GABRIEL HENRIQUE SILVA AMORIM

DESENVOLVIMENTO ÁGIL DE SOFTWARE NA DTI DIGITAL

LAVRAS – MG

2023

GABRIEL HENRIQUE SILVA AMORIM

DESENVOLVIMENTO ÁGIL DE SOFTWARE NA DTI DIGITAL

Relatório de estágio supervisionado apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

Prof. Dr. Ricardo Terra Nunes Bueno Villela
Orientador

LAVRAS – MG
2023

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos

da Biblioteca Universitária da UFLA

Amorim, Gabriel Henrique Silva.
Desenvolvimento Ágil de Software na DTI Digital / Gabriel Henrique Silva Amorim. – Lavras : UFLA, 2023.
61 p. : il.

Relatório de estágio (Graduação)–Universidade Federal de Lavras, 2023.
Orientador: Prof. Dr. Ricardo Terra Nunes Bueno Villela.
Bibliografia.

1. Aplicações web. 2. Desenvolvimento ágil. 3. Estágio. I. Universidade Federal de Lavras. II. Título.

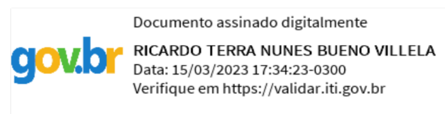
GABRIEL HENRIQUE SILVA AMORIM

DESENVOLVIMENTO ÁGIL DE SOFTWARE NA DTI DIGITAL

Relatório de estágio supervisionado apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

APROVADA em 2 de março de 2023.

Dr. Maurício Ronny de Almeida Souza UFLA
Bel. José Renato Zacaroni Barbosa UFLA



Prof. Dr. Ricardo Terra Nunes Bueno Villela
Orientador

LAVRAS – MG
2023

A Deus pelas constantes graças concedidas e que por amor me sustenta.

À Virgem Maria Santíssima.

À minha família que em todas as ocasiões me ama e apoia.

Dedico

AGRADECIMENTOS

Agradeço a Deus as iluminações durante a realização deste trabalho.

À minha família e aos meus amigos o suporte, os incentivos e as orações.

Ao professor Ricardo Terra a orientação, as considerações, a confiança e a paciência, me proporcionando muito aprendizado e evolução.

Aos membros da banca, professor Maurício e José, as contribuições realizadas a este trabalho.

À DTI Digital e aos colegas de equipe o convívio, os ensinamentos e as oportunidades.

À Universidade Federal de Lavras os recursos fornecidos durante o percurso de formação.

RESUMO

O desenvolvimento ágil de software vem sendo bastante prestigiado pelas organizações, as quais constantemente buscam obter o maior proveito possível do conjunto de práticas e ferramentas ágeis em seu contexto de desenvolvimento de software. Tendo em vista as inúmeras especificidades do aspecto prático referentes à adoção de métodos ágeis em cada fluxo de desenvolvimento, torna-se importante contribuir evidenciando uma parcela da realidade vivida durante a execução de um determinado fluxo. Diante disso, este relatório descreve as principais atividades realizadas durante o período de estágio supervisionado realizado na DTI Digital, empresa que atua na transformação digital de organizações por meio do desenvolvimento ágil, fornecendo geração de valor contínua para os clientes. Este relatório também relata as etapas do fluxo de desenvolvimento ágil experienciado durante as atividades, o qual transcorre com a realização de ritos do *framework* Scrum, a codificação da solução, a aplicação de práticas de integração e entrega contínua e, por fim, o contínuo monitoramento do software. A apresentação das etapas elucida os conceitos utilizados e descreve a realidade prática de cada um deles considerando o contexto da DTI. Durante o estágio, as principais atividades foram a criação de interfaces de usuário para uma nova aplicação web e o desenvolvimento de novas funcionalidades para microsserviços. Para isso, foram utilizadas ferramentas como a biblioteca React da linguagem JavaScript e o *framework* ASP.NET Core da linguagem C#. Tais atividades possibilitaram ao estagiário consolidar conceitos da área de Engenharia de Software ao aplicá-los com o intuito de modernizar e melhorar a manutenibilidade do software.

Palavras-chave: Aplicações web. Desenvolvimento ágil. Estágio. Microsserviços.

LISTA DE FIGURAS

Figura 2.1 – Representação do fluxo de desenvolvimento ágil 13
Figura 2.2 – Gráfico *Burndown* 17
Figura 2.3 – Representação do SCV Centralizado 22
Figura 2.4 – Representação do SCV Descentralizado 23
Figura 2.5 – Exemplo de um fluxo no modelo *Git Flow* 25
Figura 2.6 – Relação entre Integração Contínua (CI) e Entrega Contínua (CD) 30
Figura 2.7 – Representação do exemplo de estágios do *pipeline* 31
Figura 3.1 – Representação da relação dos fundamentos do React 37
Figura 3.2 – Exemplo semelhante ao fluxo da aplicação web desenvolvida 37
Figura 3.3 – Interface de conteúdo expirado da aplicação web 38
Figura 3.4 – Interface de validação de usuário da aplicação web ao inserir código inválido 39
Figura 3.5 – Representação da arquitetura de microsserviços 41
Figura 3.6 – Primeira etapa (Scrum) do desenvolvimento ágil 43
Figura 3.7 – Terceira etapa (Integração Contínua) do desenvolvimento ágil 48
Figura 3.8 – Quarta etapa (Entrega Contínua) do desenvolvimento ágil 49
Figura 3.9 – Representação do padrão arquitetural *Onion Architecture* 50
Figura 3.10 – Representação da arquitetura do microsserviço 52
Figura 3.11 – Teste de listagem de produtos por meio da interface fornecida pelo Swagger 55

LISTA DE TABELAS

Tabela 3.1 – Descrição do item referente à funcionalidade de validação. 43
Tabela 3.2 – Descrição da funcionalidade de operações de CRUD de produto. 51

LISTA DE CÓDIGOS

Código 3.1 Exemplo de utilização de um componente do MUI. 39
Código 3.2 Primeiro teste de unidade da funcionalidade de validação. 44
Código 3.3 Implementação inicial da funcionalidade de validação. 45
Código 3.4 Segundo teste de unidade da funcionalidade de validação. 46
Código 3.5 Implementação final da funcionalidade de validação. 47
Código 3.6 Implementação do cadastro de produto no componente de serviço. 53
Código 3.7 Implementação da listagem de produtos no componente de serviço. 54

SUMÁRIO

1 INTRODUÇÃO 10
2 DESENVOLVIMENTO ÁGIL NO CONTEXTO DA DTI 12
2.1 Scrum 15
2.1.1 Fundamentação Teórica 15
2.1.2 Contexto DTI 18
2.2 Codificação 21
2.2.1 Fundamentação Teórica 21
2.2.2 Contexto DTI 25
2.3 Integração Contínua 27
2.3.1 Fundamentação Teórica 27
2.3.2 Contexto DTI 28
2.4 Entrega Contínua 30
2.4.1 Fundamentação Teórica 30
2.4.2 Contexto DTI 32
2.5 Monitoramento 33
2.5.1 Fundamentação Teórica 33
2.5.2 Contexto DTI 34
2.6 Considerações finais 35
3 ATIVIDADES REALIZADAS 36
3.1 Criação de *Single Page Application* 36
3.1.1 Interface de conteúdo expirado 37
3.1.2 Interface de validação de usuário 38
3.1.3 Considerações finais 40
3.2 Criação de funcionalidades para microsserviços 41
3.2.1 Funcionalidade de validação 42
3.2.2 Funcionalidade de operações de CRUD 49
3.2.3 Considerações finais 56
4 CONCLUSÃO 57
REFERÊNCIAS 59

1 INTRODUÇÃO

O desenvolvimento ágil de software emergiu com o principal propósito de aumentar o potencial de adaptação do processo de desenvolvimento a constantes mudanças, oriundas de manutenções ou de novas requisições de usuários (AL-SAQQA; SAWALHA; ABDELNABI, 2020). Trata-se de um termo amplo que envolve práticas e ferramentas coerentes com os princípios presentes no Manifesto para Desenvolvimento Ágil de Software[1] (ALLIANCE, 2023). Outro destaque, ainda segundo Alliance (2023), é que as ferramentas ágeis procuram proporcionar às equipes um suporte para a construção de metodologias peculiares ao contexto vivido, demonstrando o quão numerosa são as maneiras de se usufruir de tais práticas.

De acordo com *State of Agile Report* da Digital.ai (2021), pesquisa realizada com cerca de mil participantes de diferentes países, a adoção de valores ágeis pelas empresas obteve um aumento de 37% em 2020 para 86% em 2021. A pesquisa também destaca que o aumento da capacidade de gerenciamento de mudanças e das entregas de software são os motivos mais importantes para a adoção do ágil pelas organizações, demonstrando que a versatilidade proporcionada por essas práticas podem de fato auxiliar bastante no processo de desenvolvimento, principalmente em situações imprevisíveis.

Este documento descreve as principais atividades realizadas durante o estágio supervisionado exercido na empresa DTI Digital de 21 de fevereiro de 2022 a 1 de fevereiro de 2023. A DTI Digital é uma empresa privada fundada em 2009, com sede localizada na cidade de Belo Horizonte, no estado de Minas Gerais. A empresa possui cerca de mil colaboradores e oferece soluções tecnológicas considerando a necessidade de negócio de cada cliente. Para isso, sustenta uma forte cultura ágil, trabalhando com equipes concisas e multidisciplinares determinadas a atuar na transformação digital de empresas que almejam se tornar ágeis.

As atividades consistiram em desenvolver interfaces para aplicações web e funcionalidades para aplicações de microsserviço, visando contribuir com a entrega contínua de modernas soluções para o cliente da empresa. O desenvolvimento das interfaces forneceu ao estagiário a oportunidade de adquirir conhecimento sobre aplicações web modernas no formato de *Single Page Application* (SPA), enquanto o desenvolvimento para microsserviços contribuiu para o aprendizado sobre padrões arquiteturais e utilização de práticas como *Test Driven Development* (TDD). Conjuntamente, a atuação nesse contexto possibilitou ao estagiário experienciar um fluxo de desenvolvimento ágil de software utilizando de práticas ágeis sob a cultura da DTI.

[1] Disponível em <https://agilemanifesto.org/iso/ptbr/manifesto.html>.

O fluxo de desenvolvimento ágil é relatado no relatório observando seus aspectos teóricos e práticos com base na visão obtida durante as atividades. Desse modo, o fluxo é destacado em cinco etapas, sendo a primeira etapa caracterizada pela realização dos ritos do *framework* Scrum. As equipes nas quais o estagiário atuou eram organizadas conforme as definições do *framework*, compostas pelo *Product Owner*, o *Scrum Master* e cerca de cinco desenvolvedores.

A partir do Scrum, o fluxo transcorre com a segunda etapa, denominada Codificação, na qual ocorre o desenvolvimento de soluções atrelado a um fluxo de trabalho com a ferramenta Git. Em seguida, na terceira etapa, são descritas as práticas de Integração Contínua, pelas quais se atinge maior controle e rapidez na integração do código desenvolvido. Na quarta etapa, é apresentada a Entrega Contínua, que permite melhor controle de implantações e entregas de novas versões do software. Na quinta e última etapa, é apresentado o Monitoramento, que lida com o acompanhamento do software e o seu estado de operação para os usuários finais.

O relatório está estruturado como a seguir. Na Seção 2, são dispostos os conceitos, as ferramentas, as práticas e as técnicas referentes ao desenvolvimento ágil, fundamentando-os teoricamente e descrevendo de que modo são considerados no contexto atuado na DTI. Na Seção 3, é descrito o modo como foram realizadas as atividades de estágio, relatando o fluxo e as situações específicas de desenvolvimento durante cada atividade, além de destacar quais conhecimentos e ferramentas foram necessários para realizá-las. Por fim, na Seção 4, são dispostas as considerações finais do relatório, discutindo sobre os conhecimentos adquiridos durante a realização das atividades.

2 DESENVOLVIMENTO ÁGIL NO CONTEXTO DA DTI

Nesta seção, são descritos os principais conceitos aplicados durante o desenvolvimento ágil no contexto da DTI, considerando também a cultura ágil e processos de desenvolvimento adotados pelo cliente. Os conceitos são dispostos em blocos sequenciais, onde cada bloco representa uma etapa do desenvolvimento ágil utilizada durante as atividades do estágio.

A seção está dividida considerando uma subseção para cada etapa do fluxo representado na Figura 2.1. O processo de desenvolvimento ágil se dá como descrito a seguir:

1. Utilização do *framework* Scrum (Bloco 1), por meio do qual uma equipe independente gerencia todo o processo de implementação e entrega dos novos incrementos de um produto. Nessa etapa, define-se o *Product Backlog*, um catálogo onde constam todos os itens que se planeja desenvolver para compor o produto final. Durante a definição do *Product Backlog*, pode ocorrer de um item ser revisado diversas vezes para descrevê-lo ou segmentá-lo de um modo mais preciso, processo conhecido como *Refinamento*. Além disso, para um melhor entendimento por parte de todos os grupos e organizações envolvidos no projeto, aplica-se a técnica de *Behavior Driven Development* (BDD), que define uma estrutura padrão de descrição na qual cada item do *Product Backlog* deve se adequar (NORTH et al., 2006). Definido o *Product Backlog*, define-se o *Sprint*, período no qual será realizado o desenvolvimento de um conjunto de itens do *Product Backlog*. O *Sprint* se inicia com uma reunião de planejamento, a *Sprint Planning*, que tem como resultado o *Sprint Backlog*, novo catálogo onde constam os itens selecionados e definidos como de maior prioridade para serem desenvolvidos e entregues ao final do *Sprint* vigente. Com o *Sprint Backlog* estabelecido, procede-se com uma sequência de reuniões diárias, chamadas de *Daily Scrum*, que permitem à equipe um alinhamento constante sobre o andamento do desenvolvimento dos itens do *Sprint Backlog*. Por fim, realiza-se a *Sprint Review*, onde são apresentados e discutidos os resultados alcançados com todos os envolvidos no projeto, e a *Sprint Retrospective*, onde a equipe levanta os acertos, as dificuldades e as possíveis melhorias para o próximo *Sprint*.

