Uma Abordagem Unificada para Testes Automatizados em Arquiteturas Web Modernas

João Victor de Morais
Universidade Federal de Lavras
Lavras, Minas Gerais, Brasil
joao.reis9@estudante.ufla.br

Rafael Durelli
Universidade Federal de Lavras
Lavras, Minas Gerais, Brasil
rafael.durelli@ufla.br

Ricardo Terra
Universidade Federal de Lavras
Lavras, Minas Gerais, Brasil
terra@ufla.br

Resumo

A crescente adoção de arquiteturas web modernas tornou a automação de testes essencial à qualidade e à evolução dos sistemas. Entretanto, suítes automatizadas ainda enfrentam desafios relacionados à fragilidade, ao elevado custo de manutenção e ao acoplamento com ferramentas de automação. Este trabalho propõe uma abordagem unificada para testes funcionais automatizados, integrando testes de API e interface Web em uma arquitetura modular orientada à reutilização, ao desacoplamento e à manutenção evolutiva. Como contribuições, o trabalho sistematiza vinte requisitos técnicos, propõe uma arquitetura que os materializa e a avalia longitudinalmente em um projeto real de grande porte. Em aproximadamente doze meses de uso contínuo, mais de 80% dos 97 *Merge Requests* adicionaram exclusivamente novos cenários, enquanto apenas três envolveram correções em testes existentes. Isso indica que a abordagem sustentou a expansão de uma suíte com 300 casos de teste, mantendo estabilidade e baixa demanda por manutenção corretiva.

Palavras-chave

Testes de Software, Testes Funcionais Automatizados, Testes de API

1 Introdução

A crescente adoção de arquiteturas web modernas, caracterizadas por interfaces ricas e aplicações distribuídas baseadas em serviços, tem aumentado a complexidade do desenvolvimento e da validação de software [22]. Nesse contexto, a automação de testes tornou-se essencial para apoiar práticas de integração e entrega contínuas, permitindo validar funcionalidades de forma repetitiva e confiável em diferentes camadas da aplicação, como APIs e interfaces Web [7].

Apesar da ampla disponibilidade de ferramentas de automação, como Selenium[1], Playwright[2] e Cypress[3], a construção e a manutenção de suítes de testes continuam representando desafios relevantes [6]. Problemas como fragilidade dos testes (*flakiness*), elevado custo de manutenção, duplicação de código e acoplamento entre os casos de teste e a tecnologia utilizada comprometem a evolução da automação ao longo do ciclo de vida do software [7, 16, 23]. Embora a literatura apresente diversas ferramentas e técnicas de automação, ainda há espaço para abordagens sistematizadas voltadas ao desenvolvimento e à manutenção do próprio código de teste [31].

Em arquiteturas web, testes de API e de ponta a ponta (*end-to-end*) são empregados como estratégias complementares [2]. Os testes de API oferecem maior rapidez e robustez para validar contratos e integrações entre serviços [27], enquanto os testes *end-to-end* aproximam-se da experiência real do usuário e validam fluxos completos por meio da interface gráfica, embora sejam mais suscetíveis a problemas de sincronização e mudanças na interface [8, 28]. Em conjunto, esses níveis apoiam testes de regressão, permitindo verificar se modificações comprometem funcionalidades previamente validadas [1, 25].

Diante desse cenário, este trabalho propõe uma abordagem unificada para testes funcionais automatizados em arquiteturas web modernas. A proposta integra testes de API e de interface Web em uma arquitetura modular estruturada em três camadas principais: (i) um *core* genérico reutilizável que abstrai o motor de automação; (ii) uma camada de extensão específica do sistema que encapsula componentes visuais e regras de negócio; e (iii) uma estratégia híbrida que combina preparação de dados via API, execução na interface e validação direta no banco de dados. Essa estrutura promove o desacoplamento entre a lógica dos cenários de teste e os mecanismos responsáveis por sua execução, reduzindo o impacto de mudanças tecnológicas, aumentando o reúso de componentes e facilitando a evolução da suíte.

Como contribuições, o trabalho: (i) sistematiza vinte requisitos técnicos para o desenvolvimento de suítes de testes automatizados sustentáveis; (ii) propõe uma arquitetura que os materializa em componentes reutilizáveis e que encapsulam as dependências da ferramenta de automação; e (iii) avalia longitudinalmente sua aplicação em um projeto real de grande porte. Durante aproximadamente doze meses de uso contínuo, mais de 80% dos 97 *Merge Requests* adicionaram exclusivamente novos cenários, enquanto apenas três envolveram correções em testes existentes. Isso indica que a abordagem sustentou a expansão de uma suíte com 300 casos de teste, mantendo estabilidade e baixa demanda por manutenção corretiva. A contribuição não reside nas práticas isoladas, mas em sua sistematização e materialização conjunta em uma arquitetura unificada para testes de API e interface Web.

Este artigo está organizado da seguinte forma. A Seção 2 apresenta os requisitos técnicos da abordagem; a Seção 3 descreve sua arquitetura e implementação; a Seção 4 apresenta a avaliação e discute seus resultados; e a Seção 5 conclui o artigo.

2 Levantamento de Requisitos Técnicos

A sustentabilidade de suítes de testes funcionais exige enfrentar problemas como a intermitência (*flakiness*) e o elevado custo de manutenção. Para fundamentar a abordagem proposta neste trabalho, estabelecem-se os requisitos técnicos descritos a seguir. Esses requisitos não foram derivados de uma revisão sistemática da literatura, mas consolidados a partir da combinação entre padrões e práticas documentados na literatura de automação de testes e problemas observados durante a evolução da suíte estudada. Cada requisito é

[1] https://www.selenium.dev/
[2] https://playwright.dev/
[3] https://www.cypress.io

acompanhado das referências que fundamentam o problema tratado ou a solução recomendada, orientando a especificação da solução desenvolvida e a mitigação sistemática desses problemas:

[RT01] Abstração do motor de execução: A dispersão do conhecimento técnico da ferramenta pelos cenários eleva a fragilidade da suíte [9]. O isolamento da biblioteca de automação (e.g., Selenium ou Playwright) por uma camada controladora desacopla a lógica dos casos de teste das ferramentas subjacentes, mitigando o impacto de atualizações de dependências [7].

[RT02] Encapsulamento de sincronismo: Falhas de espera assíncrona (*Async Wait*) constituem a causa primordial de testes intermitentes (*flaky tests*) [18]. É obrigatório substituir pausas fixas (*sleep*) por esperas inteligentes (*waitFor*) encapsuladas, tornando a execução mais fluente e previsível [4].

[RT03] Mapeamento semântico de identificadores: A dispersão de seletores técnicos eleva o custo de manutenção, pois alterações na interface quebrariam múltiplos testes simultaneamente [7]. Os localizadores devem ser centralizados em um repositório de objetos, associando identificadores físicos voláteis a constantes semânticas de negócio [9] (e.g., mapear o seletor `id="login-button"` para a constante `BOTAO_LOGIN`), garantindo que atualizações visuais exijam correção em apenas um único ponto.

[RT04] Priorização de identificadores estáticos: Caminhos estruturais complexos como XPath quebram facilmente e ampliam o esforço de reparo do código [16]. A modelagem deve priorizar seletores baseados em identificadores estáticos (e.g., ID) na interface.

[RT05] Mapeamento entre objeto e tela: O acoplamento entre a mecânica visual e as validações de negócio eleva a fragilidade e prejudica a reusabilidade do código [16]. Este requisito estabelece o mapeamento de cada tela para uma classe controladora que abstrai a complexidade da interface em métodos funcionais de alto nível (e.g., o método `fillForm` gerencia a localização e a inserção de dados nos campos) [7]. Essa camada de tradução isola os casos de teste de mudanças estruturais na página, garantindo que o esforço de automação se concentre estritamente na lógica do cenário de negócio e permitindo manutenções centralizadas [7, 16].

