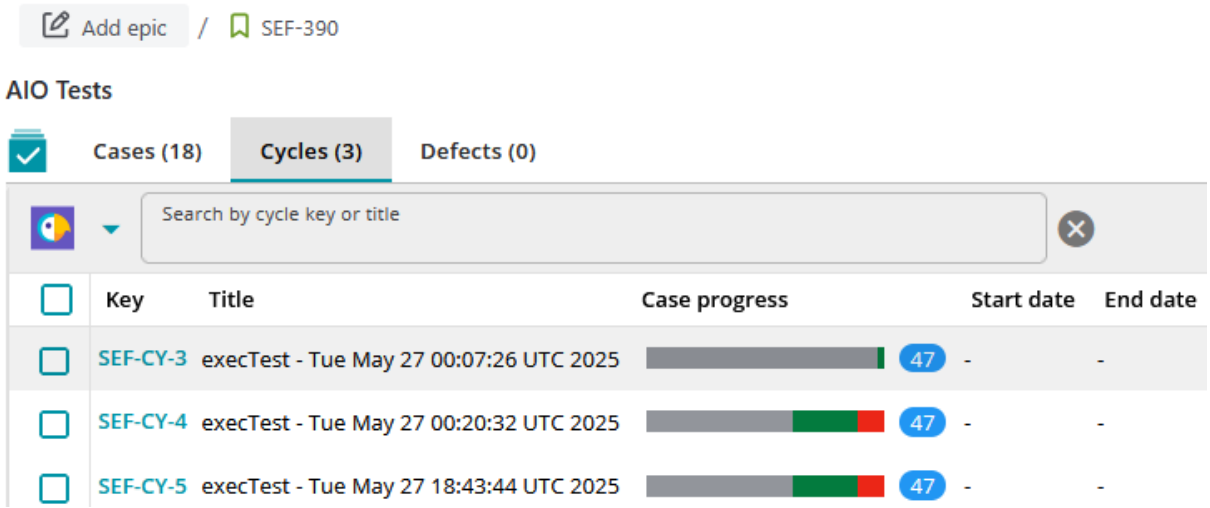
Adicionalmente, a integração proporciona a rastreabilidade completa dos testes, permitindo associar casos de teste individuais e ciclos de teste a diferentes componentes do Jira. Como demonstrado nas Figuras 17 e 18, é possível vincular os testes automatizados e seus ciclos de execução a elementos específicos do Jira, facilitando para toda a equipe a compreensão da relação entre os testes e os requisitos descritos.

Figura 17 – Testes associados à história SEF-390 do Jira




*Fonte: Elaborado pelo autor, 2025*

Figura 18 – Ciclos de testes associados à história SEF-390 do Jira




*Fonte: Elaborado pelo autor, 2025*

# 5 DISCUSSÃO

A análise dos resultados obtidos comprova que a integração e automação dos testes proporcionou ganhos significativos em comparação com a abordagem manual anteriormente utilizada. A capacidade de executar testes de forma seletiva através das tags, como no caso da tag "@Account", permitiu à equipe identificar rapidamente incompatibilidades, provenientes do processo de migração de arquitetura, e corrigi-las de imediato, evitando que elas acontecessem no ambiente de produção.

Um dos aspectos que teve uma melhoria significativa com a integração foi a rastreabilidade dos testes. Com a possibilidade de vincular os casos de teste e os requisitos funcionais no Jira, criou-se um ambiente de documentação e acompanhamento. Cada teste passou a estar associado às suas especificações originais, com todos os registros de execução armazenados. Essa rastreabilidade permitiu uma maior transparência no processo de desenvolvimento, permitindo que toda a equipe tivesse visibilidade de cada funcionalidade e dos testes relacionados. Além disso, a possibilidade de executar os testes diretamente pela interface do Jira, sem a necessidade de configurações e execuções locais, democratizou o acesso aos testes e acelerou a obtenção de resultados sobre a qualidade do software.

Em comparação com a abordagem manual anteriormente utilizada, a abordagem automatizada tornou o processo de detecção de erros mais eficiente, com problemas sendo identificados de maneira precoce. Um exemplo disso é apresentado na Figura 16, em que, ao informar apenas a respectiva tag associada, o teste foi executado e o erro identificado em apenas 6 segundos. Essa identificação precoce resultou em uma redução expressiva no tempo necessário para correções, além de diminuir significativamente a ocorrência de comportamentos indesejados em funcionalidades já implementadas no sistema. A qualidade geral do produto final melhorou, com menos problemas atingindo os usuários finais e maior confiabilidade no sistema como um todo.

No processo de implantação da abordagem, houve um esforço considerável para configurar o ambiente de integração contínua, incluindo o Jenkins e a integração com o Jira. O Jenkins precisou ser instalado em uma máquina virtual, onde foi necessário descobrir como possibilitar o acionamento remoto do *job*. Superado esse desafio, o próximo passo foi identificar e configurar um plugin capaz de realizar a comunicação dos resultados entre o Jenkins e o Jira, o plugin escolhido e configurado em ambas as ferramentas foi o AIO Tests. Esses desafios não comprometeram os benefícios alcançados, sendo superados com a experiência adquirida ao longo do processo. O ideal é que a integração seja construída por integrantes da equipe que já possuam um conhecimento prévio dessas ferramentas.

Em síntese, a implementação da integração e automação de testes no desenvolvimento do sistema demonstrou ser uma estratégia viável e benéfica. A combinação das ferramen-

tas Cucumber, Cypress, Jenkins e Jira otimizou a detecção precoce de falhas e aprimorou a rastreabilidade, a eficiência e a colaboração entre a equipe.

# 6 CONSIDERAÇÕES FINAIS

## 6.1 Conclusão

Esse estudo representou uma contribuição significativa para o Sistema de Controle e Acompanhamento da Malha Fiscal (SCAMF) da SEFAZ-PB, ao implementar a abordagem de integração contínua e automação de testes funcionais, auxiliando no ciclo de desenvolvimento do software, especialmente durante o processo de migração da arquitetura monolítica para microsserviços.

Os resultados obtidos demonstraram a eficácia da abordagem proposta. A implementação dos testes automatizados utilizando Cypress e Cucumber, seguindo o padrão Page Object Model, mostrou-se particularmente vantajosa para garantir a manutenibilidade e a escalabilidade dos testes. O Jenkins possibilitou a criação de um *job* de integração contínua altamente eficiente, incluindo a comunicação com o Jira, por meio do AIO Tests, que proporcionou transparência no processo dos testes realizados no sistema.