2. Etapa de Codificação (Bloco 2), onde se realiza o desenvolvimento dos itens do *Sprint Backlog* que exigem uma solução de codificação de software. Durante essa etapa, pode ocorrer de os responsáveis pelo desenvolvimento atuarem em itens de alto grau de dependência entre si, exigindo um esforço excessivo durante a integração dos códigos

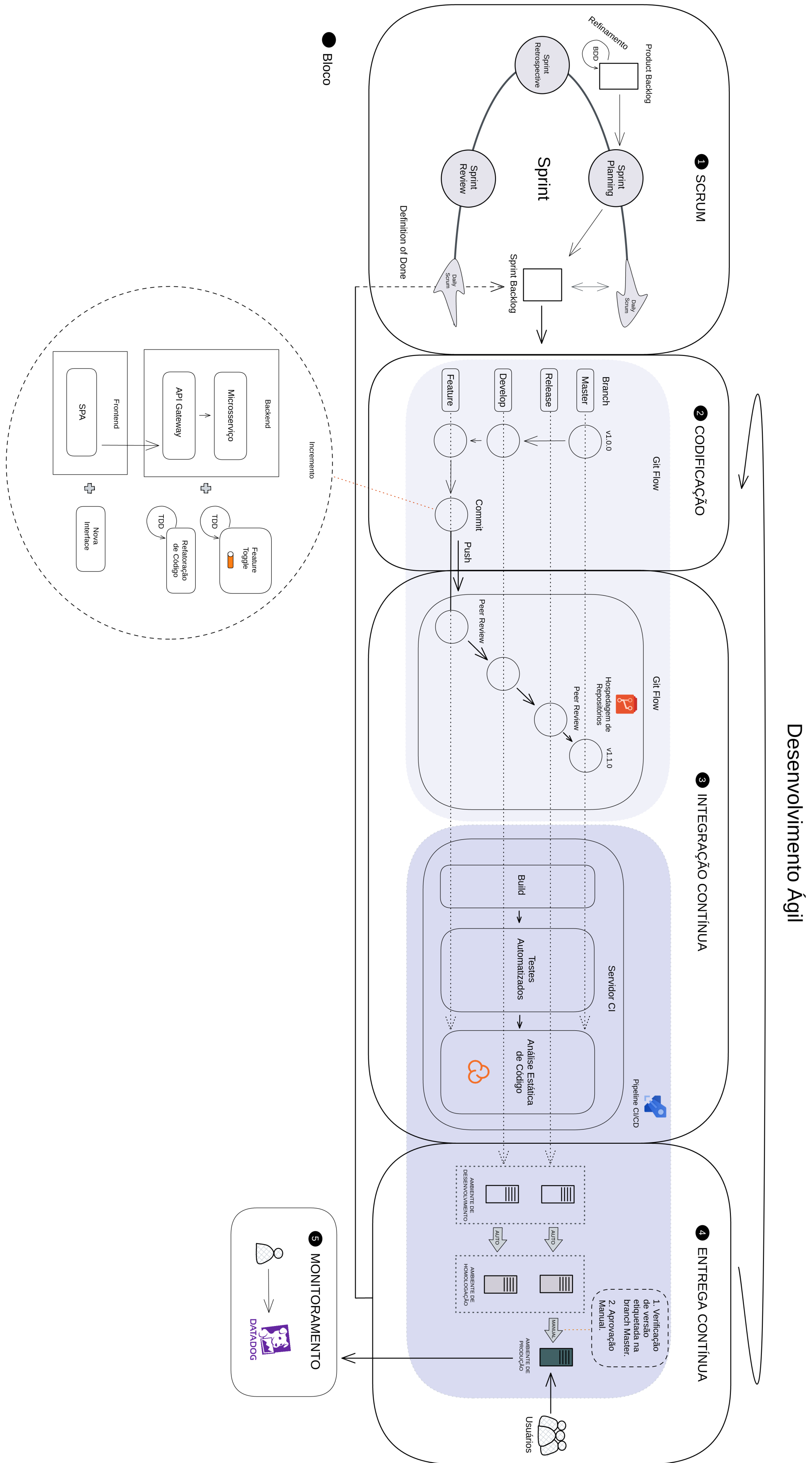


Figura 2.1 – Representação do fluxo de desenvolvimento ágil.
Fonte: Do autor (2022).

relacionados a eles. Diante disso, para se ter um maior controle e segurança durante a integração entre os diversos itens que estão sendo desenvolvidos, opta-se por seguir um fluxo de controle de versões especificado pelo modelo *Git Flow*[1]. Nessa etapa de codificação, utiliza-se do fluxo inicial do *Git Flow*, que ocorre a partir de um repositório composto por duas réplicas da última versão do código completo do sistema. Para a primeira réplica se dá o nome *master*, e para a segunda o nome *develop*, que possui a *master* como referência. Como próximo passo, cria-se uma nova réplica chamada *feature*, que faz referência à *develop*, para cada nova funcionalidade que será desenvolvida. Na réplica *feature* é realizada toda a codificação que resultará em um novo incremento ao produto. A conclusão da codificação acontece quando o desenvolvedor estabelece um último marco (*commit*), definindo assim o fim do incremento. Por fim, envia-se integralmente o código incrementado ao servidor central para posterior integração com os códigos dos demais itens desenvolvidos.

3. Etapa de Integração Contínua (Bloco 3), período no qual se realiza o fluxo de integração e de testes dos códigos desenvolvidos. Nessa etapa, executam-se os procedimentos finais do *Git Flow*, dessa vez efetuados sobre o código incrementado enviado ao servidor central remoto, onde consta o repositório fonte. De início, a equipe realiza uma revisão para discutir sobre a integração das alterações do código da *feature* com a *develop*, onde se encontra o código dos demais itens desenvolvidos. Após aprovada a integração, cria-se uma nova réplica chamada *release*, originada da *develop* atualizada. Essa réplica define que uma nova versão do produto pode ser lançada. O fluxo do *Git Flow* se encerra após a ocorrência de uma nova revisão e integração do código da *release* e *master*, momento no qual se incrementa a versão do produto, definindo que ela deve ser colocada em operação. Além da união dos códigos, durante a etapa de Integração Contínua é realizada uma série de ações automáticas a cada alteração efetuada no repositório no servidor, seja do envio ou da integração de código. Essas ações permitem a verificação e validação do funcionamento adequado do código integrado. A primeira ação realizada é o *build*, onde os pacotes do código são compilados e construídos, em seguida são executados os testes automatizados e, por fim, realizada uma análise estática de código com uma ferramenta específica, para certificar que sua qualidade se mantém.

[1] O *Git Flow* é um conjunto de procedimentos seguido pela equipe para atingir um melhor controle das ramificações de desenvolvimento, normalmente mantidas em um sistema de controle de versões (DRIESSEN, 2010).

4. Etapa de Entrega Contínua (Bloco 4), onde são realizados os procedimentos de implantação da nova versão do produto para os usuários utilizarem. A primeira implantação é realizada em servidores que compõem o ambiente de desenvolvimento, caracterizado por ser um ambiente de menor custo e proporcionar rápidos testes. Em seguida, automaticamente uma nova implantação é realizada, dessa vez no ambiente de homologação, onde a equipe realiza testes manuais referentes ao último desenvolvimento. Esse ambiente se caracteriza por sua grande semelhança ao ambiente de produção, contribuindo para a realização de testes confiáveis. Concluída a realização dos testes, a aplicação é implantada em ambiente de produção para que a sua nova versão seja disponibilizada para os usuários finais. As implantações são realizadas com base nas *branches develop*, da qual o código pode ser implantado em ambiente de desenvolvimento e homologação, e *release*, em que é permitida a implantação em ambiente de produção.

5. Etapa de Monitoramento (Bloco 5), onde o sistema é monitorado pela equipe de desenvolvimento por meio de ferramentas que auxiliam na identificação de comportamentos anormais, além de também encaminharem notificações para a ferramenta de comunicação da equipe, permitindo um constante acompanhamento dos sistemas após a implantação de sua nova versão em ambiente de produção.

2.1 Scrum

2.1.1 Fundamentação Teórica

Nesta seção, apresenta-se o Scrum considerando como principal referência as definições de Schwaber e Sutherland (2020), fonte da qual se originou o Scrum. Diante disso, as citações desta seção são majoritariamente desses autores.

O Scrum tem sido um dos *frameworks* mais utilizados pelas empresas no que diz respeito à experiência com o desenvolvimento ágil de software. Segundo o *State of Agile Report* da Digital.ai (2021), a maioria das equipes ágeis reportou o emprego do Scrum ou alguma de suas versões, como Scrumban[2] ou um modelo híbrido.

A alta adesão das empresas ao Scrum pode ser justificada pela verificação prática de seu propósito que, conforme sua definição, é de fornecer suporte para se "gerar valor através de soluções adaptativas para problemas complexos" (SCHWABER; SUTHERLAND, 2020). Para

[2] Disponível em <https://www.agilealliance.org/scrumban/>.

fornecer esse suporte, segundo Schwaber e Sutherland (2020), o Scrum se concentra em destacar práticas essenciais que devem ser aplicadas durante os eventos definidos pelo *framework*. Tudo isso, se regido por uma pequena equipe de responsabilidades específicas, potencializa o desenvolvimento de soluções.

Porém, antes de ressaltar tais assuntos, vale mencionar quais são os artefatos do Scrum, pois estão relacionados tanto com os eventos e práticas quanto com as responsabilidades da equipe. De acordo com Schwaber e Sutherland (2020), os artefatos são:

- *Product Backlog*: é o artefato principal onde consta, ordenadamente conforme as necessidades de negócio, todos os itens que devem ser desenvolvidos para compor o produto final.

- *Sprint Backlog*: é um subconjunto extraído do *Product Backlog*, contendo uma lista com os itens de maior prioridade para o desenvolvimento. O *Sprint Backlog* deve sempre refletir o estágio atual em que se encontra o desenvolvimento. Antes da seleção do subconjunto, porém, os itens são submetidos ao processo de *Refinamento*. O *Refinamento* tem o objetivo de analisar um item para torná-lo mais preciso, seja por decomposição em pequenos novos itens, seja por acréscimo de detalhes. Além disso, visando maior segurança no que diz respeito à seleção adequada dos itens, a equipe também pode utilizar-se de métricas que refletem o seu desempenho e a sua capacidade de desenvolvimento passados, para limitar o número de itens do *Sprint Backlog* e assim evitar o possível descumprimento de entrega.

- *Incremento*: é a entrega de um item que satisfaz a *Definition of Done* (DoD), nome dado à ação de cumprir todas as condições de verificação e de medidas de qualidade que a equipe estipulou para o produto.

Tendo em vista o objetivo e a importância de cada artefato do Scrum, cabe elencar quais são os eventos do *framework*. “Cada evento no Scrum é uma oportunidade formal para inspecionar e adaptar os artefatos do Scrum”, destacam Schwaber e Sutherland (2020). O principal evento, abrangedor dos demais, é o *Sprint*.

Os *Sprints* são períodos imediatamente sequenciais e de duração fixa de, no máximo, um mês. Durante esse período, além de se executar os demais eventos, a equipe tem por objetivo desenvolver todos os itens do *Sprint Backlog*, além de se atentar em mantê-lo atualizado por meio da seleção de novos itens refinados. No *Sprint*, são comumente aplicadas práticas que auxiliam

na previsão do progresso de desenvolvimento dos itens, como a utilização do *Burndown*, gráfico que correlaciona quantidade de trabalho e tempo (SCHWABER; SUTHERLAND, 2020). Importante destacar que a quantidade de trabalho é um valor definido pela equipe conforme a sua experiência, normalmente atribui-se um valor a cada item considerando o seu fator de complexidade.

O gráfico *Burndown* facilita a visualização da provável situação de desenvolvimento e entregas de itens, possibilitando avaliar atrasos ou adiantamentos. Na Figura 2.2, pode-se observar um exemplo da utilização do *Burndown*, no qual reflete o progresso de desenvolvimento dos itens de um *Sprint* com duas semanas úteis de duração. No exemplo, a pontuação ideal refere-se à quantidade de trabalho que deve ser entregue por dia para se ter um desenvolvimento linear, já a pontuação real reflete a quantidade de trabalho que de fato foi entregue pela equipe durante os dias do *Sprint*. Por meio do gráfico de exemplo, pode-se verificar que a equipe se manteve adiantada até o sexto dia, mas, após esse dia, encontrou–se em atraso. No entanto, concluiu com êxito o *Sprint*, pois no último dia realizou uma entrega acima da pontuação ideal diária, entregando cerca de 20 pontos.

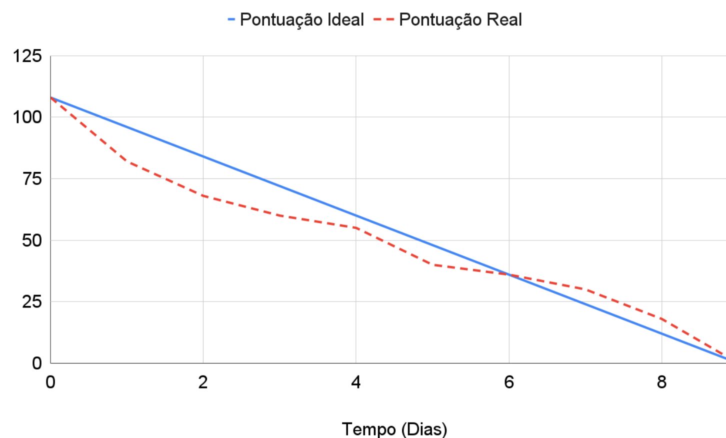


Figura 2.2 – Gráfico *Burndown*.
Fonte: Do autor (2022).

O *Sprint* se inicia com o evento de reunião de planejamento chamada *Sprint Planning*, onde a equipe determina o *Sprint Backlog*. Nessa reunião, a equipe se concentra em tornar o produto ainda mais valioso após a entrega dos itens selecionados. De início, concentra-se na questão de o que deve ser feito no *Sprint* e, em seguida, se discute de qual forma os itens devem ser feitos, considerando a experiência e a capacidade da equipe (SCHWABER; SUTHERLAND, 2020).

Após a reunião de planejamento, o *Sprint* prossegue com reuniões diárias chamadas de *Daily Scrum*. O propósito da *Daily Scrum* é fazer com que a equipe sustente uma rotina de supervisão a respeito dos itens do *Sprint Backlog* que estão sendo desenvolvidos. Essas reuniões devem ser breves, objetivas e devem ser realizadas diariamente no mesmo horário durante o *Sprint*. Segundo Schwaber e Sutherland (2020), as *Daily Scrums* são muito importantes, pois "melhoram as comunicações, identificam impedimentos, promovem a tomada de decisões rápidas e, consequentemente, eliminam a necessidade de outras reuniões".