[RT06] Padronização operacional de telas: A falta de uniformidade no *design* de interações eleva a introdução de defeitos em sistemas modulares [31]. Operações comuns (e.g., buscar um registro e colocar em modo de edição) devem seguir assinaturas padronizadas para acelerar a construção de novos cenários [20]. Essa uniformidade pode otimizar a curva de aprendizado da equipe e promover um aumento significativo na produtividade ao reduzir o esforço de codificação e manutenção [9, 25].

[RT07] Coesão funcional por interface: A dispersão de escopo em um único roteiro dificulta a localização de defeitos [20]. Cada unidade de teste deve restringir-se estritamente à fronteira do seu respectivo módulo funcional. Por exemplo, um caso de teste do módulo `Financeiro` não deve realizar asserções ou fluxos do módulo de `Estoque`, o que agiliza o diagnóstico de falhas frente à especificação [1].

[RT08] Eliminação de valores literais: A dispersão de valores literais em asserções compromete a legibilidade do código [26]. Cadeias de caracteres e valores numéricos com significado semântico devem ser substituídos por constantes, e.g., mapear o texto físico `"Usuário ou senha inválidos"` para a constante `MSG_ERRO_LOGIN`.

[RT09] Abstração de dados estáticos por enumeradores: O uso de valores numéricos arbitrários (*magic numbers*) nos casos de teste compromete a clareza semântica e a evolução dos testes [26]. Elementos que representam tabelas estáticas do banco de dados ou domínios fixos de negócio devem ser mapeados em enumeradores (*enums*) centralizados (e.g., substituir o ID numérico 1 pelo enumerador `Perfil.ADMINISTRADOR`), garantindo a legibilidade e a perfeita sincronização com o modelo de dados da aplicação [19].

[RT10] Centralização de dados dependentes: A criação de dados de teste deve ser encapsulada em métodos específicos de configuração (*fixtures*), evitando duplicação e facilitando a manutenção [10]. Cada caso de teste deve utilizar essas estruturas para criar automaticamente todos os registros necessários à sua execução. Por exemplo, a criação de um `Estabelecimento` deve cadastrar também o `Produtor` e os demais dados obrigatórios relacionados. Essa abordagem reduz dependências entre testes e contribui para sua execução independente [17, 32].

[RT11] Injeção de estado via *backend*: O uso da interface gráfica para configurar pré-condições de dados gera casos de teste lentos e instáveis [20]. Adota-se o padrão de configuração via porta dos fundos (*Back Door Setup*), utilizando chamadas de API para estabelecer o estado inicial dos casos de teste [12].

[RT12] Atomicidade e independência: Cenários que dependem de dados gerados por execuções prévias elevam a fragilidade da suíte [18]. Esse requisito estabelece que cada caso de teste deve ser isolado e auto-contido, sendo o único responsável por prover e criar as entidades necessárias para sua execução. O isolamento de dados elimina o acoplamento temporal e evita a propagação de falhas em cascata na sequência de testes [18, 29].

[RT13] Isolamento de estado global: Casos de teste que alteram recursos compartilhados, como o banco de dados, podem gerar falhas falsas devido a dados residuais deixados por execuções anteriores [18]. Para mitigar esse problema, esse requisito exige que a infraestrutura da suíte execute rotinas automáticas de limpeza de persistência antes e após cada cenário [29]. Esse ciclo de restauração assegura que cada teste inicie em um ambiente limpo e restabeleça o estado original imediatamente após sua conclusão, mitigando a intermitência por poluição de dados [18, 29].

[RT14] Validação na camada de dados: Validações restritas à interface visual podem mascarar defeitos severos devido a falhas de renderização ou *cache* [20]. Deve-se prover uma forma para inspecionar diretamente o banco de dados da aplicação para garantir a integridade real dos dados gravados.

[RT15] Componentes utilitários compartilhados: A duplicação de lógicas secundárias infla o código de teste e eleva o esforço de manutenção [9]. Métodos auxiliares (e.g., formatadores de datas do sistema ou geradores de números de CPF/CNPJ válidos) devem ser centralizados em uma classe utilitária global, evitando a repetição de código nos casos de teste.

[RT16] Separação de dados de entrada: Mitiga a rigidez e o custo de manutenção ao isolar a lógica de teste dos conjuntos de dados (estratégia de *Data-Driven Testing*) [9]. O uso de arquivos externos (e.g., JSON) permite que um único caso de teste processe múltiplas variações de entrada sem a necessidade de modificar o código-fonte, garantindo a reusabilidade e a rápida modificabilidade da suíte [24].

[RT17] Execução não interativa: A dependência de ambiente gráfico restringe a execução automatizada de testes em servidores de integração contínua [30]. A suíte deve suportar execução via linha de comando (modo *headless*) para integração nativa em *pipelines* de CI/CD [7].

[RT18] Observabilidade da execução: A falta de visibilidade sobre o comportamento dos testes compromete a análise e reprodução de falhas [11, 14]. A suíte deve gerar *logs* estruturados contendo informações relevantes sobre a execução dos casos de teste (e.g., etapas executadas e parâmetros utilizados), fornecendo evidências que facilitem a identificação e correção de erros.

[RT19] Identificação e autoria dos cenários: A ausência de informações básicas sobre os casos de teste dificulta o diagnóstico de problemas e a comunicação entre a equipe [21]. Cada caso de teste deve conter campos padronizados de identificação, detalhando o nome do autor, a data de criação e uma breve descrição do objetivo do teste.

[RT20] Vínculo com ferramentas de gestão de tarefas: A falta de conexão entre os testes automatizados e o fluxo de desenvolvimento impede o acompanhamento real da saúde do projeto [13]. Os cenários de teste devem ser mapeados e associados aos identificadores de tarefas ou *bugs* em ferramentas de gerenciamento (e.g., chaves de *issues* do Jira), permitindo a rastreabilidade entre o código e o planejamento técnico.

3 Arquitetura Proposta e Implementação

Com base nos vinte requisitos técnicos estabelecidos na Seção 2, propõe-se uma arquitetura para a construção e manutenção de suítes de testes funcionais automatizados. Organizada em componentes com responsabilidades bem definidas, a abordagem busca (i) reduzir o acoplamento entre os casos de teste e a tecnologia utilizada, (ii) aumentar a reutilização de código e (iii) facilitar a evolução da suíte.

Embora a implementação de referência tenha sido desenvolvida em Java com Selenium WebDriver, a arquitetura é independente de tecnologia específica e pode ser reproduzida em outras linguagens e motores de automação, desde que preservados os requisitos técnicos definidos anteriormente.

3.1 Síntese e Visão Geral da Arquitetura

A Figura 1 apresenta a arquitetura proposta, distinguindo o núcleo reutilizável do *framework* de sua instanciação em uma aplicação específica e evidenciando as interações entre as camadas e os elementos externos.

Organização geral: A arquitetura divide-se em dois blocos integrados. O superior, denominado *Zettalenium*, concentra o núcleo genérico e reutilizável da solução. O inferior representa sua instanciação para o sistema *System*, reunindo as customizações técnicas e os cenários específicos da aplicação. Essa separação mantém o núcleo do *framework* independente das regras de negócio do sistema sob teste.

***Framework* (Zettalenium):** Reúne a infraestrutura comum da suíte, organizada em:

- *Core dos casos de teste:* A classe abstrata `ZettaTestCase` concentra comportamentos e inicializações compartilhados. Dela derivam `ZettaAPITestCase`, para testes de API, e `ZettaWebTestCase`, para testes Web.
- *Componentes de apoio:* Incluem `Database`, responsável pelo acesso ao banco de dados e por encapsular a lógica de persistência; `BasicRest`, encarregado de implementar as requisições HTTP da camada de API; e `TestUtils`, que reúne funções utilitárias compartilhadas.