Uma das características mais relevantes observadas foi a capacidade do sistema de detectar comportamentos indesejados que poderiam passar despercebidos em testes manuais. O caso da validação de senhas expiradas no serviço Account exemplificou como a automação pode prevenir problemas operacionais antes que eles afetem os usuários finais. A redução no tempo de execução dos testes, sem a necessidade de uma configuração local, proporcionou um ganho de produtividade considerável para a equipe envolvida no sistema. Além desses aspectos técnicos, a centralização dos resultados no Jira proporcionou um ambiente mais colaborativo, onde desenvolvedores, testadores e gestores podem executar os testes e interpretar seus resultados.

Apesar dos resultados obtidos, esse estudo apresenta algumas limitações. A abordagem focou nos testes funcionais, deixando de explorar outros tipos de testes, como desempenho e segurança, que são igualmente relevantes para o sistema. Além disso, a solução depende de um conjunto específico de ferramentas, o que pode exigir adaptações futuras caso aconteçam mudanças impactantes nessas tecnologias.

## 6.2 Estudos Futuros

A abordagem de integração e automação dos testes apresentada permite diversas expansões e melhorias. Essa abordagem poderá ser aplicada futuramente em testes de desempenho, principalmente em sistemas que apresentem um crescimento considerável de usuários. Da mesma forma, a metodologia mostra-se promissora para aplicação em testes de segurança, proporcionando um maior fortalecimento dos sistemas.

A metodologia desenvolvida nesse trabalho não se restringe ao SCAMF. Os princípios e técnicas utilizados podem ser adaptados a sistemas semelhantes, contribuindo não só para a qualidade do SCAMF, mas também para a de outros sistemas.

# REFERÊNCIAS

ALFERIDAH, S. K.; AHMED, S. Automated software testing tools. In: UNIVERSITY OF TABUK, KINGDOM OF SAUDI ARABIA. *Proceedings of the 2020 International Conference on Computing and Information Technology (ICCIT 1441)*. Tabuk, Saudi Arabia, 2020. p. 183–186.

ATLASSIAN. *O que é o Jira?* 2025. Acesso em: 29 maio 2025. Disponível em: ⟨https://www.atlassian.com/br/software/jira/guides/getting-started/introduction#what-is-jira-software⟩.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. 4. ed. Boston: Pearson Education, 2022.

BERNARDO, P. C.; KON, F. A importância dos testes automatizados: Controle ágil, rápido e confiável de qualidade. *Engenharia de Software Magazine*, v. 1, n. 3, p. 54–57, 2008.

CRISPIN, L.; GREGORY, J. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Boston: Addison-Wesley, 2009.

CUCUMBER.IO. *Behaviour-Driven Development*. 2025. Acesso em: 21 maio 2025. Disponível em: ⟨https://cucumber.io/docs/bdd/⟩.

CUCUMBER.IO. *Cucumber Documentation*. 2025. Acesso em: 21 maio 2025. Disponível em: ⟨https://cucumber.io/docs/⟩.

CYPRESS.IO. *Why Cypress?* 2025. Acesso em: 20 maio 2025. Disponível em: ⟨https://docs.cypress.io/app/get-started/why-cypress⟩.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao Teste de Software*. 4. ed. Rio de Janeiro: Elsevier Editora Ltda., 2007.

DUARTE, K. C.; FALBO, R. de A. Uma ontologia de qualidade de software. In: *Anais do VII Workshop de Qualidade de Software (WQS'2000)*. João Pessoa, Brasil: [s.n.], 2000.

FOWLER, M.; HIGHSMITH, J. The agile manifesto. *Software development*, Miller Freeman, v. 9, n. 8, p. 28–35, 2001. Acesso em: 15 Jun. 2025. Disponível em: ⟨https://www.hristov.com/andrey/fht-stuttgart/The_Agile_Manifesto_SDMagazine.pdf⟩.

GARLAN, D. Software architecture. 2008. Acesso em: 30 maio 2025. Disponível em: ⟨https://kilthub.cmu.edu/articles/journal_contribution/Software_Architecture/6609593/files/12101711.pdf⟩.

GUIMARÃES, R. S. Trabalho de Conclusão de Curso, *Estudo de automação de testes funcionais e integração contínua em um sistema de gestão*. Caxias do Sul: [s.n.], 2016. Acesso em: 16 Jun. 2025. Disponível em: ⟨https://repositorio.ifsc.edu.br/bitstream/handle/123456789/1756/TCC-FINAL.pdf?sequence=1⟩.

JAISWAL, D. M. Software architecture and software design. *International Research Journal of Engineering and Technology (IRJET)*, v. 6, n. 11, p. 2452–2454, 2019. ISSN 2395-0056. Acesso em: 30 maio 2025. Disponível em: ⟨https://www.irjet.net/archives/V6/i11/IRJET-V6I11303.pdf⟩.

JENKINS.IO. *AIO Tests*. 2025. Acesso em: 21 maio 2025. Disponível em: ⟨https://plugins.jenkins.io/aio-tests/⟩.

JENKINS.IO. *Jenkins User Documentation*. 2025. Acesso em: 21 maio 2025. Disponível em: ⟨https://www.jenkins.io/doc/⟩.

LEWIS, J.; FOWLER, M. *Microservices: a Definition of this New Architectural Term*. 2014. Acesso em: 30 maio 2025. Disponível em: ⟨https://martinfowler.com/articles/microservices.html⟩.

MANYIKA, J. et al. *Digital Globalization: The New Era of Global Flows*. San Francisco: McKinsey Global Institute, 2016. Acesso em: 15 Jun. 2025. Disponível em: ⟨https://www.mckinsey.com/mgi/overview/2016-in-review/digital-globalization⟩.

MELO, L. R. d. Integração contínua para automação de testes utilizando cypress, docker e jenkins. Universidade de Passo Fundo, Passo Fundo, Brasil, 2024. Acesso em: 15 Jun. 2025. Disponível em: ⟨http://repositorio.upf.br/bitstream/riupf/2840/1/PF2024LucasRosadeMelo.pdf⟩.

MOBARAYA, F.; ALI, S. Technical analysis of selenium and cypress as functional automation framework for modern web application testing. In: *Proceedings of the 9th International Conference on Computer Science*. Auckland, New Zealand: AGI Institute, 2019.

MYERS, G. J.; BADGETT, T.; SANDLER, C. *The Art of Software Testing*. 3. ed. Hoboken, NJ: John Wiley & Sons, Inc., 2012.

NEWMAN, S. *Building Microservices*. 1. ed. Sebastopol, CA: O'Reilly Media, 2015.