No período final do *Sprint*, realiza-se a *Sprint Review*, evento no qual a equipe se reúne com os demais envolvidos no projeto para apresentar e verificar o resultado do *Sprint*. Como menciona Schwaber e Sutherland (2020), a *Sprint Review* não deve se limitar a uma apresentação, pois, durante sua realização, pode-se identificar mudanças ou oportunidades que devem ser incluídas no *Product Backlog*.

Por fim, tem-se o último evento do *Sprint*, a *Sprint Retrospective*, cujo objetivo é discutir sobre como o *Sprint* foi conduzido em relação aos elementos pelos quais a equipe se suporta, como "indivíduos, interações, processos, ferramentas e a sua *Definition of Done*", ressalta Schwaber e Sutherland (2020). Assim, a equipe identifica quais foram os problemas e quais as mudanças necessárias para tornar o próximo *Sprint* mais eficaz.

Para conduzir e manter a realização dos eventos e das práticas, o Scrum define uma equipe de três responsabilidades que, de acordo com Schwaber e Sutherland (2020), são:

- *Developers*: responsáveis pelo desenvolvimento de itens do *Sprint Backlog* e pela garantia da qualidade de cada novo *Incremento* do produto.

- *Product Owner*: responsável pelo produto e pela priorização de itens do *Product Backlog* para garantir a constante geração de valor do produto. Além disso, mantém esclarecidos à equipe os objetivos do produto e demais decisões.

- *Scrum Master*: visa garantir que as práticas do Scrum sejam realizadas eficientemente pela equipe sem impedimentos.

2.1.2 Contexto DTI

Durante a realização das atividades de estágio, adquiriu-se experiência prática com as principais práticas do Scrum. Como demonstra o Bloco 1 da Figura 2.1, a etapa inicial do fluxo

de desenvolvimento ágil empregado pela DTI e cliente consiste na utilização de práticas do Scrum.

Nesse contexto, cada equipe é normalmente composta pelo *Product Owner*, o *Scrum Master* e cerca de cinco desenvolvedores, sendo um deles o desenvolvedor líder, que se responsabiliza por manter uma maior proximidade com o *Product Owner*, visando adiantar o processo de *Refinamento* e obter importantes detalhes sobre a priorização de itens.

O desenvolvedor líder e o *Product Owner* também se atentam em acompanhar a situação do *Product Backlog* e do *Sprint Backlog*, para mantê-los organizados por meio da ferramenta Azure Boards[3]. Trata-se de uma importante ferramenta que, além de armazenar esses artefatos citados, permite à equipe customizar um quadro Kanban[4] para acompanhar os estágios de desenvolvimento de cada item, desde seu processo de refinamento até sua entrega.

A customização utilizada pela equipe define seis estágios principais: refinamento, onde constam os itens em processo de refinamento; selecionados, onde constam os itens adequados para o desenvolvimento; e os demais estágios que refletem as etapas do fluxo de desenvolvimento, como os estágios de desenvolvimento, revisão, homologação e entrega.

Outro ponto importante é em relação ao *Sprint*, evento para o qual se define uma duração de duas semanas úteis, especificamente no contexto do cliente em que se atuou, podendo variar conforme as exigências de cada cliente e equipe. A equipe também utiliza-se do *Burndown* para acompanhar o desenvolvimento dos itens durante todo esse período.

A equipe também considera a utilização da técnica *Behavior Driven Development* (BDD) para atingir uma melhor descrição dos itens durante o *Refinamento*. De acordo com North et al. (2006), ao se descrever um item a ser desenvolvido, normalmente se usa um modelo de história que facilita a identificação do valor gerado após a entrega. O modelo possui a seguinte estrutura:

Como um usuário;
Quero uma determinada funcionalidade;
Para ter um determinado benefício.

No entanto, ainda havia a necessidade de definir uma abordagem complementar para facilitar a identificação dos critérios de aceitação de um item por todos os envolvidos no pro-

[3] Disponível em <https://azure.microsoft.com/pt-br/products/devops/boards>.

[4] Kanban é um método de organização visual de fluxo de trabalho que auxilia na condução dos estágios para a realização de tarefas (TOTVS, 2022).

jeto (NORTH et al., 2006). Essa abordagem também possibilitaria a identificação de fragmentos funcionais aptos de serem verificados por testes automatizados. Com isso, definiu-se a técnica BDD, na qual se tem um modelo em termos de cenários, como a seguir:

Dado algum contexto inicial;
Quando ocorrer algum evento;
Então garantir um determinado resultado.

Nesse modelo, torna-se evidente que os termos escolhidos possuem como principal foco descrever o comportamento esperado da funcionalidade. Essa abordagem permite maior proximidade entre os envolvidos no projeto, pois se fundamenta no domínio do negócio para permitir a clientes e desenvolvedores uma comunicação sem ambiguidade.

Atentando às condições definidas para a condução do *Sprint*, realiza-se então o *Sprint Planning*. Nessa reunião de planejamento, os itens refinados são selecionados e se aplica uma técnica bastante utilizada em métodos ágeis, chamada de *Planning Poker*. Como define Grenning (2002), essa técnica consiste em estimar o esforço que um item exige para ser desenvolvido. Primeiro, um integrante da equipe repassa as informações sobre um determinado item a ser estimado, então os desenvolvedores escolhem um valor pertencente a uma sequência específica, por exemplo, Fibonacci. Posteriormente, todos revelam o valor escolhido e inicia-se uma discussão para decidir qual estimativa o item deve receber, sendo que valores mais altos representam um maior esforço de desenvolvimento. Após o consenso da equipe, o item é movido para o *Sprint Backlog*. Trata-se de uma prática importante, pois tais valores estimados são utilizados como pontuação para o acompanhamento do desenvolvimento no *Sprint* por meio do gráfico *Burndown*. A conclusão da reunião de planejamento ocorre após a soma das estimativas dos itens atingir a capacidade máxima de esforço predefinida pela equipe em um *Sprint*.

Após a *Sprint Planning*, são realizadas as *Daily Scrums*, onde cada integrante da equipe esclarece para os demais (i) o que foi desenvolvido desde a *Daily Scrum* anterior; (ii) o que será desenvolvido no dia atual; e (iii) se existe algum impedimento no desenvolvimento que necessite de suporte para ser resolvido. A *Daily Scrum* sempre é feita no início do dia e a equipe brevemente discute sobre as últimas atualizações do *Sprint Backlog* e do *Burndown*.

No encerramento do *Sprint*, a equipe realiza a *Sprint Review*, onde são apresentadas e discutidas todas as entregas realizadas, enfatizando importantes informações para demonstrar os objetivos considerados, os resultados alcançados e os valores gerados. Esse evento é conduzido

principalmente pelo *Product Owner*, visando uma comunicação objetiva para esclarecer aos líderes de negócio as entregas e análises realizadas.

Posteriormente, realiza-se a *Sprint Retrospective*, conduzida pelo *Scrum Master*. Nesse evento, são levantados pontos categorizados como: positivo, negativo ou a melhorar. Em seguida, a equipe realiza discussões e reflexões sobre cada ponto levantado, visando aumentar a eficiência do *Sprint* seguinte.

Um ponto importante se destacar é que, durante o *Sprint*, a equipe se concentra em desenvolver e entregar todos os itens contidos no *Sprint Backlog* para posteriormente apresentá-los na *Sprint Review*. No entanto, há casos atípicos em que não é possível completar totalmente a etapa de entrega aos usuários finais, por exemplo, em situações onde a equipe enfrenta problemas de infraestrutura que acarretam impedimentos ou incidentes críticos que devem ser corrigidos durante o *Sprint* vigente, ocasionando atrasos.

Como destacado, o Scrum define várias práticas que auxiliam a conduzir o desenvolvimento de soluções adaptativas, priorizando-o por meio do *Sprint Backlog* conforme o valor que será gerado aos envolvidos no projeto. Tais soluções podem exigir a criação de código por parte da equipe, a qual deve usufruir de práticas que possibilitem um desenvolvimento mais seguro e de fácil controle e integração. No entanto, deve-se atentar em mantê-lo flexível, tendo em vista que possa haver itens com alto grau de dependência entre si durante a codificação. Para atingir isso, estabelecem-se importantes processos, envolvidos em uma etapa seguinte chamada Codificação, conforme se segue no fluxo da Figura 2.1.

2.2 Codificação

2.2.1 Fundamentação Teórica

No processo de desenvolvimento de um software, normalmente ocorre muitas mudanças ocasionadas pelo acréscimo, a atualização ou a exclusão de funcionalidades, ou de documentações. Se negligenciado, o acúmulo de modificações pode gerar confusão para a equipe e também ocasionar a perda de importantes informações durante o desenvolvimento (ZOLKIFLI; NGAH; DERAMAN, 2018). Perante a isso, torna-se necessário o emprego de atividades para melhor gerenciar as mudanças ao longo do ciclo de vida do software. Esse conjunto de atividades, no qual inclui garantir a identificabilidade, a comunicabilidade e a segurança quanto

às alterações, constitui a Gestão de Configuração de Software (GCS) (PRESSMAN; MAXIM, 2016).

Considerando tais circunstâncias, a GCS se beneficia ao focar em sustentar um regimento que visa manter um Sistema de Controle de Versões (SCV) eficaz, por meio do qual se permite aos itens em desenvolvimento, adequadamente identificados, uma evolução distribuída e concorrente, porém disciplinada (SOFTEX, 2016).

De acordo com Pressman e Maxim (2016), para realizar o controle de versões apropriadamente, um SCV deve fornecer ou integrar-se a:

> "[...] um banco de dados de projeto (repositório) que armazena todos os objetos de configuração relevantes; um recurso de gestão de versão que armazena todas as versões de um objeto de configuração (ou permite que qualquer versão seja construída usando diferenças das versões anteriores); uma facilidade de construir que permite coletar todos os objetos de configuração relevantes e construir uma versão específica do software."

O SCV pode ser classificado como centralizado, onde há um banco de dados de artefatos de projeto (repositório) em um único servidor central, ou distribuído, onde há um repositório para cada usuário (ZOLKIFLI; NGAH; DERAMAN, 2018). Na Figura 2.3, pode-se observar uma representação de um SCV centralizado, no qual consta um servidor com um repositório onde se realiza todo o controle de versão a cada nova modificação enviada pelos desenvolvedores.

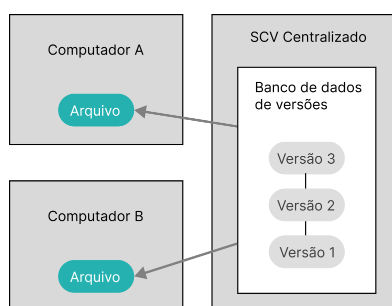


Figura 2.3 – Representação do SCV Centralizado.
Fonte: Adaptado de Chacon e Straub (2014).

O modelo de SCV distribuído ou descentralizado, como representado na Figura 2.4, no que lhe concerne, fornece um repositório individual a cada desenvolvedor, permitindo a consulta do histórico de alteração de cada artefato sem necessitar de conexão com um servidor. Nesse modelo descentralizado, o acesso a um servidor central se torna necessário quando o objetivo é unificar em um único repositório remoto as modificações realizadas pelos desenvolvedores em seus respectivos repositórios locais.

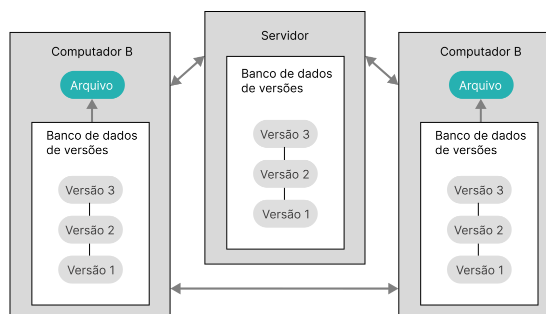


Figura 2.4 – Representação do SCV Descentralizado.
Fonte: Adaptado de Chacon e Straub (2014).

Existem diversas ferramentas de SCV, sendo o Git[5] uma das mais populares que permite o trabalho descentralizado (MILLER, 2021). O Git facilita a conexão com os servidores para a transferência de artefatos entre repositórios remotos e locais, além de fornecer inúmeras funcionalidades que o tornam um SCV eficaz. Algumas das suas principais funcionalidades, destacadas por Loeliger e McCullough (2012), possuem os seguintes objetivos:

- *commit*: demarcar uma alteração ou um conjunto de alterações em um histórico. Cada *commit* efetuado se conecta com o anterior para construir uma linha de desenvolvimento, formando assim um histórico de alterações.

- *branch*: criar uma ramificação que se origina de um ponto específico em que se encontra os artefatos do projeto, permitindo conduzir uma nova linha de desenvolvimento em paralelo e realizar novas alterações sem afetar o histórico principal de onde a ramificação se originou.

- *merge*: unificar duas *branches*, fazendo com que os seus históricos de alterações sejam automaticamente unificados considerando as alterações desde o último *commit* compartilhado. Durante a realização do *merge*, pode ocorrer de duas alterações terem sido feitas em um mesmo ponto do mesmo artefato, o *merge* então dispõe as duas alterações para os desenvolvedores realizarem um procedimento manual de resolução de conflitos. Ao se concluir um *merge*, um novo *commit* é criado como resultado da união.

- *tag*: criar uma referência em um determinado *commit* para destacá-lo como sendo o fim de uma sequência de alterações periódica. Normalmente usado para representar o lançamento de uma nova versão.

[5] Disponível em <https://git-scm.com/>.

- *pull*: atualizar o repositório local, ou uma *branch* local, para torná-lo idêntico ao remoto.
- *push*: atualizar o repositório remoto, ou uma *branch* remota, para torná-lo idêntico ao local.

Como mencionado, o Git contém muitas funcionalidades além das listadas anteriormente, porém, apenas com a utilização das principais já se possibilita cumprir grande parte das exigências do processo de controle de versões. Além disso, o Git também permite às equipes empregar uma variedade de fluxos de trabalho dependendo do contexto no qual estão inseridas, pois garante flexibilidade e torna a transferência de informações entre repositórios algo de fácil controle e integração (SPINELLIS, 2012).

Como menciona Rivero et al. (2021), diante da existência de vários modelos de controle de fluxo de trabalho com Git, o *Git Flow* se destaca como sendo um dos modelos de maior popularidade. Definido por Driessen (2010), o *Git Flow* é um conjunto de procedimentos adotado por todos da equipe para atingir um melhor controle do desenvolvimento, principalmente, quanto à utilização das ramificações fornecidas pelo Git.