Instanciação do Zettalenium: Estende o núcleo do *framework* para a aplicação específica, organizando os testes em duas frentes:

- *Testes de API:* A classe `SystemAPITestCase` estende `ZettaAPITestCase`, concentrando métodos comuns aos testes de serviços. As funcionalidades específicas são implementadas em classes como `Feature1APITestCase` a `FeatureNAPITestCase`.
- *Testes Web:* A classe `SystemWebTestCase` estende `ZettaWebTestCase` e reúne comportamentos comuns aos testes *end-to-end*, como a interação com componentes presentes em diferentes telas. As funcionalidades são organizadas em classes como `Feature1WebTestCase` a `FeatureNWebTestCase`. Essa camada utiliza `SystemFrontEnd`, que encapsula tanto a ferramenta de automação, como Selenium ou Playwright, quanto as particularidades da biblioteca visual da aplicação. Em vez de expor apenas comandos genéricos de navegação, esse componente oferece operações especializadas para manipular elementos customizados e assíncronos, como os do Angular Material[4].

Integrações externas: À direita do diagrama, os fluxos representam as interações da arquitetura com sistemas externos:

- `Database` acessa o SGBD para realizar verificações e limpar o estado dos testes.
- `BasicRest` comunica-se com o *backend* por meio de chamadas à API.
- `TestUtils` gerencia a leitura e a manipulação de arquivos JSON e centraliza métodos utilitários reutilizáveis empregados em diferentes etapas da infraestrutura de testes.
- `SystemFrontEnd` aciona a ferramenta de automação e o navegador para interagir com a interface da aplicação.

Essa arquitetura modular e desacoplada busca restringir modificações em ferramentas de automação, *endpoints* de API e estruturas de tela às respectivas camadas periféricas, preservando a estabilidade do núcleo do *framework* e facilitando a manutenção evolutiva dos testes.

[4] https://material.angular.dev

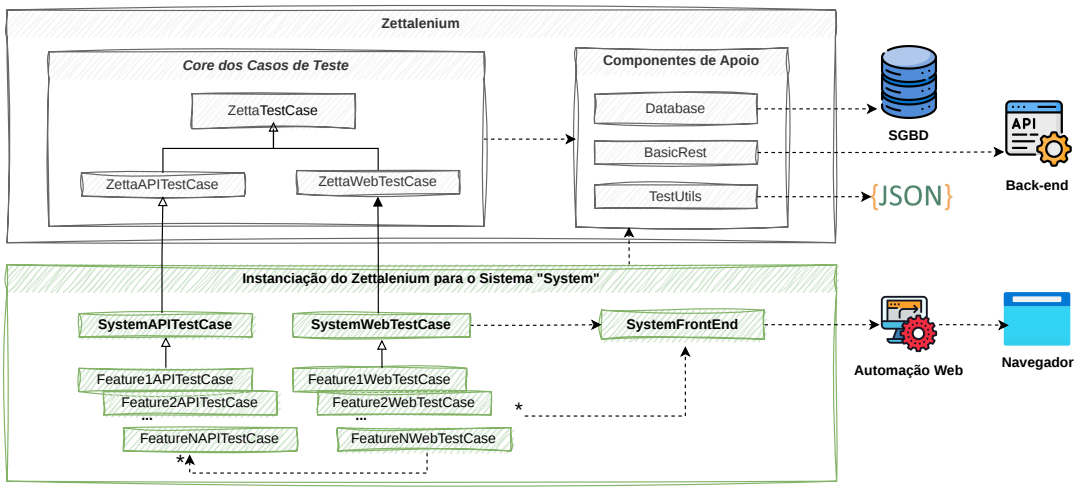

Figura 1: Visão geral da arquitetura e fluxo de interações do *framework* de testes Zettalenium.

3.2 Implementação

Esta seção apresenta a instanciação prática dos três núcleos da arquitetura, demonstrando sua viabilidade técnica por meio de cenários fictícios e implementações simplificadas em Java com Selenium WebDriver. Os exemplos evidenciam como as responsabilidades são distribuídas e isoladas entre as camadas do *framework*.

Inicialmente, apresenta-se a camada de abstração do *front-end* (Seção 3.2.1), centrada no componente `SystemFrontEnd`, que encapsula a ferramenta de automação e fornece operações de interação com a interface. Em seguida, descrevem-se os casos de teste Web (Seção 3.2.2), representados por classes `FeatureWebTestCase` que herdam os comportamentos comuns de `SystemWebTestCase`. Por fim, apresentam-se os testes de API (Seção 3.2.3), implementados por classes `FeatureAPITestCase` que reutilizam os recursos de `SystemAPITestCase` para gerenciar estados e preparar dados complementares.

3.2.1 Instanciação da Camada de Abstração do Front-end. A camada de abstração da ferramenta de automação intermedeia a lógica de negócio dos casos de teste e as APIs de baixo nível do motor de execução *web*. Seu objetivo é desacoplar os testes das ferramentas de execução, reduzindo o impacto de eventuais mudanças tecnológicas. Além disso, oculta as particularidades operacionais da interface gráfica e fornece métodos de interação unificados e semânticos.

O Código 1 apresenta o núcleo da implementação desse componente, mostrando como as interações com os elementos da tela e os mecanismos de sincronização são gerenciados de forma transparente.

O isolamento do ecossistema de automação — nesse caso, o Selenium WebDriver — em relação aos casos de teste é evidenciado no construtor da classe. Na linha 5, o ciclo de vida e a obtenção da instância do *driver* são delegados a `ChromeCrawler.getWebDriver()`, impedindo que os testes gerenciem diretamente a abertura e o encerramento do navegador. Além disso, o método `fill` (linhas 19 a 42) identifica dinamicamente o tipo de *tag* do componente *front-end*. Componentes complexos da biblioteca visual, como os elementos do Angular Material tratados no `switch` das linhas 28 a 41, têm suas particularidades de interação encapsuladas. Assim, o desenvolvedor realiza o preenchimento por meio de uma única chamada, atendendo ao requisito de *Abstração do motor de execução [RT01]*.

A centralização da criação do navegador também contribui para o requisito de *Execução não interativa [RT17]*. Na linha 5, a instância do navegador é obtida por `ChromeCrawler.getWebDriver()`, evitando que os casos de teste manipulem diretamente o `WebDriver`. Essa decisão concentra a configuração do navegador em um único componente, permitindo definir propriedades específicas do ambiente externamente. Assim, o modo *headless* pode ser ativado no `ChromeCrawler` sem modificar os cenários, possibilitando a execução da mesma suíte em estações de desenvolvimento e servidores de integração contínua sem interface gráfica.

A localização dos componentes também é centralizada por meio de identificadores estáticos, reduzindo a fragilidade associada a seletores estruturais, como XPath. O método `getWebElementById` (linhas 44 a 52) fundamenta as demais operações da classe e, na linha 46, localiza os componentes por seus IDs. O mesmo princípio é aplicado pelo método `click` na linha 8, atendendo ao requisito de *Priorização de identificadores estáticos [RT04]*.

A intermitência causada pelo carregamento assíncrono é tratada na própria infraestrutura. No método `click` (linhas 8 a 17), a localização e o acionamento do componente são encapsulados em um `Runnable`. Nas linhas 13 e 14, o objeto `defaultWait` aguarda até que a condição `elementToBeClickable` seja satisfeita. Na linha 16, o método `forceRetry` acrescenta resiliência a atrasos ou variações do ambiente. Essa combinação substitui pausas fixas e atende ao requisito de *Encapsulamento de sincronismo [RT02]*.

3.2.2 Instanciação dos Testes Web. A organização dos casos de teste constitui o núcleo da arquitetura proposta, concentrando a lógica dos cenários de negócio e coordenando a interação com os demais componentes da automação. Nessa camada, os testes são estruturados em operações de alto nível que representam funcionalidades do sistema, reduzindo a exposição a detalhes técnicos da interface gráfica, da persistência de dados e dos mecanismos de integração.