POWELL, P.; SMALLEY, I. *Monolithic Architecture*. 2024. Acesso em: 30 maio 2025. Disponível em: ⟨https://www.ibm.com/think/topics/monolithic-architecture⟩.

PRESSMAN, R. S.; MAXIM, B. R. *Engenharia de Software: Uma Abordagem Profissional*. 9. ed. Porto Alegre: AMGH Editora Ltda., 2021.

RICHARDS, M. *Software Architecture Patterns: Understanding Common Architecture Patterns and When to Use Them*. 1. ed. Sebastopol, CA: O'Reilly Media, 2015.

SCHWAB, K. *The Fourth Industrial Revolution*. Geneva, Switzerland: World Economic Forum, 2016. ISBN 978-1-944835-01-9.

SOMMERVILLE, I. *Engenharia de Software*. 10. ed. São Paulo: Pearson Universidades, 2019.

WHOLIN, C. Case study research in software engineering—it is a case, and it is a study, but is it a case study? *Information and Software Technology*, Elsevier, v. 133, p. 106514, 2021. Acesso em: 04 Jun. 2025. Disponível em: ⟨https://www.sciencedirect.com/science/article/pii/S0950584921000033⟩.

## APÊNDICE A - Estrutura e dependências do Cypress e Cucumber

Estrutura e Dependências Necessárias para os Testes Automatizados

Inicialmente, para que seja possível desenvolver os testes automatizados, foram instaladas as seguintes dependências de desenvolvimento:

- @badeball/cypress-cucumber-preprocessor: Integra o Cucumber ao Cypress, permitindo a escrita de cenários de teste utilizando a sintaxe Gherkin.
- @bahmutov/cypress-esbuild-preprocessor: Responsável por otimizar e agilizar a transpilação de arquivos no ambiente de testes.
- cypress@12.0.2: Framework principal para automação de testes de interface.
- esbuild@0.16.4: Empregado como empacotador e transpilador de arquivos, essencial para a integração com o esbuild preprocessor.

A instalação de cada uma das dependências foi realizada por meio do comando "npm install" seguido do nome da dependência.

A estrutura padrão do Cypress gera automaticamente algumas pastas. Para implementar o padrão Page Object Model (POM) e garantir a integração adequada com o fluxo de integração contínua, deve-se adicionar as seguintes pastas e arquivos:

- ./Cypress/e2e/features: Armazena os arquivos ".feature" escritos em Gherkin.
- ./Cypress/e2e/step_definitions: Contém as implementações dos passos descritos nos arquivos ".feature".
- ./Cypress/page_elements: Pasta que armazena seletores centralizados para elementos da interface.
- ./Cypress/pages: Pasta com arquivos que definem métodos e ações encapsuladas que representam o comportamento de páginas.
- ./.cypress-cucumber-preprocessorrc.json: Arquivo de configuração do Cypress Cucumber Preprocessor.
- ./cucumber-json-formatter: Executável que gera os resultados dos testes no formato JSON.

O arquivo “cypress.config.js” é o responsável pela configuração que garante o funcionamento do Cypress. Para que o Cypress consiga funcionar em conjunto com o Cucumber, é necessário realizar a configuração apresentada na Figura 1.

Figura 1 – Arquivo de configuração cypress.config.js

cypress.config.js > ...

You, há 2 meses | 1 author (You)

```
const { defineConfig } = require("cypress");
const createBundler = require("@bahmutov/cypress-esbuild-preprocessor");
const preprocessor = require("@badeball/cypress-cucumber-preprocessor");
const createEsbuildPlugin = require("@badeball/cypress-cucumber-preprocessor/esbuild");

async function setupNodeEvents(on, config) {
  await preprocessor.addCucumberPreprocessorPlugin(on, config);
  on(
    "file:preprocessor",
    createBundler({
      plugins: [createEsbuildPlugin.default(config)],
    })
  );
  return config;
}

module.exports = defineConfig({
  e2e: {
    setupNodeEvents,
    specPattern: [
      "cypress/e2e/features/*.feature",
      "cypress/e2e/features/*/*.feature",
      "cypress/e2e/features/*/*/*.feature",
    ],
    stepDefinitions: [
      "cypress/e2e/step_definitions/**/*.{js,ts}",
      "cypress/e2e/step_definitions/*.{js,ts}",
      "cypress/e2e/step_definitions/*/*.{js,ts}",
      "cypress/e2e/step_definitions/*/*/*/*.{js,ts}",
    ],
  },
});
```

*Fonte: Elaborado pelo autor, 2025*

O arquivo “.cypress-cucumber-preprocessorrc.json” deve ser configurado conforme a Figura 2. Nesse arquivo é habilitado o “cucumber-json-formatter”, possibilitando a geração e o armazenamento dos resultados dos testes em formato JSON.

É importante destacar que os dois arquivos possuem configurações voltadas para o “step_definitions”, elas precisam estar alinhadas para evitar erros na identificação da pasta e dos arquivos, garantindo que os testes sejam executados com sucesso.

Figura 2 – Arquivo de configuração .cypress-cucumber-preprocessorrc.json


*Fonte: Elaborado pelo autor, 2025*

## APÊNDICE B - Casos de Teste

Casos de teste desenvolvidos para o serviço Account

Cada um dos casos de teste do serviço Account está relacionado a pelo menos um requisito do sistema. Nesse sentido, os principais requisitos associados ao Account são:

- Requisito 01 - Autenticação de Contribuinte por Chave de Acesso: define que um usuário do tipo “Contribuinte” consegue acessar o sistema utilizando uma chave de acesso. Ele cobre tanto o sucesso do login quanto as principais falhas de validação da chave. Os casos de teste para esse requisito estão representados nas Figuras 1, 2, 3, 6 e 7.

- Requisito 02 - Validação de Formato da Chave de Acesso: garante que o campo de entrada da chave de acesso possua validações de formato, rejeitando entradas que não seguem o padrão definido para uma chave. Os casos de teste para esse requisito estão representados nas Figuras 8, 9 e 10.

- Requisito 03 - Validação da Interface de Login do Contribuinte: descreve o comportamento esperado da interface de usuário na tela de login do contribuinte, focando em validações que ocorrem antes mesmo do envio dos dados ao servidor. Os casos de teste para esse requisito estão representados nas Figuras 4 e 5.