A Figura 2.5 demonstra um exemplo de fluxo de desenvolvimento considerando o modelo *Git Flow*. No exemplo, os *commits* são representados por círculos coloridos para destacar o tipo de *branch* que faz parte. Além disso, cada *branch* é representada por uma linha de desenvolvimento em relação ao tempo, onde consta a sequência de *commits* da *branch*. No início do fluxo, pode-se observar a existência de um primeiro *commit* realizado na *master*, *branch* principal onde constam os *commits* que normalmente refletem as versões do software. Para prosseguir com o desenvolvimento, cria-se uma nova *branch*, originada da *master*, chamada *develop*, onde constam *commits* relacionados às integrações das funcionalidades ou das mudanças desenvolvidas. A *master* e a *develop* possuem tempo de existência indeterminado, pois são a base das *branches* de suporte. A *feature* é a primeira *branch* de suporte, usada para manter o desenvolvimento de uma nova funcionalidade, a *develop* é a sua origem e o seu destino. Como se pode observar no fluxo do exemplo, duas *branches features* são criadas e, ao final do desenvolvimento, são unificadas novamente à *develop*. Quando há funcionalidades integradas o bastante, cria-se, a partir da *develop*, a *branch* de suporte *release*, onde é realizada a preparação para o lançamento de uma nova versão do software. A cada novo *commit* na *release*, atualiza-se a *develop* por meio da união das duas *branches* e, ao final da preparação, unifica-se a *release* com a *master*, atentando em atribuir uma nova *tag* na *master* para indicar o lançamento de uma nova versão do software. Por fim, tem-se a última *branch* de suporte, nomeada *hotfix*, cujo

objetivo é possibilitar a imediata resolução de problemas emergentes, portanto, deve se originar da *master* e, posteriormente, se unificar à *develop* e também à *master*.

Figura 2.5 – Exemplo de um fluxo no modelo *Git Flow*.
Fonte: Adaptado de Vietro (2015).

2.2.2 Contexto DTI

A etapa de Codificação se inicia após os ritos de planejamento definidos pelo Scrum, é o período no qual se desenvolve a solução de itens do *Sprint Backlog* que exige a utilização de uma linguagem de programação. Além do desenvolvimento do código, executa-se um importante processo de controle de versões para melhor conduzir e, posteriormente, integrar o trabalho realizado pela equipe.

Diante de sua popularidade e relatos positivos, o modelo *Git Flow* pode ser uma alternativa promissora para se adotar durante a realização do processo de controle de versões. No fluxo de desenvolvimento ágil representado no Bloco 2 da Figura 2.1, pode-se observar que no cenário da DTI e cliente, emprega-se o modelo *Git Flow* durante a etapa de Codificação. A etapa se inicia após a seleção de um item do *Sprint Backlog* que esteja apto para o desenvolvimento. Em seguida, executam-se os procedimentos definidos pelo *Git Flow* previamente descritos e detalhados na Seção 2.2.1. Assim como especificado pelo modelo, adota-se a mesma configuração de *branches* com suas respectivas origens e destinos: *master*, *develop*, *release*, *feature* e *hotfix*.

A execução do primeiro procedimento se dá pela atualização do repositório local para obter as recentes alterações do projeto, mantidas em seu repositório remoto. Após a atualização do repositório local, cria-se uma *branch* conforme o tipo do item a ser desenvolvido: *hotfix* para a correção de incidente ou *feature* para a criação de uma nova funcionalidade. Considerando o fluxo simplificado do Bloco 2, após a criação de uma ramificação de *feature*, desenvolve-se

nessa nova *branch* o código que vai solucionar e satisfazer todos os requisitos do item em desenvolvimento, atentando em demarcar, por meio de *commits*, as alterações realizadas conforme a necessidade, pois o conjunto de *commits* efetuados na *branch* é tido como uma referência ao código desenvolvido, delimitando e possibilitando o rastreio do incremento realizado.

Para a criação do código, dado o contexto das equipes em que se atuou durante as atividades do estágio, utiliza-se das linguagens de programação C# e JavaScript, do *framework* ASP.NET e da biblioteca React.

Uma grande parte do código criado é composta por implementações de testes de unidade, i.e., testes que validam o comportamento dos menores trechos ou módulos de código, identificados como unidades isoladas. Esses testes auxiliam no controle de qualidade do código e evitam impactos inadvertidos após uma determinada alteração (RUNESON, 2006). Por conta de sua importância, empenha-se em criar um elevado número de testes de unidade para abranger todos os fluxos possíveis durante a execução de uma funcionalidade, buscando garantir o seu comportamento esperado.

Além disso, adota-se a prática *Test Driven Development* (TDD) que, segundo Beck (2003), consiste em orientar o desenvolvimento de código atrelando-o à criação de testes automatizados. Com essa prática, a criação do teste automatizado é antecipada ao desenvolvimento do trecho do código que será testado e validado. Para isso, o TDD define três etapas:

- Vermelha: um novo teste é desenvolvido e executado. Contudo, o teste deve falhar devido à inexistência ou inconformidade do código que será testado.
- Verde: um código mínimo é desenvolvido ou modificado visando satisfazer o teste criado anteriormente. Após a sua criação, todos os testes devem ser executados com sucesso.
- Refatoração: o código desenvolvido é refatorado com o intuito de aumentar a sua qualidade, removendo duplicações, ajustando métodos longos, entre outras melhorias.

As etapas do TDD são realizadas de maneira cíclica, implicando em um desenvolvimento gradativo de uma nova funcionalidade, mas que provê uma cobertura de testes aprimorada a cada novo incremento de trecho de código.

A etapa de Codificação se conclui com o envio das alterações referentes ao desenvolvimento para o repositório remoto, atualizando-o por meio de um *push* realizado na *branch* local utilizada. A partir desse momento, inicia-se a etapa de Integração Contínua, onde o desenvolvedor responsável pela codificação realiza um processo de comunicação com a equipe para

revisar e integrar o código da melhor forma, além de acompanhar uma sequência de verificações automatizadas aplicadas no código desenvolvido ou integrado.

2.3 Integração Contínua

2.3.1 Fundamentação Teórica

A Integração Contínua ou *Continuous Integration* (CI) é uma prática de desenvolvimento ágil em que são diariamente realizadas integrações e testes a cada nova alteração ou novo item desenvolvido, reduzindo o número de problemas lidados pela equipe durante uma integração de software, e tornado-a um processo mais rápido e seguro (FOWLER; FOEMMEL, 2006). Em consequência disso, essa prática possibilita um aumento da execução de testes e diminuição não apenas do tempo investido em busca de problemas, como também do custo para solucioná-los (AMRIT; MEIJBERG, 2018).

Como destaca Fitzgerald e Stol (2017), o processo de integração contínua normalmente possui uma configuração para acionamentos automáticos e ordenados de compilação de código, execução de testes de unidade, validação da porcentagem de cobertura do código pelos testes, validação da qualidade e padronização do código, e construção de pacotes compilados para a operação do software. Por isso, para se alcançar o funcionamento adequado dos processos de CI, deve-se atentar a importantes práticas que, de acordo com Fowler e Foemmel (2006), são:

- Manter uma única fonte de repositório: além da utilização de um SCV adequado, deve-se manter um só repositório como fonte única e prepará-lo para permitir a independente configuração do software em um servidor desocupado. Diante disso, deve-se colocar nesse repositório todos os artefatos necessários para a execução de testes e para as configurações de banco de dados, de dependências de terceiros e de compilação.

- Automatizar o processo de *build*: a configuração de um servidor e a preparação da execução do processo de *build* (isto é, a compilação e construção de pacotes) é algo custoso de se fazer manualmente. Por isso, é muito importante a construção de *scripts* com o intuito de automatizar e tornar eficiente esse processo.

- Execução de testes ao realizar *build*: a inclusão de uma ferramenta para configurar a execução de testes automatizados que indique falhas e interrompa o processo de *build* quando alguma falha ocorrer. Isso auxilia na captura de problemas antes da operação do software.

- Comunicação e compromisso com a linha de desenvolvimento principal: a comunicação é essencial durante o processo de integração, logo, deve-se adotar meios para torná-la eficaz. A comunicação também deve ser frequente entre todos da equipe, principalmente para se evitar grandes integrações na linha de desenvolvimento principal e assim facilitar a resolução de conflitos.

- *Builds* frequentes da linha principal em um servidor de CI: a utilização de um servidor para realizar *builds* automáticos a cada alteração incluída na linha principal de desenvolvimento, permitindo a realização de testes frequentes de compilação. Os servidores de CI também possuem a funcionalidade de envio de notificações sobre a situação do *build*.

- Corrigir falhas de compilação rapidamente: a correção de falhas de compilação, principalmente na linha principal, deve ser feita de forma rápida, com o intuito de não manter o fluxo de integrações interrompido por muito tempo. Normalmente a correção é realizada revertendo as alterações incluídas.

- Acelerar o processo de *build*: a prática de realizar integrações frequentes pode consumir bastante tempo, pois exige uma maior quantidade de compilações e testes, executados constantemente. Diante disso, pode-se utilizar da ferramenta *pipeline*, onde são definidos estágios ordenados, possibilitando priorizar ações de maior dependência. A priorização da compilação da linha principal após novas alterações, por exemplo, permite um retorno mais rápido da situação do processo de integração. Como estágio posterior do *pipeline*, pode-se executar testes mais lentos, além da possibilidade de paralelização.

- Fornecer uma interface para visualização dos estados da integração: uma interface intuitiva e que dê *feedbacks* claros auxilia no acompanhamento das compilações e demais ações durante uma integração.

2.3.2 Contexto DTI

Após a conclusão da etapa de Codificação, que ocorre após a atualização do repositório remoto com novas alterações no código, prossegue-se com uma sequência de ações referentes a etapa de Integração Contínua.

Nessa etapa, são executados os últimos procedimentos do *Git Flow*, conduzidos com o auxílio da ferramenta de hospedagem e gerenciamento de repositórios Azure Repos[6], mantene-

[6] Disponível em <https://azure.microsoft.com/pt-br/services/devops/repos/>.

dor dos repositórios remotos dos projetos. Além de se integrar com as funcionalidades do Git, o Azure Repos fornece uma interface que possibilita a prática da revisão por pares no código desenvolvido, utilizando-se de *Pull Requests* (PRs). A funcionalidade de PR pode ser definida como uma solicitação ou sugestão de união de ramificações, onde a equipe de desenvolvimento é notificada e convidada a avaliar um conjunto de alterações que está em condição de integração.

De acordo com Wiegers (2002), a revisão por pares ou *peer review* é uma prática onde outros desenvolvedores, além do autor da entrega, examina a qualidade do que se desenvolveu. Essa atividade pode ser realizada com o auxílio de um *checklist*, tornando a inspeção padronizada.

Conforme representado no Bloco 3 da Figura 2.1, os procedimentos do *Git Flow* da etapa Integração Contínua acontecem conforme a definição do modelo. Por se tratar de uma representação simplificada do fluxo, voltado à criação de uma simples funcionalidade, não se considera a *branch hotfix* e demais cenários específicos. Outro ponto a ser ressaltado é que, antes de cada *merge*, utiliza-se de PRs para possibilitar à equipe realizar as avaliações necessárias que resultarão na aprovação ou reprovação da realização do *merge*.

De início, realiza-se a união da ramificação *feature* com a *develop* para integrá-la com as demais alterações e funcionalidades desenvolvidas. Nesse momento, cria-se um PR por meio da interface do Azure Repos, onde são listadas todas as alterações realizadas durante o tempo de vida da *branch*. Concluído o *peer review*, aprova-se o PR e a união das *branches* é realizada. Em seguida, cria-se a *branch release* a partir da *develop* atualizada para preparar o lançamento de uma nova versão do software. Concluída a preparação, cria-se novamente um PR para realizar um novo *peer review* das possíveis correções efetuadas na *release*. Por fim, após a aprovação do PR anterior, realiza-se o *merge* das *branches release* e *master*, definindo a nova versão do software.

Prosseguindo no fluxo da etapa Integração Contínua, têm-se as ações referentes ao Servidor CI, orquestrado pela ferramenta Azure Pipeline[7]. Pode-se observar que, em cada *branch* do Azure Repos, são aplicadas ações de (i) *build*, para a compilação e a construção dos pacotes de código; (ii) testes automatizados, para a execução dos testes de unidade automaticamente; e (iii) análise estática de código com a ferramenta SonarCloud[8], para a validação da cobertura por testes, da qualidade e da padronização do código. As ações são aplicadas às *branches* a cada

[7] Disponível em <https://azure.microsoft.com/pt-br/services/devops/pipelines/>.
[8] Disponível em <https://www.sonarsource.com/products/sonarcloud/>.

alteração realizada. Portanto, a cada novo *push*, *merge* ou modificação de arquivo no repositório remoto, o Servidor de CI é acionado.

A etapa se encerra quando a validação da qualidade do código é bem sucedida, então prossegue-se para a Entrega Contínua, etapa cujo objetivo é estender e complementar o processo de CI para possibilitar automatização e facilitar a implantação do software após a integração previamente realizada.

2.4 Entrega Contínua

2.4.1 Fundamentação Teórica

A Entrega Contínua ou *Continuous Delivery* (CD) é uma prática adotada no desenvolvimento de software ágil e consiste em determinar pequenos ciclos de entrega de incrementos do software, certificando que uma nova versão possa ser lançada a qualquer momento de forma rápida, confiável e eficiente (CHEN, 2015). Para atingir isso, Humble e Farley (2014) definem o *pipeline* de implantação que "baseia-se no processo de integração contínua e é essencialmente o princípio de integração contínua levado à sua conclusão lógica", tendo como finalidade automatizar a entrega e implantação de um sistema em ambiente de homologação, servidor onde se realiza testes, ou em ambiente de produção, servidor onde ocorre a operação do sistema para os usuários finais.

A Figura 2.6 basicamente ilustra de que modo a CD estende a CI. Observa-se que a CD inclui etapas que não apenas focam em verificar o grau de aptidão do software para um lançamento adequado e rápido (estágio de realização de testes de aceitação), como também proporciona sua imediata operação, efetuada no estágio de implantação em ambiente de produção, após realizadas as verificações necessárias.

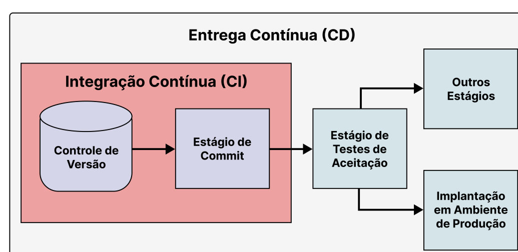


Figura 2.6 – Relação entre Integração Contínua (CI) e Entrega Contínua (CD).
Fonte: Adaptado de Laukkanen, Itkonen e Lassenius (2017).