O Código 2 apresenta um exemplo fictício de caso de teste focado no fluxo de cadastro de uma espécie animal. O cenário de ponta a

```
public class SystemFrontEnd extends AbstractFrontEnd {
  protected static final Duration DEFAULT_DURATION =
      Duration.ofSeconds(Props.delay());

  public SystemFrontEnd() {
    super(ChromeCrawler.getWebDriver(), DEFAULT_DURATION);
  }

  public final void click(final String id) {
    Runnable click = () -> {
      LOGGER.trace("click :: id={}", id);
      By by = By.cssSelector(String.format("[id$='%s']", id));

      defaultWait.until(ExpectedConditions
        .elementToBeClickable(by)).click();
    };
    this.forceRetry(click);
  }

  public void fill(final String elementId, final Object text) {
    if (text == null) {
        return;
    }
    WebElement we = this.getWebElementById(elementId);
    if (we == null) {
      throw new TestException("Campo nao foi encontrado");
    }

    switch (we.getTagName()) {
      case "mat-select":
      case "mat-form-field":
        this.matSelect(we, text);
        return;
      case "mat-radio-group":
        this.matRadioGroup(we, text.toString());
        return;
      case "mat-checkbox":
        this.matCheckBox(we, (Boolean) text);
        return;
      case "input":
        this.matInput(we, text);
    }
  }

  private WebElement getWebElementById(final String id) {
    Callable<WebElement> getter = () -> {
      By by = By.cssSelector(String.format("[id$='%s']", id));
      WebElement we = this.getAndEnsureUniqueness(by);
      this.moveToElementById(we);
      return we;
    };
    return this.forceRetry(getter, "getWebElementById");
  }
  ...
}
```

Código 1: Camada de abstração da interface gráfica.

```
public class EspecieWebTestCase extends SystemWebTestCase {
  private static final Logger LOGGER =
      LogManager.getLogger(EspecieWebTestCase.class);

  public static class EspecieForm {
    private Integer codigoMapa;
    private GrupoEspecie grupo;
    private String nomeEspecie;
    private String habitat;
    private String nomeCientifico;
  }

  public static void create(final SystemFrontEnd view, final
      EspecieForm form) {
    LOGGER.trace("Criando uma nova espécie.");
    openMenu(view, Menu.ESPECIE);

    fillForm(view, form);

    view.click(Botao.ADICIONAR);
    view.assertMessage(MSG_CRIACAO);

    checkOnDatabase(form);
  }

  public static void fillForm(final SystemFrontEnd view, final
      EspecieForm form) {
    if (form != null) {
      view.fill("grupo-select", form.grupo.getLabel());
      view.fill("nome-especie-input", form.nomeEspecie);
      view.fill("nome-cientifico-input", form.nomeCientifico);
      view.fill("habitat-select", form.habitat);
      view.fill("codigo-mapa-input", form.codigoMapa);
    }
  }

  private static void checkOnDatabase(EspecieForm form) {
    Database.assertResultInDBMS(new
      SQL("especie-check.sql").addParam("codigoMapa",
      form.codigoMapa),
        form.grupo.getValue(),
        form.nomeEspecie,
        form.nomeCientifico,
        TestUtils.normalizeStr(form.habitat)
    );
  }

  @Test
  @ZettaTest(
    qaAnalyst = "Maria do Carmo",
    description = "Fluxo de criação de uma espécie.",
    createdAt = "27/06/2026",
    sprint = 3,
    cards = {905, 919, 1009, 1337}
  )
  @Clean(value = {"Especie:1234567890"}, otherClassesTestData =
      EspecieAPITestCase.class)
  public void testFluxoSimples() {
    LOGGER.trace("Cadastrando entidades complementares pela API.");
    EspecieAPITestCase.TestData defaultTestData =
      EspecieAPITestCase.insertDefaultTestData();

    LOGGER.trace("Carregando dados do cenário.");
    EspecieForm form = TestUtils.fromJson(EspecieForm.class,
      "create.json");
    form.habitat = defaultTestData.getHabitat();

    create(view, form);
  }
  ...
}
```

Código 2: Exemplo da camada de casos de teste *web*.

ponta inicia-se com a preparação do estado do sistema, delegando a criação das dependências à camada de API; em seguida, carrega os dados de entrada em formato JSON, preenche e submete os campos na interface gráfica e, por fim, verifica a persistência das informações diretamente no banco de dados.

A declaração da classe `EspecieWebTestCase` na linha 1 materializa o requisito de *Coesão funcional por interface [RT07]*. Seu escopo restringe-se ao módulo de Espécies, concentrando ações, interações de tela e asserções em um único domínio de negócio. Essa separação evita a inclusão de fluxos de outros módulos no mesmo roteiro, reduz o acoplamento e facilita o diagnóstico de falhas. A estrutura `EspecieForm` (linha 4) representa os dados manipulados pela tela e estabelece a correspondência entre o domínio de negócio e os elementos da interface, materializando o requisito de *Mapeamento entre objeto e tela [RT05]*.

Os métodos `create` (linha 12) e `fillForm` (linha 24) exemplificam a padronização das operações do módulo. Essa padronização pode abranger outras operações, como `find`, responsável pela busca de registros, e `putOnEditMode`, que busca o registro e o coloca em modo de edição. Em vez de repetir sequências de interação com a interface em cada cenário, essas ações são encapsuladas em métodos reutilizáveis de alto nível, reduzindo duplicações e atendendo ao requisito de *Padronização operacional de telas [RT06]*.

Outro aspecto relevante é a utilização de identificadores semânticos, como `Menu.ESPECIE` (linha 14) e `Botao.ADICIONAR` (linha 18),

em substituição a seletores físicos. Essa estratégia reduz o acoplamento dos testes aos detalhes de implementação da aplicação e atende ao requisito de *Mapeamento semântico de identificadores [RT03]*.

Na linha 19, o método de validação de mensagens utiliza a constante `MSG_CRIACAO`, materializando o requisito de *Eliminação de valores literais [RT08]*. Ao centralizar mensagens de *feedback* e textos informativos em constantes globais, a arquitetura reduz o impacto de pequenas alterações ortográficas ou revisões gramaticais na interface do sistema sob teste.

Adicionalmente, a linha 26 evidencia o requisito de *Abstração de dados estáticos por enumeradores [RT09]* pelo uso de `form.grupo`, cujo valor `GrupoEspecie.BOVIDEOS` é carregado a partir do JSON (linha 57). Enquanto o requisito anterior isola textos da interface, este requisito atua na representação tipada do domínio de negócio. O enumerador substitui identificadores numéricos ou chaves primárias fixas do banco de dados, tornando o código semanticamente expressivo e menos dependente de variações nas cargas de dados dos ambientes.

As chamadas `LOGGER.trace()` (linhas 13 e 53) registram o fluxo de execução dos cenários e auxiliam na identificação de falhas. Esse mecanismo atende ao requisito de *Observabilidade da execução [RT18]*, fornecendo rastros das etapas executadas pelo teste.

A integração e o reúso entre as camadas são evidenciados na linha 54. O método `EspecieAPITestCase.insertDefaultTestData()` atende ao requisito de *Injeção de estado via backend [RT11]* ao recorrer à camada de testes de API, detalhada na Seção 3.2.3, para criar de forma síncrona as entidades complementares exigidas pelo fluxo. Essa estratégia evita etapas adicionais na interface gráfica para preparar os dados, reduzindo o tempo de execução e contribuindo para a estabilidade da automação. O mesmo método atende ao requisito de *Centralização de dados dependentes [RT10]* ao utilizar uma *fixture* da camada de API que encapsula as entidades necessárias ao cenário. Assim, o teste *web* delega a geração dos dados complementares ao componente de integração e constrói seu próprio contexto de execução. Com isso, também atende ao requisito de *Atomicidade e independência [RT12]*, tornando-se autocontido e independente de dados deixados por execuções anteriores.