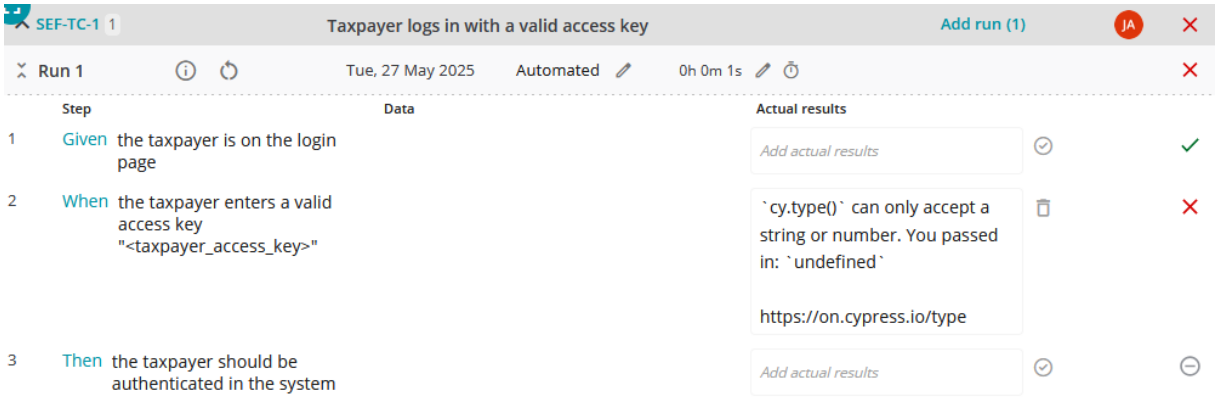
- Requisito 04 - Autenticação do Fiscal: define que os usuários do tipo “Fiscal” devem se autenticar no sistema usando credenciais de usuário e senha, e como o sistema deve tratar casos específicos como senhas expiradas. Os casos de teste para esse requisito estão representados nas Figuras 13, 14 e 18.

- Requisito 05 - Geração de Nova Chave de Acesso (Perfil Fiscal): descreve a capacidade que um usuário “Fiscal” tem de gerar novas chaves de acesso para os contribuintes. O caso de teste para esse requisito está representado na Figura 11.

- Requisito 06 - Validação da Interface de Login do Fiscal: descreve o comportamento esperado da interface de usuário na tela de login do Fiscal, focando em validações de preenchimento dos campos. Os casos de teste para esse requisito estão representados nas Figuras 15, 16 e 17.

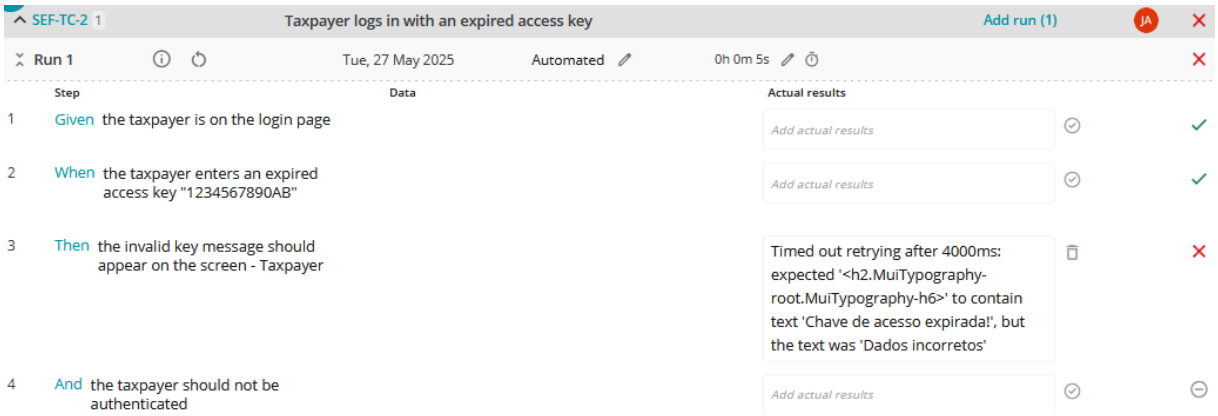
- Requisito 07 - Consulta de Status da Chave de Acesso: define a funcionalidade que permite a um fiscal, já autenticado no sistema, verificar os detalhes de uma chave de acesso existente. O caso de teste para esse requisito está representado na Figura 12.

Figura 1 – Caso de teste “Taxpayer logs in with a valid access key”



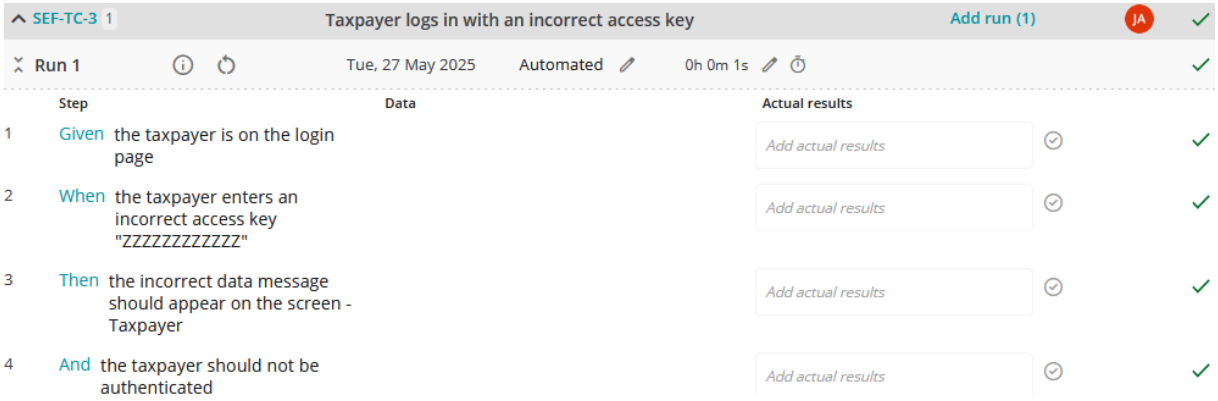

*Fonte: Elaborado pelo autor, 2025*

Figura 2 – Caso de teste “Taxpayer logs in with an expired access key”