De acordo com Arachchi e Perera (2018), o *pipeline* de implantação (também chamado de *pipeline* de CI e CD), resultante da combinação das práticas de CI e CD, estimula as equipes

a frequentemente entregarem melhorias de software devido à automatização de *builds*, implantações em servidores, e ao fornecimento de *feedbacks* constantes durante a execução de suas ações, proporcionando maior produtividade e eficiência nos processos de desenvolvimento ágil.

Na Figura 2.7, consta-se a representação de um exemplo de estágios do *pipeline* de CI e CD. Segundo Humble e Farley (2014), o início da execução do processo do *pipeline* ocorre no momento em que o sistema de integração contínua identifica, por meio do sistema de controle de versões, a realização de uma nova alteração de artefatos. A partir disso, segue-se com a execução dos estágios do *pipeline* que, de acordo com Chen (2015) e Humble e Farley (2014), podem ser definidos da seguinte forma:

Figura 2.7 – Representação do exemplo de estágios do *pipeline*.
Fonte: Adaptado de Chen (2015).

- Estágio de *commit* ou integração: envolve as etapas de execução de *build*, testes automatizados, análise estática de código e geração de artefatos compilados.
- Estágio de testes de aceitação: período de validação no qual um ambiente semelhante ao de produção é configurado automaticamente. Assim, implanta-se o software nesse ambiente e executam-se testes para confirmar se o software atende os requisitos de usuários.
- Estágio de testes de capacidade: verificação de possíveis alterações de desempenho do software após as mudanças realizadas. Configura-se um novo ambiente específico para a realização dos testes.
- Estágio de testes manuais: realização de testes manuais, necessários dependendo do contexto da equipe. Para isso, o *pipeline* novamente prepara um ambiente para a realização dos testes.
- Estágio de produção: implantação do software, devidamente testado, em ambiente de produção, por meio de um único acionamento manual.

Vale ressaltar que, em caso de erro em algum dos estágios, o *pipeline* interrompe a execução do estágio e notifica o desenvolvedor. Também é importante certificar a existência de uma forma de reversão de implantação no *pipeline*, para rápido contorno após a identificação de comportamentos inesperados do software em ambiente de produção.

Observa-se que o *pipeline* possibilita a existência de vários estágios voltados à realização de testes específicos, em que se configura um ambiente propício para implantar o software para testá-lo adequadamente. Esses estágios podem ou não ser acionados automaticamente, porém, prefere-se manter um acionamento simples que permita à equipe requisitar uma rápida implantação pelo *pipeline* (HUMBLE; FARLEY, 2014).

2.4.2 Contexto DTI

Concluídas as ações automáticas de integração, executadas pelo Servidor de CI na etapa de Integração Contínua, iniciam-se os procedimentos referentes à etapa de Entrega Contínua, efetuados para realizar a implantação, os testes e a entrega da nova versão do software de forma eficiente.

A etapa de Entrega Contínua no contexto do desenvolvimento ágil empregado pela DTI e cliente é ilustrada no Bloco 4 da Figura 2.1, em que são dispostos três servidores contendo tipos de ambientes específicos nos quais se implanta o sistema. As implantações em ambientes e seus respectivos testes são vistos como estágios de CD, também orquestrados pela ferramenta Azure Pipeline, assim como as ações de CI foram previamente.

De início, tem-se o ambiente de desenvolvimento, caracterizado como um ambiente menos restritivo e de menor custo. Esse ambiente permite a rápida verificação do comportamento do sistema após uma implantação, além de permitir a realização de testes de uma forma menos impactante, pois conta com uma base de dados de testes de fácil manipulação e de baixa dependência por parte das equipes.

Em seguida, implanta-se o sistema em um ambiente de homologação, o qual possui maior semelhança com o ambiente de produção. Nesse momento, a equipe inicia os testes de aceitação, realizando-o manualmente com o auxílio de um roteiro de teste previamente definido para as funcionalidades ou alterações desenvolvidas.

Após a execução bem sucedida dos testes, aciona-se manualmente o estágio de implantação do sistema em ambiente de produção. No instante do acionamento, o *pipeline* verifica se uma *tag* de nova versão foi devidamente criada na *branch* principal *master* e se foram realizadas todas as aprovações necessárias para a implantação final.

No fluxo também se pode notar que os estágios de CD são executados pelo *pipeline* apenas nas ramificações *develop*, que atinge como estágio máximo a implantação em ambiente

de homologação, e *release*, em que se possibilita a implantação do software em ambiente de produção para ser utilizado pelos usuários finais.

Com a nova versão do software em operação para os usuários finais, a etapa de Entrega Contínua se conclui e prossegue-se para a etapa de Monitoramento, onde se utiliza de ferramentas por meio das quais a equipe mantém um constante acompanhamento da operação e do desempenho do software em ambiente de produção.

2.5 Monitoramento

2.5.1 Fundamentação Teórica

No processo de desenvolvimento de software, são frequentemente tomadas decisões que podem gerar impactos no seu desempenho operacional. Contudo, analisar a relevância de tais impactos somente é possível após a implantação do sistema (PÉREZ; WANG; CASALE, 2015). Tendo isso em vista, realizar adequadamente o monitoramento de um software implantado é uma prática essencial para garantir a confiabilidade e o desempenho esperados, além de possibilitar antecipada detecção de problemas relacionados a essas qualidades (HOORN et al., 2009). De acordo com Amaradri e Nutalapati (2016), o constante monitoramento do comportamento do sistema e de sua infraestrutura permite se obter importantes dados que contribuem para a otimização contínua do software.

A automatização de processos relacionados à integração e implantação do software fornece significativa vantagem ao desenvolvimento ágil, proporcionando rápidos incrementos e entregas aos usuários finais. Porém, como destacam Liu, Li e Liu (2014), a adoção da prática de monitoramento é um fator decisivo quando se visa atingir consistência quanto à otimização do sistema durante o seu percurso de numerosos ciclos de desenvolvimento e implantação. Do mesmo modo, torna-se vantajosa a utilização de mecanismos automatizados para a realização do monitoramento. Segundo Gottesheim (2015), adotar esses mecanismos em todos os ambientes, desde o ambiente de integração contínua até o ambiente de produção, permite-se padronizar e se ter melhor medição e compreensão do desempenho do sistema. Além disso, também se torna possível realizar ações de contingência, a fim de evitar a propagação de problemas de desempenho nos ambientes no decorrer da execução do *pipeline* de implantação (WALLER; EHMKE; HASSELBRING, 2015).

De acordo com Hoorn et al. (2009), o monitoramento pode ser empregado em qualquer camada do sistema de software, desde a análise dos recursos de hardware até a análise do comportamento do software durante sua operação. Para isso, utiliza-se de ferramentas que fornecem funcionalidades que, por exemplo, não apenas monitoram o uso de CPU, memória, espaço livre e tráfego de rede, como também disparam notificações de alerta quando são detectados problemas que afetam a integridade do sistema. Essas ferramentas também contam com painéis interativos constando gráficos que refletem a situação dos recursos de infraestrutura e da aplicação, facilitando a criação de relatórios e o planejamento de atualizações realizados pela equipe (EBERT et al., 2016).

De acordo com Ahmed et al. (2016), tais ferramentas, que podem ser consideradas ferramentas de *Application Perfomance Management* (APM), periodicamente coletam e fornecem métricas, como o tempo de resposta às requisições em aplicações web. A análise de desempenho realizada pode ser baseada em linha de base, em que se consulta históricos de desempenho para definir uma métrica de referência ou por valor limite predefinido que, se excedido, indica uma anomalia no desempenho da aplicação.

2.5.2 Contexto DTI

Após a efetiva implantação do software em ambiente de produção, considerado o último estágio da etapa de Entrega Contínua, é necessário certificar que sua operação ocorra de maneira confiável e com desempenho satisfatório durante a interação com os usuários finais. Essas verificações são realizadas na etapa de Monitoramento, onde a equipe utiliza ferramentas previamente configuradas para exibir informações sobre a situação dos recursos utilizados durante a execução do software.

Como se pode observar no Bloco 5 da Figura 2.1, a ferramenta de APM Datadog[9] é predominantemente utilizada pela equipe para a realização do monitoramento dos sistemas implantados em ambiente de produção. O Datadog é configurado para monitorar e alertar em casos de identificação de anormalidades no tráfego de rede e de surgimento de novos erros após a última implantação, além de verificar a ocorrência de altas taxas de erro em requisições ou de alto valor médio de latência das requisições, analisados com base nos valores limites predefinidos. A ferramenta também é configurada para que as notificações de alerta sejam enviadas a um

[9] Disponível em <https://www.datadoghq.com/>.

canal específico da principal ferramenta de comunicação da equipe, permitindo mobilizações rápidas para realizar correções.

A etapa de Monitoramento é a última etapa do fluxo cíclico de desenvolvimento ágil no contexto da DTI e cliente, porém as ações de acompanhamento do desempenho das aplicações são realizadas durante todo o período de operação do software.

2.6 Considerações finais

Nesta seção, foram apresentados os conceitos e as ferramentas utilizados durante o desenvolvimento ágil, dispondo suas relações e importância em cada etapa do fluxo de desenvolvimento experienciado durante as atividades realizadas no estágio na DTI Digital. Por se tratar de um processo fundamental para efetivas entregas, o fluxo foi observado e vivido atenta e intensamente pelo estagiário.

Importante ressaltar que o ciclo de desenvolvimento ágil não se restringe em obrigatoriamente completar todas as etapas para retornar à etapa inicial. Durante o desenvolvimento ágil, frequentemente se fazem necessárias mudanças com base na coleta de *feedbacks* fornecidos por ações executadas em qualquer etapa do fluxo. Um exemplo disso é a constatação de código de baixa qualidade durante a análise estática de código, implicando na realização de novos *commits* na etapa de codificação para correções.

Além disso, também não se obriga a conclusão de uma etapa para prosseguir o fluxo, principalmente na etapa do Scrum, pois nessa etapa se visa concluir o processo de desenvolvimento dos itens antes da realização das últimas reuniões do Sprint, possibilitando a apresentação de todos os itens priorizados. Outro exemplo é a relação das etapas de Codificação e Integração Contínua, em que, visando maior segurança de armazenamento de códigos em desenvolvimento, frequentemente se torna necessário enviá-los para o repositório remoto, apesar de ainda não se encontrarem em condições para a integração.

Outra observação é que o fluxo representado foi elaborado tendo em vista as oportunidades proporcionadas pelas atividades realizadas, tornando-o direcionado e limitado a tais circunstâncias.

3 ATIVIDADES REALIZADAS

Nesta seção, são dispostas as atividades realizadas durante o estágio na DTI Digital, período em que o estagiário atuou em duas equipes. A principal atividade realizada durante a atuação na primeira equipe tratou-se da criação de duas interfaces para uma aplicação web, enquanto a atuação na segunda equipe se deu na criação e refatoração de funcionalidades para aplicações de microsserviço. Vale ressaltar que, por fins sigilosos, não se aborda detalhes da lógica de negócio do cliente, utilizando-se, portanto, de exemplos apartados.

3.1 Criação de *Single Page Application*

Considerando a necessidade de o cliente desenvolver uma nova solução por meio de uma aplicação web, a primeira equipe na qual o estagiário atuou iniciou o desenvolvimento de uma *Single Page Application* (SPA), instaurando itens no *Product Backlog* dentre os quais dois se categorizaram como atividade de estágio: a criação da interface de conteúdo expirado e a criação da interface de validação de usuário.

De acordo com Jadhav, Sawant e Deshmukh (2015), SPA é uma aplicação web composta por componentes individuais em que todos os recursos da aplicação são inteiramente carregados logo na primeira requisição realizada ao servidor. Tais componentes, no que lhe concerne, são exibidos ou substituídos por outros na página web ao decorrer da interação com o usuário, sem a necessidade de recarregar toda a página.

Para a criação da SPA, utilizou-se da biblioteca React[1] da linguagem JavaScript, tecnologia adotada pelo cliente por conta da biblioteca facilitar consideravelmente a criação de interfaces web para o usuário. No React, a implementação do código das interfaces ocorre por meio da linguagem JSX[2], uma extensão da linguagem JavaScript que possibilita sua utilização junto à sintaxe de *Extensible Markup Language* (XML).

Com isso, o React possibilita a criação de partes independentes de interface de usuário com o uso de classes ou funções da linguagem JavaScript, as quais são denominadas componentes. Além disso, cada componente possui o seu próprio conjunto de estados, dados voláteis utilizados para controlar a mudança de comportamento do componente na página web, e um conjunto de propriedades, parâmetros não modificáveis que são recebidos e utilizados para permitir a troca de informações entre os componentes. Na Figura 3.1, há uma representação da

---

[1] Disponível em <https://pt-br.reactjs.org/>.
[2] Disponível em <https://facebook.github.io/jsx/>.

relação desses fundamentos, onde se pode observar a relação de um componente React com o *Document Object Model* (DOM), na qual se dá o processo onde a biblioteca realiza a preparação para a inclusão do componente na página HTML para a renderização.

Figura 3.1 – Representação da relação dos fundamentos do React.
Fonte: Adaptado de Sakhniuk (2022).

O primeiro passo realizado para iniciar o desenvolvimento da SPA foi a comunicação e o alinhamento com a equipe de *design* responsável pela elaboração dos fluxos e do *design* das interfaces. Na Figura 3.2, dispõe-se um exemplo simplificado de um fluxo semelhante ao que deveria ser satisfeito pela aplicação web.

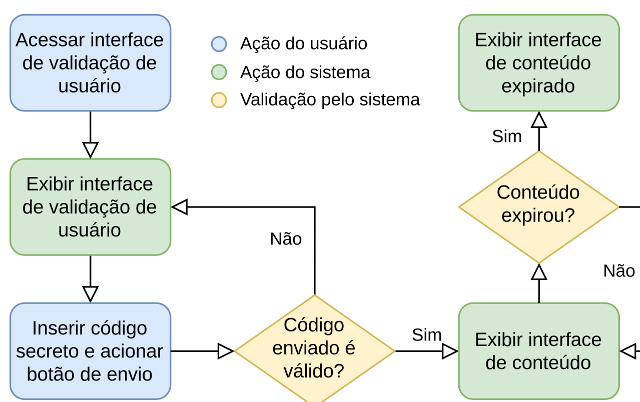


Figura 3.2 – Exemplo semelhante ao fluxo da aplicação web desenvolvida.
Fonte: Do autor (2022).