A anotação `@Clean` na linha 51 atende ao requisito de *Isolamento de estado global [RT13]* ao controlar declarativamente a limpeza do ambiente. O parâmetro `value` identifica a entidade e o código `1234567890`, utilizado pelo *framework* na consulta de exclusão apresentada no Código 3, executada antes e após o teste. Já `otherClassesTestData` inclui no ciclo de limpeza os dados criados por classes adicionais, como `EspecieAPITestCase.class`, reduzindo dependências entre os contextos da suíte.

```
1 delete from ESPECIE where CODIGO_MAPA = ':id';
```

Código 3: *Script* SQL de limpeza da entidade Espécie.

A arquitetura complementa o isolamento do ambiente com uma estratégia de validação multicamada, ampliando a verificação para além da interface gráfica. Na linha 21, o método `checkOnDatabase` (linhas 34 a 41) compara os dados enviados pelo formulário com os registros armazenados no banco de dados, atendendo ao requisito de *Validação na camada de dados [RT14]*. A consulta utilizada nessa verificação é apresentada no Código 4.

```
1 select e.GRUPO_ESPECIE,
2        e.NOME_ESPECIE,
3        e.NOME_CIENTIFICO,
4        h.NOME_HABITAT
5 from ESPECIE e
6 inner join HABITAT h on h.ID = e.ID_HABITAT
7 where e.CODIGO_MAPA = ':codigoMapa';
```

Código 4: Consulta SQL de verificação de persistência.

Essa abordagem é importante para mitigar falsos positivos, em que a operação é considerada bem-sucedida apenas porque a interface exibiu uma mensagem de sucesso ou não travou. A dupla verificação, na interface e no banco de dados, aumenta a confiabilidade do teste e ajuda a revelar defeitos mascarados por *cache* ou estados efêmeros do *front-end*.

Os componentes utilitários compartilhados concentram funcionalidades auxiliares reutilizadas por diferentes módulos, evitando duplicações e padronizando comportamentos, conforme o requisito de *Componentes utilitários compartilhados [RT15]*. Na linha 57, por exemplo, o método `TestUtils.fromJson()` lê e desserializa o arquivo externo apresentado no Código 5, transferindo seus dados para o objeto de formulário `EspecieForm`, definido na linha 4 do Código 2. Esse mecanismo separa os dados de entrada da lógica de execução do roteiro e atende ao requisito de *Separação de dados de entrada [RT16]*.

```
1 {
2   "codigoMapa": 1234567890,
3   "grupo": "GrupoEspecie.BOVIDEOS",
4   "nomeEspecie": "Boi Doméstico",
5   "habitat": "Pastagens",
6   "nomeCientifico": "Bos taurus"
7 }
```

Código 5: Exemplo de *payload* JSON para cadastro de espécie.

Como pode ser observado na linha 3, o campo `grupo` recebe o valor `"GrupoEspecie.BOVIDEOS"`, convertido automaticamente pela infraestrutura para o enumerador correspondente. Essa estratégia atende ao requisito de *Abstração de dados estáticos por enumeradores [RT09]* e facilita a manutenção, pois eventuais alterações no identificador do grupo ficam restritas à definição central do enumerador, sem a necessidade de modificar diversos arquivos JSON manualmente.

A mesma classe utilitária centraliza outras operações auxiliares, como a normalização de textos pelo método `normalizeStr` na linha 39. Essa centralização reduz redundâncias, simplifica a manutenção e promove consistência na manipulação de dados entre os cenários.

Por fim, a anotação `@ZettaTest` incorpora aos cenários metadados como autoria, descrição funcional, data de criação, *sprint* e associação com tarefas do projeto. Essa prática atende aos requisitos de *Identificação e autoria dos cenários [RT19]* e *Vínculo com ferramentas de gestão de tarefas [RT20]*.

3.2.3 Instanciação dos Testes de API. A instanciação dos testes de API encerra a apresentação prática da arquitetura. Esse núcleo demonstra a manipulação direta de requisições HTTP e a preparação ágil de estados para os demais testes. Uma descrição mais detalhada dessa camada pode ser encontrada em trabalho anterior dos autores [3].

```
1  public class EspecieAPITestCase extends SystemAPITestCase {
2    private static final Logger LOGGER =
         LogManager.getLogger(EspecieAPITestCase.class);
3    public final static String API_PATH = "/especie";
4    public final static String FILE_PATH = "especie/";
5
6    @Builder
7    public static class TestData {
8      private Integer idHabitat;
9      private String habitat;
10   }
11
12   @Clean({"Habitat:12345678"})
13   public static TestData insertDefaultTestData() {
14     LOGGER.trace("Cadastra um novo habitat.");
15     JSONObject habitatCreate =
         HabitatAPITestCase.novo(TestUtils.jsonFromFile("create.json"));
16     return TestData.builder()
17       .idHabitat(habitatCreate.getInt("id"))
18       .habitat(habitatCreate.getString("nome"))
19       .build();
20   }
21
22   public static JSONObject novo(JSONObject especie) {
23     LOGGER.trace("Salva uma nova espécie.");
24     return BasicRest.post(API_PATH, HttpStatus.SC_CREATED, especie,
         JSONObject.class);
25   }
26
27   @Test
28   @ZettaTest(...)
29   @Clean(value = {"Especie:5300100212"},
30   otherClassesTestData = {HabitatAPITestCase.class})
31   public void testFluxoSimples() {
32     LOGGER.trace("Cria o habitat que será vinculado à espécie.");
33     TestData defaultTestData = insertDefaultTestData();
34
35     LOGGER.trace("Salva uma nova espécie");
36     JSONObject especie = TestUtils.jsonFromFile("create.json");
37     especie.put("idHabitat", defaultTestData.getIdHabitat());
38     JSONObject resultCreate = novo(especie);
39     assertNotNull(resultCreate.optNumber("id"));
40   }
41   ...
42 }
```

Código 6: Exemplo da camada de casos de teste de API.

O Código 6 apresenta a implementação simplificada dos testes de API para o *endpoint* de espécies, mostrando como a infraestrutura de baixo nível é encapsulada para uso interno e por outros módulos. A estrutura contempla a preparação programática de dependências, como o cadastro prévio de um habitat, o encapsulamento de requisições HTTP POST e a execução de um cenário funcional que valida a criação de uma espécie sob regras de limpeza e isolamento da base.

A declaração da classe na linha 1 mostra a extensão de `SystemAPITestCase`, que fornece métodos-base reutilizáveis aos casos de teste de API. Esse núcleo desempenha um duplo papel: além de comportar os testes funcionais da API, atua como provedor rápido de estados para os testes *web*, conforme apresentado na subseção anterior. Esse segundo papel é concretizado pelo método `insertDefaultTestData()` (linha 13), que coordena a criação programática de dependências via *backend*, atendendo ao requisito de *Centralização de dados dependentes [RT10]*. As anotações `@Clean` (linhas 12 e 29) e `@ZettaTest` utilizam a mesma infraestrutura de isolamento e auditoria descrita para os testes *web*, mantendo a uniformidade da suíte.

O método auxiliar `novo()` (linha 22) fornece um ponto de criação reutilizável por outras classes. Na linha 24, o método `post()`, provido por `BasicRest`, envia o *payload* e verifica de forma síncrona se o servidor respondeu com o código esperado (`HttpStatus.SC_CREATED`).

Por fim, o cenário `testFluxoSimples()` (linha 31) é executado independentemente da interface gráfica. Na linha 36, o teste utiliza a classe utilitária para carregar dados armazenados externamente, atendendo ao requisito de *Separação de dados de entrada [RT16]*. Em seguida, submete a requisição diretamente à API na linha 38 e verifica o sucesso da operação por meio do identificador retornado.