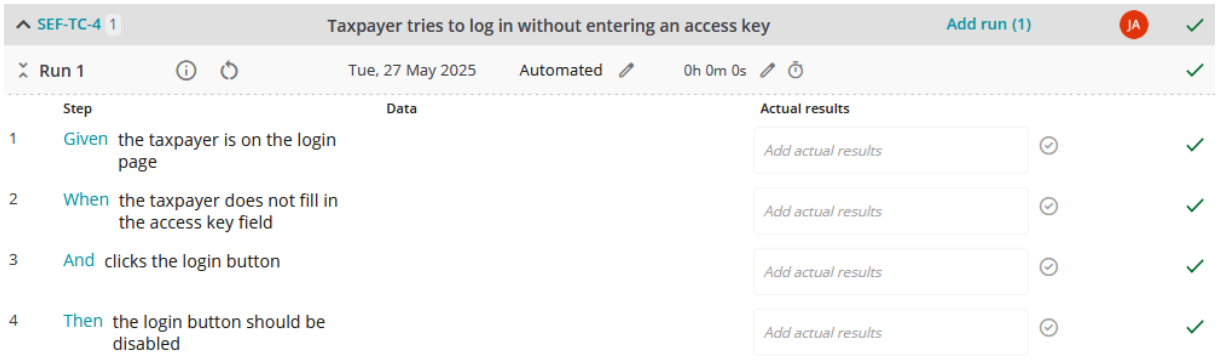

*Fonte: Elaborado pelo autor, 2025*

Figura 3 – Caso de teste“Taxpayer logs in with an incorrect access key”



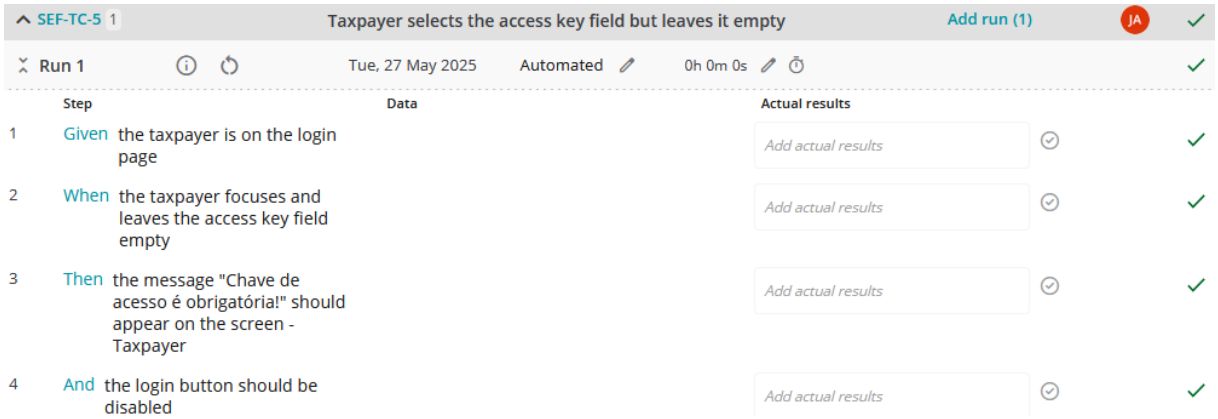

*Fonte: Elaborado pelo autor, 2025*

Figura 4 – Caso de teste “Taxpayer tries to log in without entering an access key”



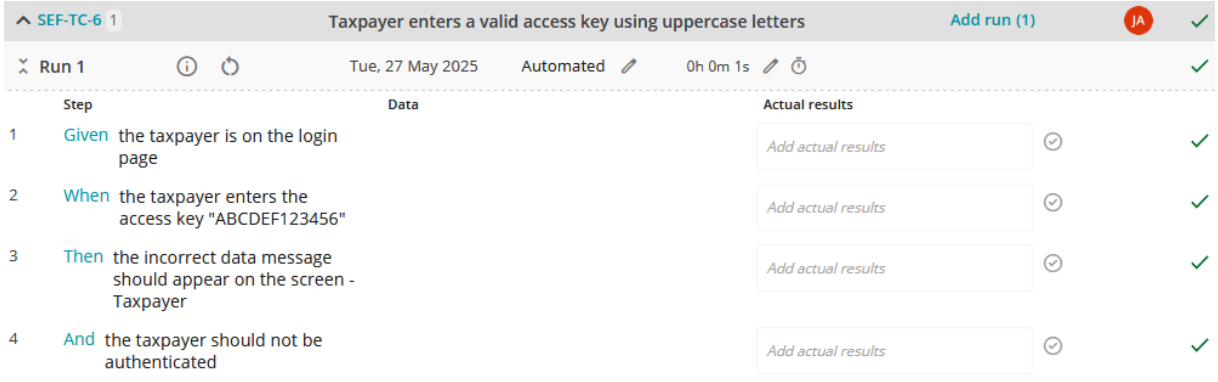

*Fonte: Elaborado pelo autor, 2025*

Figura 5 – Caso de teste “Taxpayer selects the access key field but leaves it empty”



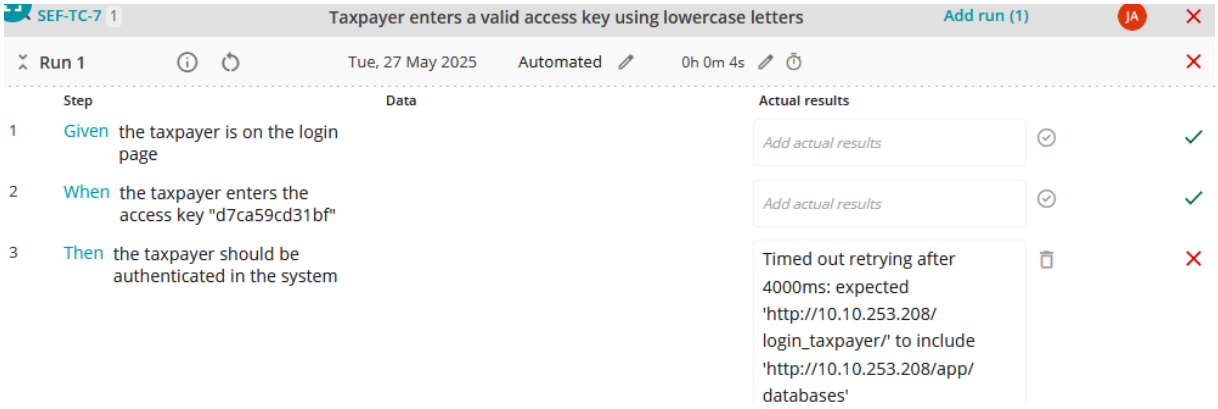

*Fonte: Elaborado pelo autor, 2025*

Figura 6 – Caso de teste “Taxpayer enters a valid access key using uppercase letters”