De início, tem-se a interface de validação de usuário por meio da qual se insere um código secreto. Se o código estiver correto, prossegue-se para a próxima interface onde um conteúdo restrito ao usuário é exibido. Nessa interface, verifica-se se o conteúdo expirou ou se ainda está apto para ser exibido. Em caso de expiração, prossegue-se para a interface de conteúdo expirado onde se exibe uma mensagem sobre a situação atual do conteúdo.

3.1.1 Interface de conteúdo expirado

O primeiro item desenvolvido foi a interface de conteúdo expirado. Como se pode observar no exemplo da Figura 3.3, a interface contém um cabeçalho com o logotipo da aplicação, um texto e uma imagem para informar a expiração do conteúdo. Para a criação da interface, considerou-se desenvolver um componente para cada tipo de item da página, os quais foram

nomeados `Imagem`, `Texto`, `Cabecalho` e `Layout`. O componente `Cabecalho`, localizado na parte superior da interface, é composto por uma figura do logotipo da aplicação. Assim sendo, foi possível reutilizar o componente `Imagem` para exibir as duas figuras presentes na interface, visto que esse componente mantém o código que possibilita a exibição de figuras. A criação do componente `Layout` também foi de grande importância, pois se responsabiliza pela organização dos demais componentes da interface, permitindo que interfaces semelhantes sejam organizadas rapidamente com a reutilização do componente.

Figura 3.3 – Interface de conteúdo expirado da aplicação web.
Fonte: Do autor (2022).

Concluído o desenvolvimento, prosseguiu-se no fluxo de desenvolvimento ágil para implantar a aplicação incrementada em ambiente de homologação. Após a implantação, notificou-se a equipe de *design* para que fosse realizada a validação do item desenvolvido, momento no qual se verificou a correspondência da interface com o *design* elaborado. Por fim, aprovou-se a implantação da aplicação em ambiente de produção.

3.1.2 Interface de validação de usuário

O segundo item desenvolvido foi a interface de validação de usuário, onde se exibe não somente os componentes já definidos, como também os novos componentes `Codigo`, responsável pelo recebimento do código secreto de quatro dígitos a ser inserido pelo usuário, e `Botao`, usado para o envio do código inserido.

O componente `Codigo` é composto por quatro entradas de texto respectivas aos dígitos do código secreto. Por serem dados que se alteram durante o ciclo de vida do componente, esses caracteres são incluídos no seu conjunto de estado. Como se pode observar na Figura 3.4, além dos elementos de entrada de texto, o componente `Codigo` também se responsabiliza pela exibição de uma mensagem de erro quando um código inválido for enviado.

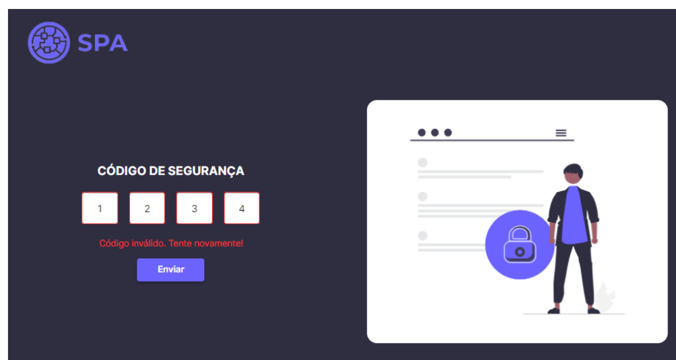


Figura 3.4 – Interface de validação de usuário da aplicação web ao inserir código inválido.
Fonte: Do autor (2022).

Um ponto que vale ser ressaltado é que, embora a biblioteca React tenha sido adotada pelo cliente há um tempo considerável, ainda não havia pacotes de componentes previamente criados e padronizados em um *Design System*[3] para permitir a reutilização, exigindo um maior esforço para o desenvolvimento completo de cada componente. Contudo, a equipe de *design* elaborou as interfaces se baseando em componentes fornecidos pela biblioteca MUI[4]. Elaborada para o React, o MUI fornece inúmeros componentes prontos, funcionais e customizáveis, possibilitando maior agilidade durante o desenvolvimento das interfaces. O Código 3.1 é um exemplo de utilização e customização de um componente do MUI, também aplicados ao componente `Botao` presente nas interfaces implementadas.

```
import { Button } from "@mui/material";

const Interface = () => {
  return (
    <Button
      color="primary"
      variant="contained"
      sx={{
        textTransform: "capitalize",
        fontWeight: 600,
        width: { xs: 300, md: 107 },
        height: 36
      }}
    >
      Enviar
    </Button>
  )
}
```

Código 3.1 – Exemplo de utilização de um componente do MUI.

A criação de uma interface em React se torna algo ainda mais rápido de se desenvolver ao utilizar-se de componentes do MUI. Como se pode observar no Código 3.1, o MUI fornece o componente `Button` para criação de botões de fácil customização por meio de propriedades

[3] *Design System* representa um conjunto de diretrizes que orienta a forma de reutilizar os componentes visuais, suas funções e interações em um sistema (SMITH, 2021).

[4] Disponível em <https://mui.com/>.

predefinidas, conforme a documentação da biblioteca. De início, importa-se o componente `Button` (linha 1) que deve ser adicionado à interface que está sendo desenvolvida (linha 5). Para customizá-lo, utiliza-se de três de suas propriedades: a propriedade **`color`** (linha 6), que define qual segmento de cor deve ser usado; a propriedade **`variant`** (linha 7), que define de que modo a cor deve ser disposta no botão; e a propriedade **`sx`** (linha 8), que permite a utilização direta de recursos da linguagem CSS por meio de um objeto da linguagem JavaScript.

Concluído o desenvolvimento da nova interface, realizou-se a revisão do código e a implantação da aplicação em ambiente de homologação, para a realização de testes da nova interface. Durante a validação, a equipe de *design* relatou um comportamento inesperado do componente `Codigo`. Esperava-se que, a cada dígito inserido, o componente automaticamente selecionasse o campo de texto seguinte, garantindo uma melhor experiência para o usuário. Por conta disso, retornou-se ao desenvolvimento para implementar o comportamento. Após a correção, a aplicação foi aprovada para ser implantada em produção e o novo incremento foi entregue com sucesso.

3.1.3 Considerações finais

As atividades referentes ao desenvolvimento do SPA contribuíram significativamente para a aquisição de experiência prática de importantes conceitos relacionados ao funcionamento de aplicações web modernas, principalmente no emprego da biblioteca React.

Ademais, as atividades relacionadas ao desenvolvimento das interfaces iniciais também envolveram a execução dos primeiros procedimentos de criação do SPA. Diante disso, o estagiário pôde participar e contribuir com o processo de configuração inicial da aplicação, o qual incluiu a criação de um novo repositório, a inicialização do projeto e a preparação para tornar a aplicação reconhecida pelo *pipeline* de CI e CD.

Durante o processo inicial de configuração do SPA, também se comunicou com os integrantes de outras equipes que possuíam experiência com a utilização da ferramenta React no contexto em que se atuava. Assim, em conjunto com as informações obtidas de várias fontes, o processo de configuração inicial foi realizado apropriadamente e foi possível adquirir conhecimento prático quanto à adequação aos padrões de execução das práticas de CI e CD no contexto da DTI e cliente.

A atuação do estagiário na primeira equipe foi breve e teve-se a oportunidade de realizar a entrega de apenas duas interfaces do SPA. No entanto, as atividades realizadas contribuíram

bastante para a rápida criação de posteriores interfaces por parte da equipe, tendo em vista que foram componentes fundamentais, portanto, deveriam ser utilizados em todas as interfaces da aplicação.

Durante as sessões de revisão de código, o estagiário também teve a oportunidade de contribuir com a realização de repasses de conhecimentos à equipe, repassando as informações sobre o funcionamento da aplicação e suas configurações, e as informações adquiridas durante a comunicação com os integrantes de outras equipes.

Constatou-se também que as ferramentas utilizadas auxiliaram bastante e foram consoantes com o contexto de desenvolvimento ágil de software, pois permitiram entregas breves que geraram valor para o cliente.

3.2 Criação de funcionalidades para microsserviços

Segundo Thönes (2015), um microsserviço é uma pequena aplicação de software de responsabilidade única que pode ser implantada, escalada e testada de forma independente. Uma arquitetura de microsserviços é uma aplicação distribuída constituída por um conjunto de microsserviços, considerados módulos independentes da aplicação (DRAGONI et al., 2017). Essa arquitetura, representada na Figura 3.5, também conta com um serviço centralizador, denominado API *Gateway*, que se comporta como uma fronteira responsável por redirecionar as requisições recebidas aos microsserviços correspondentes, simplificando a comunicação e integração com interfaces externas (ZHAO; JING; JIANG, 2018).

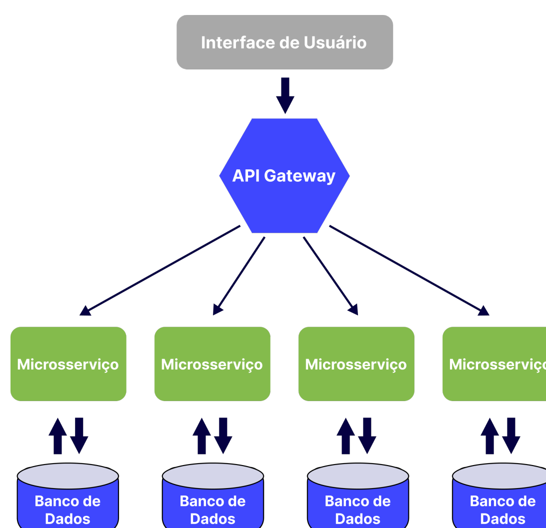


Figura 3.5 – Representação da arquitetura de microsserviços.
Fonte: Adaptado de Fybish (2021).

Dentre as atividades realizadas durante a atuação do estagiário na segunda equipe, destacou-se a criação de uma nova funcionalidade de validação e outra para a realização de operações de *Create*, *Read*, *Update* e *Delete* (CRUD) em informações de entidades relacionadas ao negócio do cliente. As operações de CRUD são funções implementadas para a inserção, leitura, atualização e remoção de registros, proporcionando a manipulação de dados normalmente persistidos em um banco de dados (SULEMANI, 2021). As tecnologias utilizadas para as implementações foram o *framework* ASP.NET Core[5] e a linguagem C#.

3.2.1 Funcionalidade de validação

Um dos microsserviços em que se atuou tinha como domínio realizar um conjunto de validações em produtos conforme as necessidades de negócio do cliente. Devido às novas exigências, necessitou-se adicionar uma funcionalidade para realizar uma nova validação. Além disso, deveria ser possível desabilitar o seu uso por meio de um sistema integrado aos microsserviços. Tal condição era necessária para possibilitar rápida alteração em caso de obsolescência da nova validação ou ocorrência de erros. Como enfatiza Hodgson (2017), essa técnica é nomeada *Feature Toggle* e consiste em fornecer meios para se alterar o comportamento de um sistema sem efetuar mudanças no código. Por conta disso, é uma técnica bastante útil em situações inesperadas após a integração de funcionalidades.

Conforme o fluxo inicial de desenvolvimento ágil dado pela realização dos ritos do Scrum, evidenciado na Figura 3.6, a equipe inseriu um novo item no *Product Backlog* para iniciar o desenvolvimento da nova funcionalidade. Em seguida, foi realizado o seu processo de refinamento, atentando-se em descrevê-lo conforme orientado pela técnica BDD.

Para demonstrar o processo e as práticas de desenvolvimento exercidos durante a atividade realizada, assume-se um exemplo simplificado da funcionalidade de validação, baseado em um contexto onde há uma entidade de produto e uma entidade de item, relacionados de modo que um conjunto de itens possa constituir um produto. Adaptado a esse contexto, o item referente a nova funcionalidade, inserido pela equipe no *Product Backlog*, é descrito na Tabela 3.1.

Como se pode observar, o modelo de história de usuário foi utilizado para descrever o objetivo da funcionalidade e para facilitar a identificação do valor gerado após a sua entrega.

---

[5] Disponível em <https://learn.microsoft.com/pt-br/aspnet/core/introduction-to-aspnet-core>.

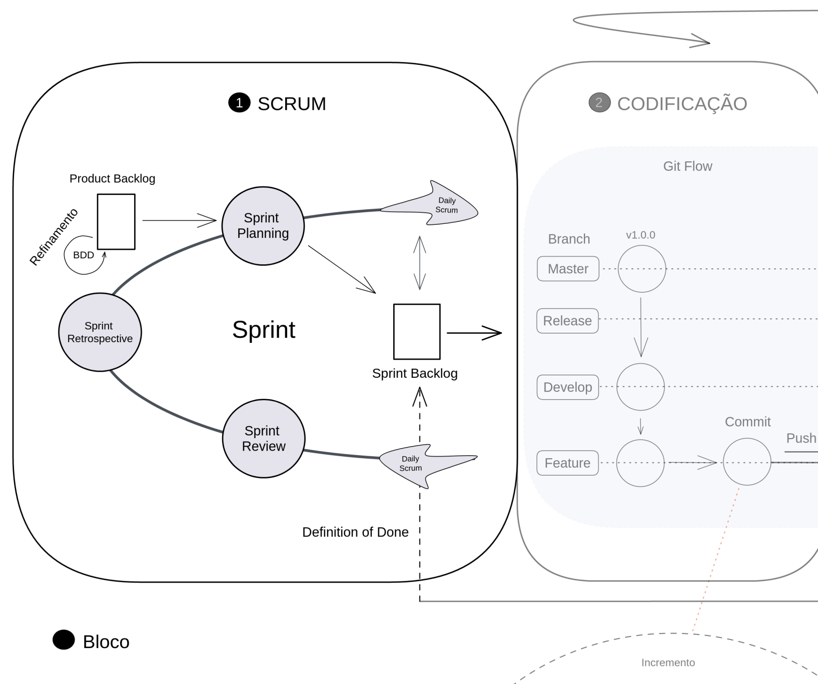


Figura 3.6 – Primeira etapa (Scrum) do desenvolvimento ágil.
Fonte: Do autor (2022).

Tabela 3.1 – Descrição do item referente à funcionalidade de validação.