3.3 Atendimento aos Requisitos Propostos

A Tabela 1 sintetiza como os componentes e as decisões de projeto da arquitetura atendem aos requisitos técnicos da Seção 2, relacionando-os às evidências observadas na implementação.

Tabela 1: Mapeamento dos requisitos técnicos

Req.	Evidência na Arquitetura
RT01	Classe `SystemFrontEnd` encapsula o Selenium WebDriver.
RT02	Sincronização dinâmica com `defaultWait` e `forceRetry`.
RT03	Identificadores semânticos (e.g., `Botao.ADICIONAR`).
RT04	Busca centralizada por IDs via `getWebElementById()`.
RT05	Estrutura `EspecieForm` vincula dados à interface.
RT06	Métodos reutilizáveis como `create()` e `fillForm()`.
RT07	Classe `EspecieWebTestCase` exclusiva para Espécies.
RT08	Substituição de valores literais por constantes de domínio.
RT09	Enumerador `GrupoEspecie.BOVIDEOS` para dados estáticos.
RT10	Centralização de dependências via `insertDefaultTestData()`.
RT11	Configuração do estado diretamente via serviços *backend*.
RT12	Cenários autocontidos constroem seu próprio contexto.
RT13	Anotação `@Clean` restaura automaticamente o banco de dados.
RT14	Checagem direta de persistência via `checkOnDatabase()`.
RT15	Funcionalidades globais concentradas na classe `TestUtils`.
RT16	Separação de dados em objetos e arquivos externos.
RT17	Instanciação parametrizada em modo *headless* para CI/CD.
RT18	Geração de rastros detalhados por chamadas `LOGGER.trace()`.
RT19	Registro de autoria e metadados via anotação `@ZettaTest`.
RT20	Vínculo com tarefas de gestão através do atributo `cards`.

4 Projeto e Avaliação

O objetivo desta seção é avaliar a aplicação da abordagem de testes proposta a partir de evidências quantitativas e qualitativas obtidas durante um ano de uso contínuo em um projeto real. O estudo foi conduzido no desenvolvimento do sistema *Geosidagro*, reformulação completa do *Sidagro*, antigo sistema de defesa agropecuária de Minas Gerais. Desenvolvido por meio de um convênio entre o Instituto Mineiro de Agropecuária (IMA) e a Universidade Federal de Lavras (UFLA), o projeto modernizou as funcionalidades e a pilha tecnológica da aplicação. O *Geosidagro* será a plataforma oficial do IMA para gestão das políticas de defesa agropecuária do estado, caracterizando-se como um sistema de missão crítica projetado para atender centenas de milhares de usuários. Nesse contexto, a arquitetura proposta foi utilizada continuamente durante 12 meses no desenvolvimento e manutenção da suíte de testes automatizados.

Para organizar a criação desses testes e garantir a qualidade do modelo ao longo do tempo, foi estruturada uma equipe dividida por níveis de experiência. A equipe conta com quatro analistas de teste em nível júnior, um analista em nível pleno (que atuou como líder da equipe) e um especialista em qualidade de software, responsável

por conferir se os padrões previamente estabelecidos estavam sendo seguidos corretamente.

O fluxo de trabalho baseou-se em duas etapas de revisão de código. Inicialmente, os analistas de nível júnior desenvolviam os cenários de teste para as telas do sistema, que eram revisados pelo líder da equipe, responsável por apontar correções na lógica, nos dados e na implementação. Após os ajustes, o código passava pela revisão final do especialista que verificava a conformidade da implementação com a arquitetura proposta antes da integração ao repositório.

Para fundamentar a avaliação, foi desenvolvido um extrator baseado na API REST do GitLab[5], responsável por recuperar o histórico completo de interações dos *Merge Requests* (MRs) do repositório oficial do projeto de testes entre 10/06/2025 e 25/06/2026. Os dados utilizados na avaliação foram consolidados no pacote de replicação do estudo. Os indicadores agregados são apresentados na Tabela 2.

Tabela 2: Métricas agregadas do ciclo de vida

Métrica Analisada	Valor Obtido
Total de *Merge Requests* Integrados	97
Quantidade de casos de Testes Web	115
Quantidade de casos de Testes API	185
Volume Total de Alterações (Linhas de Código)	53.520
Tempo Médio para Integração (*Dias até o Merge*)	4,0 dias
Média de Participantes por *Merge Request*	2,13
Volume Total de Comentários em Revisões	1.613

Os dados evidenciam um volume expressivo de contribuições, com 97 MRs integrados que movimentaram mais de 53 mil linhas de código na implementação de 115 testes Web e 185 de API. A média de 2,13 participantes por *Merge Request* evidencia a revisão colaborativa entre analistas de teste e revisores, indicando que os cenários eram sistematicamente avaliados antes da integração.

Os 1.613 comentários registrados ao longo dos 12 meses refletem o processo contínuo de revisão, correção e aprendizado da equipe. Em conjunto com o tempo médio de integração de 4,0 dias, esses dados evidenciam que os testes automatizados foram submetidos a um processo sistemático de revisão antes da integração, evitando a incorporação de testes mal estruturados ou que comprometem a confiabilidade da validação.

4.1 Rastreabilidade Histórica dos Requisitos

Para avaliar qualitativamente a aplicação dos Requisitos Técnicos (RTs), foi auditado o histórico de *Code Review*, composto por 1.613 comentários. A análise das discussões identificou evidências da adoção das diretrizes propostas, destacando os seguintes casos extraídos do repositório:

Encapsulamento de sincronismo [RT02]: A proibição do uso de pausas estáticas é recorrente no histórico. No *MR 22*, o revisor é enfático: *"Não se deve usar DELAY em nenhuma hipótese"*. Essa diretriz evoluiu para a criação de métodos resilientes, como o `forceRetry`, projetado para forçar novas buscas caso o elemento não esteja imediatamente pronto, combatendo a intermitência (*flakiness*). O mesmo rigor é observado no *MR 15*, onde se exige a substituição de atrasos manuais por lógicas de busca dinâmica.

Priorização de identificadores estáticos [RT04]: A preferência por identificadores estáticos em detrimento de seletores frágeis (*XPaths*) é evidenciada no *MR 86*. O revisor rejeita explicitamente o uso de caminhos estruturais: *"Não tem como ter ID? Não gostei desse método nem do XPath para ele"*. Em resposta, o desenvolvedor informa que a equipe de *front-end* foi acionada para disponibilizar identificadores e que, até lá, a limitação seria registrada como dívida técnica. Essa necessidade motivou a criação da anotação `@TechDebt`, permitindo identificar pontos do teste com dívidas técnicas [15].

Mapeamento entre objeto e tela [RT05]: No *MR 86*, observa-se a criação de classes como `HabilitacaoMormoForm` para representar semanticamente os dados da tela, seguindo o padrão *Page Object*. O desenvolvedor justifica a decisão pela necessidade de maior semântica, destacando a existência de múltiplos formulários na tela e a adoção de classes específicas para cada tipo: *"Considerei a questão semântica. A tela tem três formulários [...] faria sentido manter uma classe de formulário específica para cada tipo"*. O revisor complementa exigindo a documentação da decisão arquitetural via JavaDoc: *"Comentar com JavaDoc no código [...] mesmo que seja um texto mais sucinto, comente"*.

Padronização operacional de telas [RT06]: No *MR 40*, reforça-se a adoção do método `putOnEditMode` com assertivas padronizadas para validar a entrada em modo de edição para todas as entidades do sistema. A governança ativa desse padrão é comprovada pela diretriz do revisor técnico: *"Em todos os putOnEditMode, seria interessante um assert para verificar se entrou na tela de edição. Tipo confirmar se um campo da pesquisa aparece em algum input na tela de edição"*. Essa interação resultou no compromisso da equipe em replicar a lógica em toda a suíte.