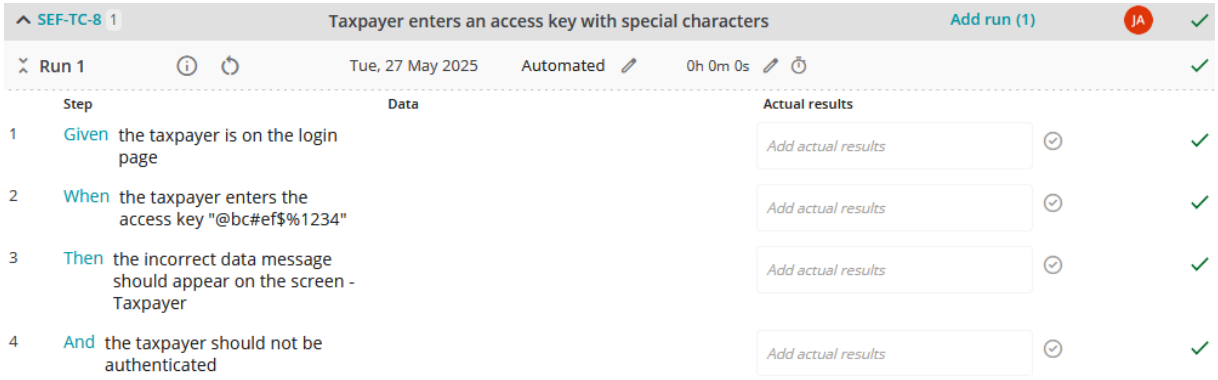

*Fonte: Elaborado pelo autor, 2025*

Figura 7 – Caso de teste “Taxpayer enters a valid access key using lowercase letters”



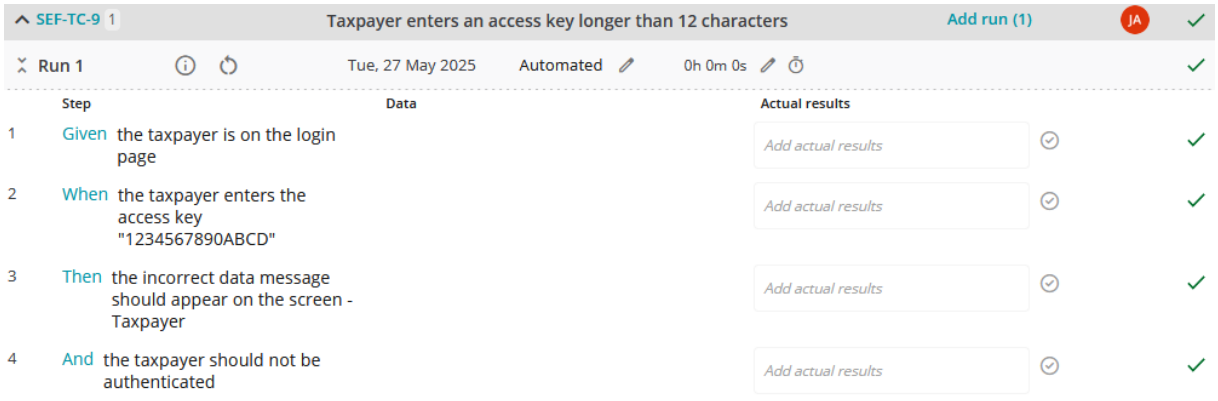

*Fonte: Elaborado pelo autor, 2025*

Figura 8 – Caso de teste “Taxpayer enters an access key with special characters”



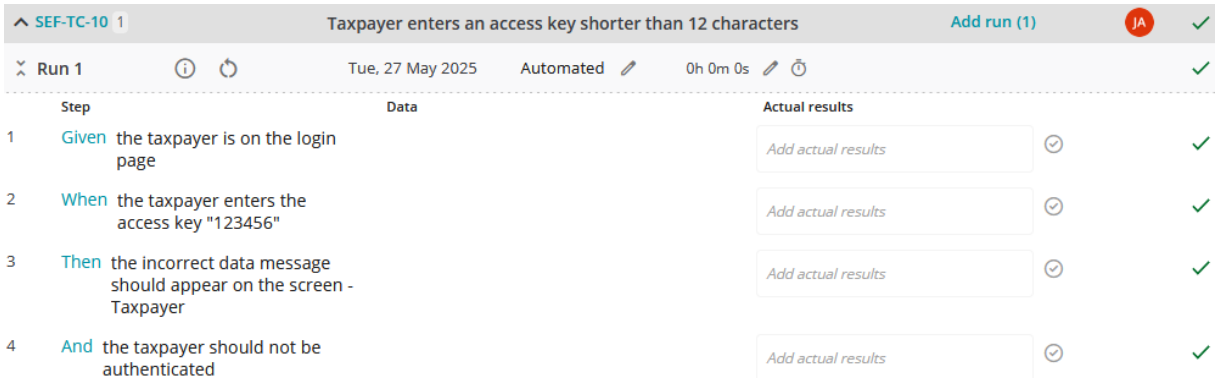

*Fonte: Elaborado pelo autor, 2025*

Figura 9 – Caso de teste “Taxpayer enters an access key longer than 12 characters”



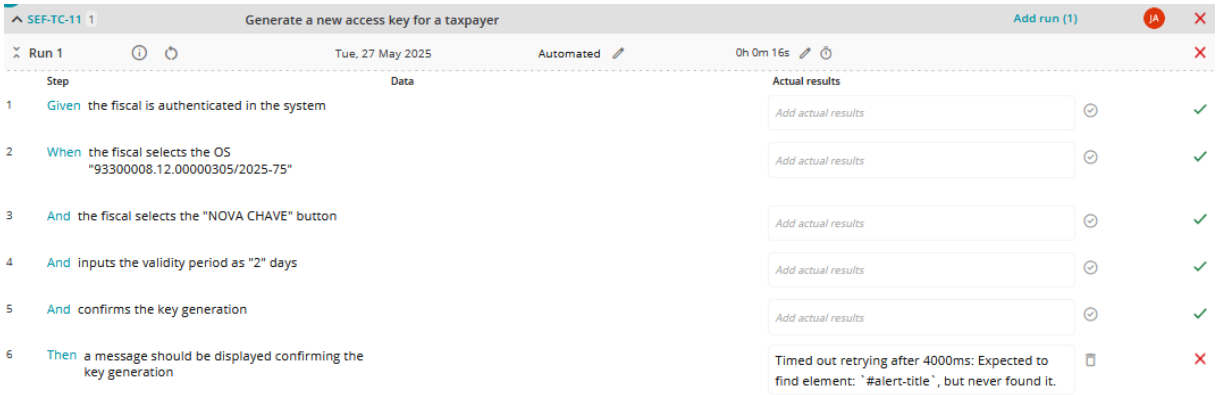

*Fonte: Elaborado pelo autor, 2025*

Figura 10 – Caso de teste “Taxpayer enters an access key shorter than 12 characters”