Funcionalidade	Descrição
Validação de quantidade total de itens de um produto.	**Como um** usuário do sistema de validações automáticas; **Quero** a existência de uma validação automática que verifique a soma do total da quantidade de cada item de um produto; **Para** impedir ou não o cadastro de um produto.
Cenários	Descrição
Cenário 1: Validação desabilitada.	**Dada** a obtenção do parâmetro de controle da validação; **Quando** valor do parâmetro é falso; **Então** encerrar a execução da validação.
Cenário 2: Quantidade total de itens maior que a permitida.	**Dada** a soma do total da quantidade de itens; **Quando** maior que 300; **Então** retornar uma mensagem de erro informando que a quantidade de itens do produto é maior que o limite predefinido.
Cenário 3: Quantidade total de itens menor ou igual que a permitida.	**Dada** a soma do total da quantidade de itens; **Quando** menor ou igual a 300; **Então** não realizar nenhuma ação.

Ademais, como complemento para evidenciar os critérios de aceitação do item, foram descritos cenários indicando os comportamentos esperados da funcionalidade.

Com os requisitos descritos detalhadamente e também esclarecidos durante o rito *Sprint Planning*, prosseguiu-se para a etapa de Codificação para iniciar a criação do código da nova

solução. Conforme o fluxo inicial dessa etapa, foram executados os procedimentos iniciais do modelo *Git Flow*, criando-se uma *branch feature* para realizar o desenvolvimento.

O desenvolvimento transcorreu atrelado a aplicação da prática TDD, orientando-se pelos cenários previamente descritos. Para a implementação dos testes, utilizou-se do xUnit[6], ferramenta de testes de unidade para ASP.NET Core, e da biblioteca Moq[7], usada para a criação de objetos simulados definidos com valores fictícios para serem manipulados durante a execução dos testes.

Considerando a primeira etapa do ciclo do TDD, adicionou-se um novo teste de unidade para certificar que a nova funcionalidade de validação não prossiga a sua execução quando estiver desabilitada por meio do parâmetro de controle, conforme requisitado no Cenário 1.

Exemplificada no Código 3.2, a implementação do teste se deu com a definição de um método na classe `TesteValidacaoQuantidadeTotalItens`, a qual foi exclusivamente criada para manter os testes referentes a nova funcionalidade de validação. Como atributo dessa classe, define-se o objeto `_centralControleParametrosMock` (linha 3) para simular o serviço de obtenção de parâmetros de controle, dada a necessidade de sua utilização durante a execução da funcionalidade. Em seguida, define-se o método `SeValidacaoDesabilitada_EntaoEncerrarExecucao` (linha 11), onde consta a lógica do teste. O teste foi estruturado conforme o padrão *Arrange-Act-Assert* (AAA) que, segundo Gomes (2017), define seções que delimitam os estágios do teste:

- *Arrange*: preparação dos objetos necessários para a execução do método a ser testado.
- *Act*: execução do método que deve ser testado.
- *Assert*: ratificação de que o resultado e o comportamento correspondem com os esperados.

```
1  public class TesteValidacaoQuantidadeTotalItens
2  {
3      public Mock<ICentralControleParametros> _centralControleParametrosMock;
4
5      public TesteValidacaoQuantidadeTotalItens()
6      {
7          _centralControleParametrosMock = new Mock<ICentralControleParametros>();
8      }
9
10     [Fact]
11     public async Task SeValidacaoDesabilitada_EntaoEncerrarExecucao()
12     {
```

[6] Disponível em <https://xunit.net/>.
[7] Disponível em <https://github.com/Moq/moq4>.

```
13         //Arrange
14         var validacaoQuantidadeTotalItens = new ValidacaoQuantidadeTotalItens(
               _centralControleParametrosMock.Object);
15         IEnumerable<string> mensagensValidacao = new List<string>();
16
17         _centralControleParametrosMock
               .Setup(m => m.ValidacaoQuantidadeTotalItensHabilitada())
               .ReturnsAsync(false);
18
19         //Act
20         bool resultado = await validacaoQuantidadeTotalItens
               .Executar(It.IsAny<IEnumerable<Item>>(), mensagensValidacao);
21
22         //Assert
23         _centralControleParametrosMock
               .Verify(m => m.ValidacaoQuantidadeTotalItensHabilitada(), Times.Once);
24         _centralControleParametrosMock.VerifyNoOtherCalls();
25         Assert.False(resultado);
26         Assert.Empty(mensagensValidacao);
27     }
28 }
```

Código 3.2 – Primeiro teste de unidade da funcionalidade de validação.

Inicialmente, na seção *Arrange* do método criado, instancia-se o objeto `validacaoQuantidadeTotalItens` (linha 14), responsável por executar a validação da quantidade total de itens, e define-se uma lista vazia (linha 15) para conter as possíveis mensagens de validação após a execução. O objeto `validacaoQuantidadeTotalItens` recebe como parâmetro o objeto simulado `_centralControleParametrosMock`, configurado-o (linha 17) de modo que, quando o seu método `ValidacaoQuantidadeTotalItensHabilitada` for acionado, seja retornado **`falso`** como o valor do parâmetro de controle, indicando que a funcionalidade de validação está desabilitada. Em seguida, na seção *Act*, executa-se a nova funcionalidade de validação (linha 20), enviando como parâmetro a lista de itens e a lista vazia referente as mensagens de validação. Por fim, na seção *Assert*, verifica-se se houve o acionamento apenas do método `ValidacaoQuantidadeTotalItensHabilitada` (linhas 23 e 24). Além disso, tendo em vista a condição de desabilitação da validação, o valor resultante retornado após a sua execução deve ser **`falso`** (linha 25) e a sua lista de mensagens de validação deve permanecer vazia (linha 26).

Concluída a implementação, prosseguiu-se com a prática de TDD, i.e., a execução e, como previsto, a posterior falha do teste implementado. Então, iniciou-se a criação da primeira versão do código da nova funcionalidade para se adequar e satisfazer o teste adicionado.

Como demonstra o Código 3.3, é realizada uma requisição ao sistema central para consultar o valor do parâmetro de controle da validação (linha 12). Se o valor do parâmetro for **`falso`**, a validação encerra a sua execução retornando esse valor, adequando-se ao comportamento esperado definido no teste.

```
public class ValidacaoQuantidadeTotalItens
{
  private readonly ICentralControleParametros _centralControleParametros;

  public ValidacaoQuantidadeTotalItens(ICentralControleParametros centralControleParametros)
  {
    _centralControleParametros = centralControleParametros;
  }

  public Task<bool?> Executar(IEnumerable<Item> itens)
  {
    return _centralControleParametros.ValidacaoQuantidadeTotalItensHabilitada();
  }
}
```

Código 3.3 – Implementação inicial da funcionalidade de validação.

Em seguida, iniciou-se a implementação do segundo teste de unidade para a funcionalidade, buscando satisfazer o Cenário 2. Como se pode observar no Código 3.4, um novo método foi adicionado na classe `TesteValidacaoQuantidadeTotalItens`, nomeado de `SeQuantidadeTotalDeItensExcedida_EntaoInserirMensagemDeErro`. Na implementação, além da redefinição do objeto `validacaoQuantidadeTotalItens` (linha 5) e da lista vazia de mensagens de validação (linha 6), também se define uma lista de itens com valores fictícios (linha 7). Vale ressaltar que, no contexto de exemplo considerado, o objeto `Item` é composto por descrição, quantidade e valor unitário. Considerando as circunstâncias impostas para o teste, define-se valores de modo que o somatório da quantidade de cada item resulte em um valor acima do valor definido no Cenário 2. Ainda na seção *Arrange* do teste, o objeto fictício `_centralControleParametrosMock` é novamente configurado, mas de modo que seu método `ValidacaoQuantidadeTotalItensHabilitada` retorne valor **`verdadeiro`** (linha 9), indicando que a validação está habilitada. Concluída a preparação, tem-se a execução da validação com o envio das listas de itens e de mensagens de validação como parâmetros (linha 12). Em seguida, verifica-se se a funcionalidade obteve o comportamento esperado após a sua execução, o qual se confirma pela inserção correta da mensagem de erro por parte da nova validação (linha 18).

```
[Fact]
public async Task SeQuantidadeTotalDeItensExcedida_EntaoInserirMensagemDeErro()
{
    //Arrange
    var validacaoQuantidadeTotalItens = new ValidacaoQuantidadeTotalItens(
        _centralControleParametrosMock.Object);
    IEnumerable<string> mensagensValidacao = new List<string>();
    IEnumerable<Item> itensMock = new List<Item>
    {
        new Item { Id = 1, Descricao = "Item 1", Quantidade = 150, Valor = 500 },
        new Item { Id = 2, Descricao = "Item 2", Quantidade = 120, Valor = 650 },
        new Item { Id = 3, Descricao = "Item 3", Quantidade = 100, Valor = 750 }
    };

    _centralControleParametrosMock
        .Setup(m => m.ValidacaoQuantidadeTotalItensHabilitada())
        .ReturnsAsync(true);
```

```
10
11     //Act
12     bool resultado = await validacaoQuantidadeTotalItens
           .Executar(itensMock, mensagensValidacao);
13
14     //Assert
15     _centralControleParametrosMock
           .Verify(m => m.ValidacaoQuantidadeTotalItensHabilitada(), Times.Once);
16     _centralControleParametrosMock.VerifyNoOtherCalls();
17     Assert.True(resultado);
18     Assert.All(mensagensValidacao, mensagem =>
           Assert.Equal("A quantidade total de itens do produto foi excedida", mensagem));
19 }
```

Código 3.4 – Segundo teste de unidade da funcionalidade de validação.

Concluída a criação do teste, prosseguiu-se com o desenvolvimento da funcionalidade. Como se demonstra no Código 3.5, após a verificação do parâmetro de controle (linha 12), realiza-se o somatório da quantidade de todos os itens (linha 14). Em seguida, tem-se a condição para verificar se o somatório da quantidade de itens é maior que o limite predefinido (linha 15). Se a condição for verdadeira, insere-se a mensagem de erro na lista de mensagens de validação recebida como parâmetro (linha 16). Por fim, um valor **`verdadeiro`** é retornado indicando a correta execução da validação (linha 17).

```
1  public class ValidacaoQuantidadeTotalItens
2  {
3    private readonly ICentralControleParametros _centralControleParametros;
4
5    public ValidacaoQuantidadeTotalItens(ICentralControleParametros centralControleParametros)
6    {
7      _centralControleParametros = centralControleParametros;
8    }
9
10   public async Task<bool> Executar(IEnumerable<Item> itens, IEnumerable<string>
       mensagensValidacao)
11   {
12     if (!await _centralControleParametros.ValidacaoQuantidadeTotalItensHabilitada())
13       return false;
14     int somaQuantidadeTotalItens = itens.Sum(item => item.Quantidade);
15     if (somaQuantidadeTotalItens > LIMITE_QUANTIDADE_ITENS)
16       mensagensValidacao.Append("A quantidade total de produtos foi excedida");
17     return true;
18   }
19 }
```

Código 3.5 – Implementação final da funcionalidade de validação.

Adicionado o novo código na implementação da funcionalidade, atingiu-se o comportamento esperado. Porém, ainda foram adicionados novos testes para certificar que todos os fluxos possíveis foram tratados, por exemplo, em caso de erro ao obter o parâmetro de controle ou em casos de quantidade total de itens específica.

Após o envio das alterações para o repositório remoto do microsserviço, iniciou-se a prática de revisão de código efetuada na etapa de Integração Contínua, destacada na Figura 3.7. Durante a revisão, a equipe constatou a necessidade de se realizar uma refatoração devido a uma duplicidade de código evidenciada durante o processo de integração. A duplicidade se deu

por conta da inserção do trecho de código criado para realizar cálculos específicos que, no real contexto, também estava sendo realizado por outra funcionalidade de validação da aplicação. Para corrigir, criou-se uma nova classe com um método para receber o código extraído e, nos locais de extração, colocou-se a chamada ao método criado.

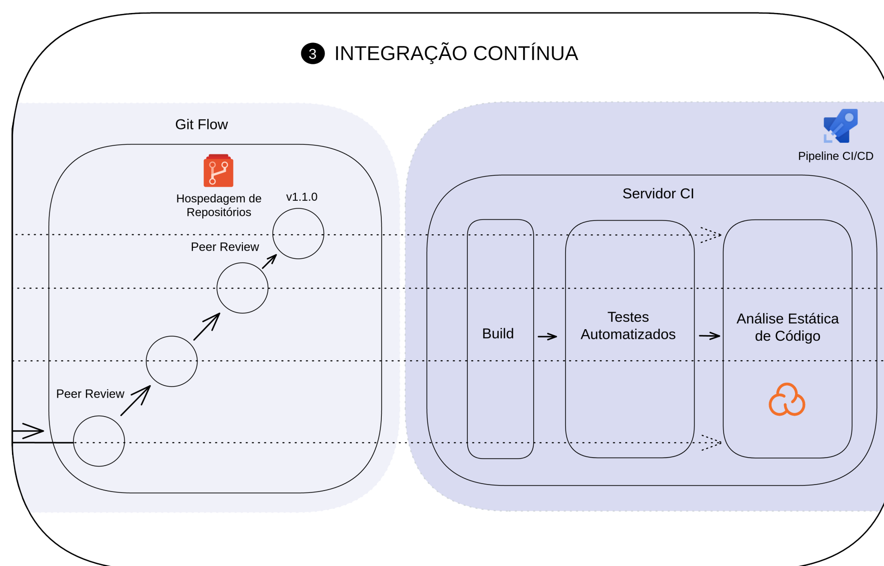


Figura 3.7 – Terceira etapa (Integração Contínua) do desenvolvimento ágil.
Fonte: Do autor (2022).

O próximo passo após a aprovação e integração das novas alterações, foi a implantação da aplicação incrementada em ambiente de homologação na etapa de Entrega Contínua. Contudo, para a realização dos testes manuais referentes a nova funcionalidade durante essa etapa, foi necessário cadastrar o novo parâmetro de controle na interface do sistema central em seu ambiente de homologação. Com o parâmetro criado, elaborou-se um roteiro de testes manuais para orientar a equipe durante a execução dos testes. Após executá-los com sucesso, cadastrou-se o parâmetro de controle em ambiente de produção e, por fim, implantou-se o microsserviço com a nova funcionalidade em ambiente de produção.

Vale ressaltar que, apesar da possibilidade de implantação em ambiente de desenvolvimento, como consta na Figura 3.8, não foi necessária a sua utilização. Tal ambiente fornece maior segurança ao realizar testes de funcionalidades ou alterações potencialmente impactantes que, se realizados em um ambiente de homologação utilizado por diversas equipes relacionadas, poderiam prejudicar a base de dados, tornando o ambiente menos confiável para a realização dos testes finais de aceitação. Como a funcionalidade desenvolvida não se submetia a essa condição, a equipe decidiu que seria mais proveitoso e ágil realizar a validação diretamente em

ambiente de homologação, dado o baixo impacto da nova funcionalidade nas bases de dados compartilhadas.