Eliminação de valores literais [RT08]: No *MR 84*, registra-se uma ação de refatoração sistêmica para eliminar números mágicos de protocolos de rede: *"Ao invés de 200, vamos colocar import static do HttpStatus e usar SC_OK... consegue alterar em TUDO?"*.

Abstração de dados estáticos por enumeradores [RT09]: No *MR 94*, são registradas cobranças para eliminação de valores numéricos literais (*magic numbers*)[6]. O histórico de *commits* documenta a substituição dessas constantes por estruturas tipadas e semânticas, como em *"Criando Enum para Grupo Espécie"*, reforçando a adoção de abstrações para representação de dados estáticos.

Centralização de dados dependentes [RT10]: Nas discussões do *MR 88*, a equipe debate a padronização das pré-condições dos cenários. A interação *"Beleza, vamos começar a adotar esse método. Já fazemos igual nos testes web e lá chamamos de create"* comprova a preocupação em consolidar a geração de dependências em *fixtures* reaproveitáveis, evitando a duplicação de chamadas de configuração da API ao longo da suíte.

Injeção de estado via *backend* [RT11]: Consolidada entre os *MR 88* e *92*, a equipe padronizou a configuração de pré-condições via API por meio do método `insertDefaultTestData`. A iniciativa

[5]https://about.gitlab.com/

[6]Valores constantes inseridos diretamente no código, sem um identificador semântico, o que dificulta a compreensão, a manutenção e a rastreabilidade de seu significado [5].

surgiu no *MR 88*, a partir da sugestão do revisor: *"São sempre os mesmos em todos os testes? Se for, podemos padronizar a existência de um método com um nome padrão, por exemplo, setupTestData ou algo assim"*.

Atomicidade e independência [RT12]: Identificado no *MR 71*, o requisito estabelece a criação isolada das dependências de cada teste. Para evitar duplicação sem violar esse princípio, a discussão propõe o uso de dados padronizados e métodos utilitários para criação de entidades, mantendo no teste apenas os relacionamentos específicos do cenário. A diretriz é reforçada pelo revisor técnico: *"Você deve criar todas as dependências isoladamente (em cada método de teste de unidade) e aqui só criar o [objeto] de fato"*. O tema evolui para a adoção de métodos auxiliares, como no trecho: *"Criar um método que recebe um prefixo e cria tudo que um evento pecuário precisa, mas passando o nome do teste e isso constar no clean"*.

Isolamento de estado global [RT13]: As interações técnicas nos *MRs 93* e *94* revelam debates rigorosos acerca da limpeza de persistência após a execução dos roteiros. Cobranças como *"Falta clean desse método e limpeza do clean dos demais"* motivaram a reengenharia da rotina de limpeza, evidenciada pelo *commit "Adaptando Clean para funcionar recursivamente"*, garantindo a independência entre os cenários.

Validação na camada de dados [RT14]: A exigência de validação multicamada é evidenciada no *MR 90*. O revisor técnico intercepta a submissão de um teste que se limitava à interface visual, cobrando a inspeção direta: *"Faltou fazer a consulta no BD para ver se alterou mesmo né?!"*. Essa prática força a integração de *scripts* SQL de verificação para garantir a integridade real dos dados processados.

Componentes utilitários compartilhados [RT15]: No processo de revisão do *MR 95*, o revisor técnico intercepta a tentativa de criação de lógicas isoladas para a manipulação de datas, emitindo a seguinte diretriz explícita: *"Negativo. Vamos criar um método no TestUtils e sempre usá-lo, ok?"*. A subsequente centralização da rotina atesta o cumprimento do princípio de reutilização para evitar a inflação do código.

Observabilidade e governança de *logs* [RT18]: O histórico revela uma preocupação constante com a rastreabilidade de falhas. No *MR 90*, sugere-se o uso de `LOGGER` para reportar estados inválidos. Já no *MR 21*, o revisor exige o detalhamento das etapas de decisão lógica nos *logs*: *"Cria um LOGGER.trace só dizendo [...]"*.

Identificação e autoria dos cenários [RT19]: Os *MRs 83* e *50* evidenciam a padronização das informações de identificação para facilitar a rastreabilidade e a comunicação. No *MR 83*, definiu-se que a data de criação deveria corresponder à primeira liberação do teste, consolidada posteriormente no ajuste do campo *createdAt*. Já no *MR 50*, estabeleceu-se a padronização da identificação dos analistas: *"Vamos padronizar os nomes da equipe para serem mais simbólicos. Nome e sobrenome"*.

Vínculo com ferramentas de gestão [RT20]: O alinhamento entre os *cards* do sistema de gestão e as anotações do código é evidenciado nos *MRs 64* e *86*, garantindo a rastreabilidade da automação. Esse vínculo também é observado em diferentes pontos do histórico de *commits*, como no *MR 15* (*"Mapeando métodos de testes com cards"*) e no *MR 61* (*"Adicionando cards referentes à Sprint 7"*), além de ser confirmado no *MR 98* pela observação do desenvolvedor: *"Criamos uma task para ajuste e mapeamos no método"*.

O volume de interações evidencia que a maturidade da abordagem resultou de um processo contínuo de evolução ao longo de um ano, no qual o *Code Review* atuou como principal mecanismo de governança arquitetural do projeto de testes, preservando a aderência às diretrizes propostas. A ausência dos requisitos *Abstração do motor de execução [RT01]*, *Mapeamento semântico de identificadores [RT03]*, *Coesão funcional por interface [RT07]*, *Separação dos dados de entrada [RT16]* e *Execução não interativa [RT17]* nas discussões pode indicar que esses requisitos já estavam incorporados à infraestrutura e, por isso, demandaram menos intervenções explícitas durante o período avaliado.

4.2 Impacto e Benefícios ao Projeto

A adoção da abordagem proposta durante um ano permitiu observar os seguintes resultados no projeto Geosidagro:

Indícios de estabilidade da suíte: Dos 97 *Merge Requests* integrados, mais de 80% correspondem exclusivamente à implementação de novos cenários de teste. Em contrapartida, apenas 3 MRs (*MR 33*, *MR 78* e *MR 87*) demandaram correções em testes existentes. Esses resultados sugerem baixa demanda por manutenção corretiva e permitiram que a equipe concentrasse esforços na expansão da cobertura da suíte.

Indícios de contenção do custo de manutenção: A análise das contribuições aponta medianas de 9 arquivos modificados e 338 mudanças por *Merge Request*, com 307 linhas adicionadas e 16 removidas. Optou-se pela mediana por ela ser menos sensível a contribuições excepcionalmente grandes, representando melhor o comportamento típico das entregas. Esses valores indicam que a evolução da suíte ocorreu de forma predominantemente incremental e localizada. Como mais de 80% das entregas consistiram na criação de novos cenários, a necessidade de cerca de 300 linhas para implementar um fluxo completo — incluindo dados de entrada, validação de resultados, *scripts* SQL, mapeamento de interface e casos de teste — sugere elevado reaproveitamento do *framework* e baixo impacto estrutural sobre a suíte existente.

Indícios de produtividade e velocidade de entrega: A padronização da arquitetura e a reutilização de componentes favoreceram a produtividade da equipe. O fluxo de desenvolvimento manteve alta cadência, com média de 9,5 *commits* por entrega e mais de 1.140 tópicos de discussão técnica resolvidos. Enquanto os primeiros *Merge Requests* (MR 01 a 05) continham menos de 200 mudanças, a maturidade da arquitetura permitiu entregar alterações substancialmente maiores no mesmo ritmo, como os *MR 93* (2.181 mudanças) e *MR 94* (2.115 mudanças). Da mesma forma, módulos complexos, como o *MR 98*, foram desenvolvidos em apenas um dia, fornecendo indícios de que a arquitetura, em conjunto com o processo de revisão adotado, apoiou a evolução da suíte.

Como limitação, os resultados não permitem isolar completamente os efeitos da arquitetura daqueles decorrentes do processo rigoroso de *Code Review*, que também contribuiu para a aderência aos requisitos e a baixa incidência de correções. Além disso, não foram realizadas comparações controladas com arquiteturas alternativas.