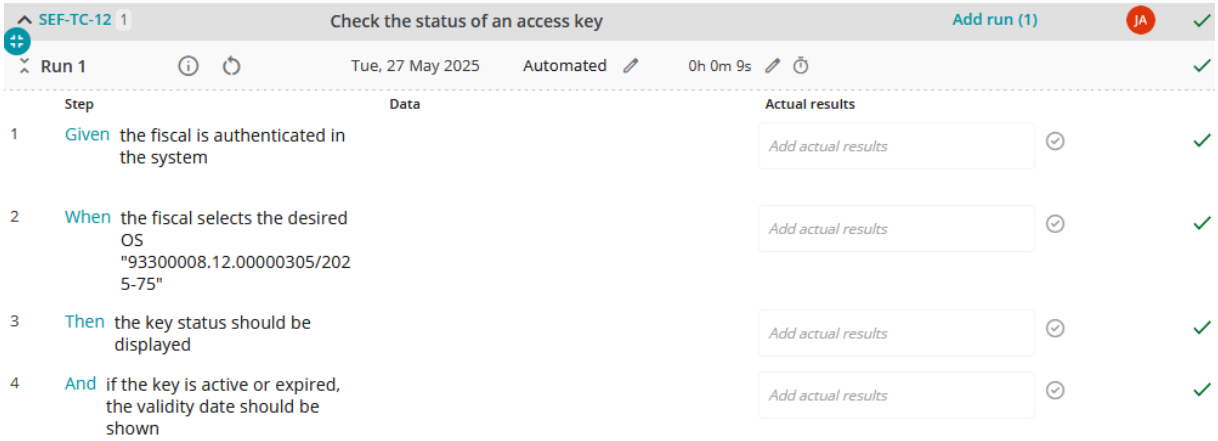

*Fonte: Elaborado pelo autor, 2025*

Figura 11 – Caso de teste “Generate a new access key for a taxpayer”



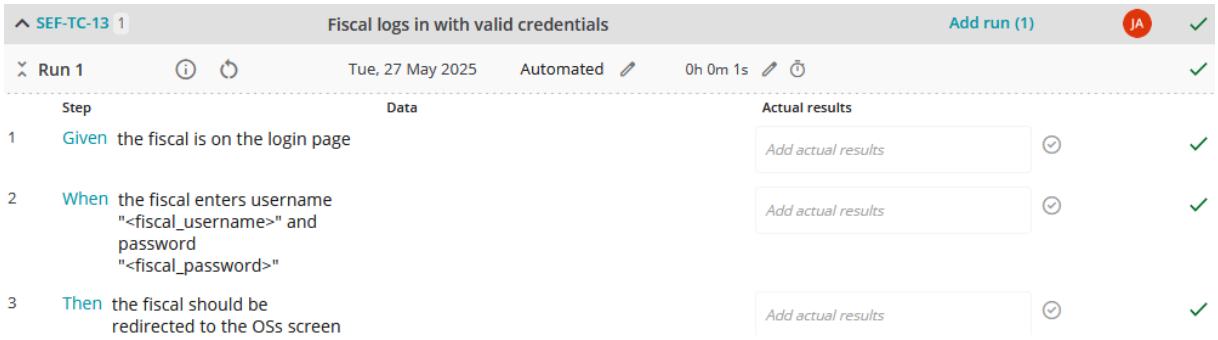

*Fonte: Elaborado pelo autor, 2025*

Figura 12 – Caso de teste “Check the status of an access key”



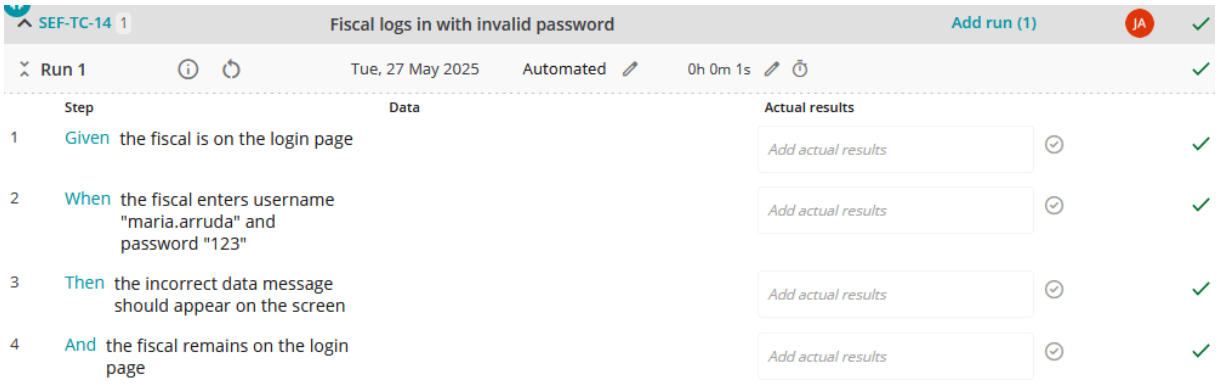

*Fonte: Elaborado pelo autor, 2025*

Figura 13 – Caso de teste “Fiscal logs in with valid credentials”



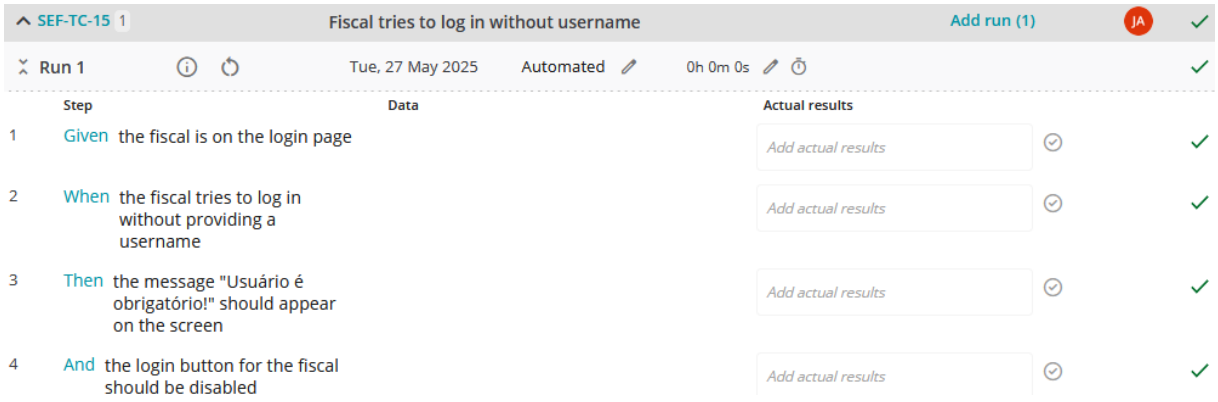

*Fonte: Elaborado pelo autor, 2025*

Figura 14 – Caso de teste “Fiscal logs in with invalid password”



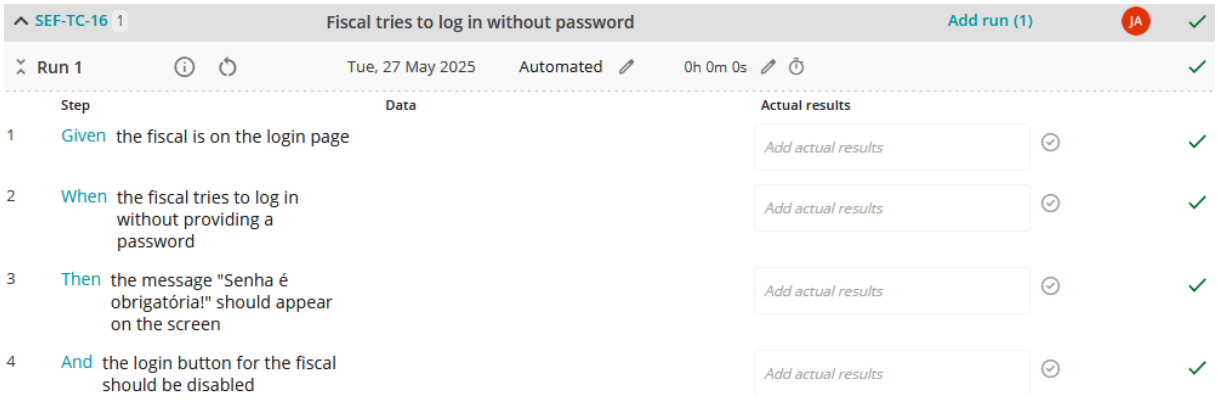

*Fonte: Elaborado pelo autor, 2025*

Figura 15 – Caso de teste “Fiscal tries to log in without username”



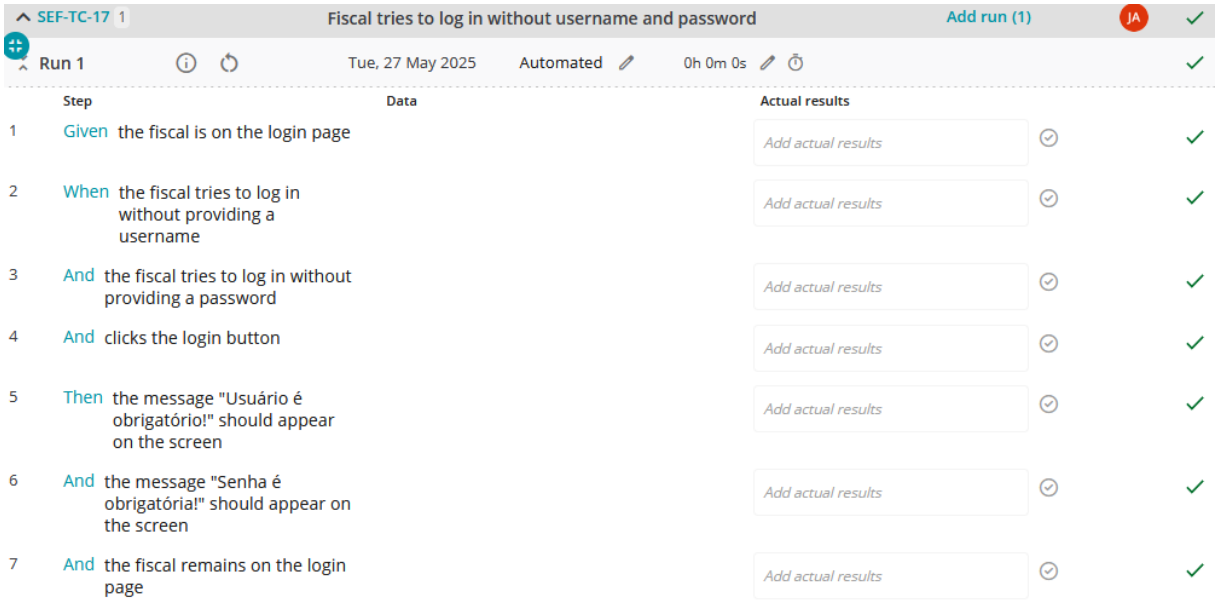

*Fonte: Elaborado pelo autor, 2025*

Figura 16 – Caso de teste “Fiscal tries to log in without password”



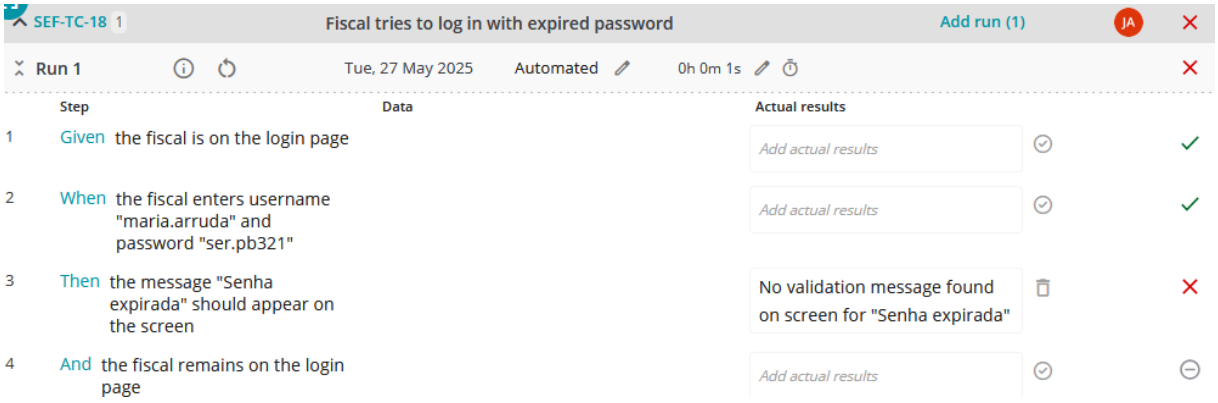

*Fonte: Elaborado pelo autor, 2025*

Figura 17 – Caso de teste “Fiscal tries to log in without username and password”




*Fonte: Elaborado pelo autor, 2025*

Figura 18 – Caso de teste “Fiscal tries to log in with expired password”




*Fonte: Elaborado pelo autor, 2025*