5 Conclusão

Este trabalho apresentou uma abordagem unificada para o desenvolvimento de testes funcionais automatizados em arquiteturas web modernas. A proposta integra testes de API e de interface Web em uma arquitetura modular que encapsula as dependências da ferramenta de automação e promove o desacoplamento entre a lógica dos cenários de teste e os mecanismos responsáveis por sua execução. Com isso, busca direcionar o esforço do analista principalmente à modelagem dos cenários de negócio, promovendo padronização, reúso e facilidade de manutenção da suíte.

Como contribuições, o trabalho sistematizou vinte requisitos técnicos para a construção de suítes de testes automatizados sustentáveis, propôs uma arquitetura que os materializa e avaliou longitudinalmente sua aplicação em um projeto real de grande porte. Os requisitos consolidam boas práticas observadas na literatura e estabelecem diretrizes voltadas à redução da fragilidade dos testes, ao reúso de componentes e à evolução da suíte ao longo do ciclo de vida do software.

A abordagem foi avaliada durante doze meses de uso contínuo, abrangendo 97 *Merge Requests* e 300 casos de teste automatizados. A análise dos comentários de revisão mostrou ainda que os requisitos técnicos orientaram decisões, correções e refatorações ao longo da evolução da suíte. Mais de 80% dos *Merge Requests* adicionaram exclusivamente novos cenários, enquanto apenas três tiveram como objetivo principal corrigir testes previamente integrados. Em conjunto, esses resultados sugerem que a abordagem favoreceu a expansão da suíte com estabilidade e baixa demanda por manutenção corretiva.

Como trabalhos futuros, pretende-se ampliar a avaliação quantitativa dos benefícios da abordagem, investigando seu impacto sobre a produtividade, o esforço de manutenção e a estabilidade da suíte. Também se pretende compará-la com outras arquiteturas ou linhas de base em projetos de diferentes domínios, tecnologias e equipes.

Disponibilidade de Artefatos

O pacote de replicação, com os dados dos *Merge Requests*, guia de implementação e artefatos da pesquisa, está publicamente disponível em: https://doi.org/10.5281/zenodo.21865807.

Agradecimentos

Os autores agradecem à Universidade Federal de Lavras (UFLA), ao Instituto Mineiro de Agropecuária (IMA) e à Fundação de Desenvolvimento Científico e Cultural (FUNDECC) – Convênio nº 24/2024 pelo apoio para realização deste trabalho.

Referências

[1] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing.* Cambridge University Press, Cambridge, UK.

[2] Mike Cohn. 2009. *Succeeding with Agile: Software Development Using Scrum.* Addison-Wesley Professional, Upper Saddle River, NJ.

[3] João Victor de Morais, Neumar Costa Malheiros, and Ricardo Terra. 2025. Automação de Testes de APIs REST no SisNIR. In *X Simpósio Brasileiro de Testes de Software Sistemático e Automatizado (SAST).* 138–140.

[4] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *27th International Symposium on the Foundations of Software Engineering (ESEC/FSE).* 830–840.

[5] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, Boston, MA.

[6] Boni García, Jose M. del Alamo, Maurizio Leotta, and Filippo Ricca. 2024. Exploring Browser Automation: A Comparative Study of Selenium, Cypress, Puppeteer, and Playwright. In *21st International Conference on the Quality of Information and Communications Technology (QUATIC).* 142–149.

[7] Boni García, Filippo Ricca, Maurizio Leotta, and Mario Munoz-Organero. 2026. Test Automation with Selenium: A Survey. *Information and Software Technology* 194 (2026), 108077.

[8] Vahid Garousi and Mika V. Mäntylä. 2016. When and What to Automate in Software Testing? A Multi-Vocal Literature Review. *Information and Software Technology* 76 (2016), 92–117.

[9] Satish Gojare, Rahul Joshi, and Dhanashree Gaigaware. 2015. Analysis and Design of Selenium WebDriver Automation Testing Framework. *Procedia Computer Science* 50 (2015), 341–346.

[10] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. 2013. Automated Detection of Test Fixture Strategies and Smells. In *6th International Conference on Software Testing, Verification and Validation (ICST).* 322–331.

[11] IEEE. 1998. IEEE Std 829-1998 – IEEE Standard for Software Test Documentation. IEEE Standard.

[12] A. Ilie et al. 2018. Test automation pyramid from theory to practice. In *22nd International Conference on Automation, Quality and Testing, Robotics (AQTR).* 1–5.

[13] ISO. 2013. *ISO/IEC/IEEE 29119-2:2013 Software and systems engineering — Software testing — Part 2: Test processes.* Technical Report. International Organization for Standardization.

[14] ISO. 2021. *ISO/IEC/IEEE 29119-3:2021 Software and systems engineering — Software testing — Part 3: Test documentation.* Technical Report. International Organization for Standardization.

[15] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software* 29, 6 (2012), 18–21.

[16] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. 2013. Comparing the Maintainability of Selenium WebDriver Test Suites Employing Different Locators: A Case Study. In *1st International Workshop on Joining Academia and Industry Contributions to Testing Automation (ISSTA).* 53–58.

[17] Douglas H. Longo, Patricio F. Padilha, Ana B. Zaina, and Rosane G. M. Caetano. 2015. Investigating Automated Test Patterns in Erratic Tests by Considering Complex Objects. *International Journal of Information Technology and Computer Science* 7, 3 (2015), 54–59.

[18] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *22nd International Symposium on Foundations of Software Engineering (FSE).* 643–653.

[19] Steve McConnell. 2004. *Code Complete: A Practical Handbook of Software Construction.* Microsoft Press, Redmond, WA.

[20] Gerard Meszaros. 2007. *xUnit Test Patterns: Refactoring Test Code.* Addison-Wesley Professional, Boston, MA.

[21] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing.* Wiley, Hoboken, NJ.

[22] Sam Newman. 2015. *Building Microservices: Designing Fine-Grained Systems.* O'Reilly Media, Sebastopol, CA.

[23] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2021. A Survey of Flaky Tests. *Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–74.

[24] Lalji Prasad, Rashmi Yadav, and Niti Vore. 2021. A Systematic Literature Review of Automated Software Testing Tool. In *3rd International Conference on Computing Informatics and Networks (ICCIN).* 101–123.

[25] Roger S. Pressman and Bruce R. Maxim. 2016. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, New York, NY.

[26] Taryn Takebayashi, Anthony Peruma, Mohamed Wiem Mkaouer, and Christian D. Newman. 2023. An Exploratory Study on the Usage and Readability of Messages Within Assertion Methods of Test Cases. In *2nd International Workshop on Natural Language-Based Software Engineering (NLBSE).* 32–39.

[27] Hafiz Muhammad Umar. 2025. *A Comparative Analysis of API Testing Approaches in Continuous Integration/Continuous Deployment (CI/CD) Pipelines in Cloud Based Systems.* Technical Report. Lappeenranta–Lahti University of Technology.

[28] Marco Tulio Valente. 2020. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade.* Independente, Belo Horizonte, MG.

[29] George Vegelien, Carolin Brandt, Bas Graaf, and Arie van Deursen. 2026. Addressing Test Flakiness: Practical Approaches in a Database-Reliant Industrial System. In *48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).*

[30] Ham Vocke and Martin Fowler. 2018. The Practical Test Pyramid. Disponível em: <https://martinfowler.com/articles/practical-test-pyramid.html>.

[31] Vahid Garousi Yusifoğlu, Yasaman Amannejad, and Aysu Betin Can. 2015. Software Test-Code Engineering: A Systematic Mapping. *Information and Software Technology* 58 (2015), 123–147.

[32] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanc Muslu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *23rd International Symposium on Software Testing and Analysis (ISSTA).* 385–396.