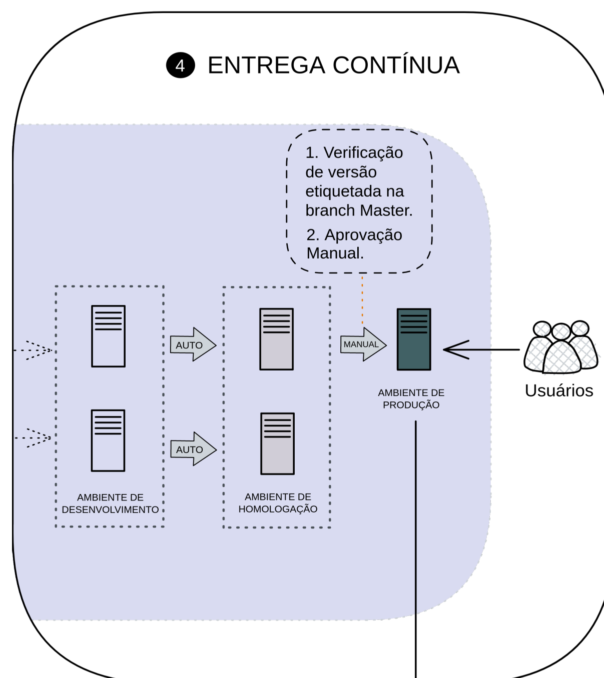


Figura 3.8 – Quarta etapa (Entrega Contínua) do desenvolvimento ágil.
Fonte: Do autor (2022).

Após a implantação em ambiente de produção, manteve-se um acompanhamento mais próximo da aplicação em fluxos e alterações de dados relacionados a nova funcionalidade, buscando a rápida identificação de possíveis comportamentos inesperados. Com a realização do acompanhamento, constatou-se que tudo se comportou conforme o esperado e a entrega do item foi bem sucedida.

3.2.2 Funcionalidade de operações de CRUD

Outra atividade realizada durante o estágio foi a criação de uma funcionalidade para possibilitar a realização de operações de CRUD em entidades relacionadas ao negócio do cliente. A funcionalidade integraria o conjunto de principais funcionalidades que deveria compor a nova solução, dada pela criação e operação de um novo microsserviço.

Para realizar a atividade de implementação da nova funcionalidade, foi preciso compreender o *Onion Architecture*, padrão arquitetural no qual o microsserviço baseou sua estrutura. Esse passo foi crucial para o correto desenvolvimento da funcionalidade, mantendo-a em conformidade quanto às relações entre os componentes arquiteturais da aplicação.

De acordo com Palermo (2008), as arquiteturas tradicionais frequentemente adotadas para construção de aplicações acarretam uma forte dependência entre os componentes de regras de negócio e os recursos de infraestrutura, como os sistemas de banco de dados, tornando-se uma desvantagem para os sistemas potencialmente grandes. Diante disso, o *Onion Architecture* é proposto visando fornecer um maior controle do grau de dependência entre os componentes. Para isso, define-se um *design* específico organizado em camadas e subcamadas constituídas pelos componentes da aplicação. Nessa estrutura, representada na Figura 3.9, os componentes de camadas externas possuem dependência com os componentes de camadas mais internas, o contrário, porém, é restrito (PALERMO, 2008).

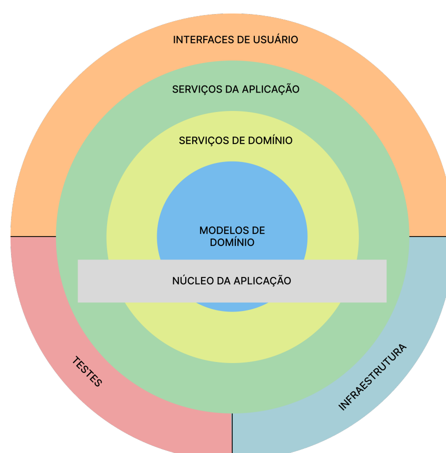


Figura 3.9 – Representação do padrão arquitetural *Onion Architecture*.
Fonte: Adaptado de Ziemoński (2017).

Como demonstra a representação, o centro da estrutura é considerado o Núcleo da Aplicação, contendo as camadas Serviços da Aplicação, Serviços de Domínio e Modelos de Domínio. Por possuírem os componentes responsáveis pelas entidades e pelas lógicas de regras de negócio, essas camadas devem ocupar a região central para obter maior independência. Como resultado, tem-se o isolamento de tais componentes de modo que, ao realizar alterações em camadas mais externas, por exemplo, de Infraestrutura ou de Interfaces de Usuário, não impacte as camadas de domínio centrais.

O processo de desenvolvimento da nova funcionalidade é demonstrado considerando a realização de operações de CRUD na entidade de exemplo `Produto`, composto por identificador, nome e descrição. Considerando esse contexto, a funcionalidade desenvolvida deve satisfazer os critérios definidos na Tabela 3.2. Observa-se que os critérios que devem ser satisfeitos pela funcionalidade são dispostos em cenários correspondentes às operações de CRUD.

Tabela 3.2 – Descrição da funcionalidade de operações de CRUD de produto.

Funcionalidade	Descrição
Operações de CRUD de produtos.	**Como um** usuário do sistema mantenedor de produtos; **Quero** que existam meios para cadastrar, listar, atualizar e remover produtos; **Para** facilitar e tornar eficiente o controle de produtos existentes.
Cenários	Descrição
Cenário 1: Cadastro de produto.	**Dado** o recebimento das informações de nome e a descrição de um produto; **Quando** preenchidas com valores válidos; **Então** inseri-las na base de dados gerando um identificador único de novo produto.
Cenário 2: Listagem de produtos.	**Dada** uma requisição para a listagem de informações de produtos; **Quando** verificada o preenchimento de filtragem por identificador, nome ou descrição com valores válidos; **Então** retornar todos os produtos correspondentes ao conjunto de parâmetros preenchido.
Cenário 3: Atualização de produto.	**Dado** o recebimento do identificador de um produto e pelo menos uma de suas informações restantes; **Quando** preenchidas com valores válidos; **Então** atualizá-las na base de dados.
Cenário 4: Remoção de produto.	**Dado** o recebimento do identificador de um produto; **Quando** preenchido com valores válidos; **Então** remover todas as informações do produto correspondente na base de dados.

Conforme o fluxo de desenvolvimento ágil, o item foi catalogado e detalhado no *Product Backlog* pela equipe e, após a reunião de planejamento, iniciaram-se as tarefas para desenvolvê-lo. O primeiro passo realizado pelo estagiário se deu com a compreensão e a definição dos componentes que deveriam ser implementados para compor a funcionalidade, atentando-se à arquitetura do microsserviço e aos requisitos definidos. Na Figura 3.10, representa-se, de forma simplificada, as principais camadas da arquitetura junto aos componentes relacionados com a nova funcionalidade.

Nesse contexto da funcionalidade de operações de CRUD para produtos, o componente `ControladorProduto` é o responsável por receber e retornar informações de produtos às interfaces de usuário conforme a operação de CRUD solicitada. Por conta disso, situa-se na camada externa Interfaces de Usuário. Após o recebimento, o controlador transmite as informações para as camadas mais internas, para serem processadas segundo as regras de negócio da aplicação.

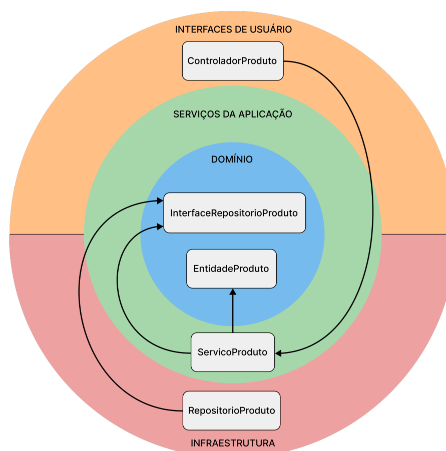


Figura 3.10 – Representação da arquitetura do microsserviço.
Fonte: Do autor (2022).

Como demonstra o fluxo da representação, o `ServicoProduto`, localizado na camada Serviços da Aplicação, é o componente de serviço acionado pelo controlador para realizar a validação das informações recebidas e executar possíveis regras com base no domínio da aplicação. Para isso, o serviço manipula a entidade de produto, dada pelo componente `EntidadeProduto`, e acessa o repositório de produto para a obtenção ou persistência de suas informações no banco de dados da aplicação.

No entanto, para contornar o surgimento de uma violação arquitetural no microsserviço, que ocorreria pelo acesso restrito do componente `ServicoProduto` ao componente `RepositorioProduto`, define-se o componente `InterfaceRepositorioProduto` na camada de Domínio, para interpor e conformar o acesso do `ServicoProduto` ao repositório de produto. Por conseguinte, o `RepositorioProduto`, responsável pelo acesso ao banco de dados, implementa a interface definida.

Vale ressaltar que, para o funcionamento adequado da comunicação entre os componentes definidos, utiliza-se de recursos fornecidos pelo *framework* ASP.NET Core que auxiliam na respectiva configuração.

Compreendido como a nova funcionalidade deveria ser implementada na aplicação, prosseguiu-se com o seu desenvolvimento em concordância com a prática TDD. Inicialmente, além da criação de uma classe para o componente `EntidadeProduto`, também foi necessária as definições de `ModeloProduto` e de `DadosProduto` para possibilitar a manipulação de informações de produto nas camadas de Interfaces de Usuário e de Infraestrutura, respectivamente, visto que a entidade de produto é acessada somente pelo `ServicoProduto`.

Em seguida, foram criadas as demais classes ControladorProduto, ServicoProduto e RepositorioProduto, e a interface IRepositorioProduto, nas quais são definidos os métodos InserirProduto, ListarProdutos, AtualizarProduto e RemoverProduto, implementados conforme a responsabilidade de cada classe em sua respectiva camada. O Código 3.6 exibe um exemplo de implementação do cadastro de produtos na classe ServicoProduto.

```
public class ServicoProduto
{
    private readonly IRepositorioProduto _repositorioProduto;

    public ServicoProduto(IRepositorioProduto repositorioProduto)
    {
        _repositorioProduto = repositorioProduto;
    }

    public ResultadoServico<ModeloProduto> InserirProduto(ModeloProduto modeloProduto)
    {
        Produto produto = new Produto(modeloProduto.Nome, modeloProduto.Descricao);
        if (!produto.InformacoesValidas)
            return ResultadoServico<ModeloProduto>.Erro(produto.MensagensValidacoes);
        Produto produtoInserido = _repositorioProduto.InserirProduto(produto);
        return ResultadoServico<ModeloProduto>.Ok(new ModeloProduto {
            Id = produtoInserido.Id,
            Nome = produtoInserido.Nome,
            Descricao = produtoInserido.Descricao
        });
    }
}
```

Código 3.6 – Implementação do cadastro de produto no componente de serviço.

Como demonstrado na representação arquitetural, o componente de serviço aciona o repositório e é acionado pelo controlador. Diante disso, tem-se a definição do atributo _repositorioProduto (linha 3) para possibilitar o acesso ao repositório de produto durante a execução de uma operação. Então, segue-se com a definição do método InserirProduto (linha 10), responsável pela operação de cadastro de produto. O método recebe como parâmetro as informações do produto por meio do ModeloProduto procedente do ControladorProduto. Em seguida, instancia-se a entidade Produto (linha 12), que realiza a validação das informações internamente. Caso as informações sejam inválidas, o serviço retorna mensagens de erro ao controlador (linha 14). Se válidas, o serviço aciona o método InserirProduto do repositório enviando a instância de produto (linha 15). Por fim, após o retorno da entidade inserida pelo repositório, realiza-se um mapeamento da entidade para o ModeloProduto e o retorna ao controlador, indicando o sucesso da operação (linha 16).

Outra implementação importante ser mencionada é a operação de listagem de produtos. Como se pode observar no Código 3.7, ainda no ServicoProduto, o novo método ListarProdutos foi adicionado, o qual recebe como parâmetros opcionais o identificador, o nome ou a descrição de um produto.

```
public ResultadoServico<IEnumerable<ModeloProduto>> ListarProdutos(int? id, string nome,
     string descricao)
{
    IEnumerable<Produto> produtos = _repositorioProduto.ListarProdutos();

    if (id.HasValue)
        produtos = produtos.Where(produto => produto.Id == id);
    else
    {
        if (!string.IsNullOrEmpty(nome))
            produtos = produtos.Where(produto => produto.Nome.Contains(nome));
        if (!string.IsNullOrEmpty(descricao))
            produtos = produtos.Where(produto => produto.Descricao.Contains(descricao));
    }

    List<ModeloProduto> modeloProdutos = new List<ModeloProduto>();

    foreach (var produto in produtos)
        modeloProdutos.Add(new ModeloProduto
        {
            Id = produto.Id,
            Nome = produto.Nome,
            Descricao = produto.Descricao
        });

    return ResultadoServico<IEnumerable<ModeloProduto>>.Ok(modeloProdutos);
}
```

Código 3.7 – Implementação da listagem de produtos no componente de serviço.

A princípio, aciona-se o método `ListarProdutos` do repositório para obter todos os produtos cadastrados na base de dados (linha 3). Em seguida, conforme descrito nos requisitos dessa operação, realiza-se a filtragem de produtos, em que é obtido o produto correspondente ao identificador fornecido (linha 6), ou são obtidos os produtos com nome e descrição correspondentes aos textos de busca recebidos nos respectivos parâmetros do método (linhas 10 e 12). Por fim, os produtos são retornados ao controlador após a realização do mapeamento da lista de Produto para a lista de `ModeloProduto` (linhas 17 a 20).

Um ponto importante de se esclarecer é o motivo da realização da filtragem de produtos no nível de memória, em vez da utilização de recursos de filtragem do banco de dados. No real contexto, o repositório da entidade estava configurado para usufruir de um recurso de *cache* em memória e com armazenamento de longo prazo, fornecido por um banco de dados especializado. Assim, ciente da quantidade de dados manipulada, do fato de que os dados já estariam em memória e que se atingiria um melhor desempenho ao listá-los, decidiu-se pela realização da filtragem no nível de memória.

Outra observação é que a comunicação com o microsserviço se dava por meio de requisições *HyperText Transfer Protocol* (HTTP) provindas de uma interface web. Sendo assim, o controlador foi configurado para tal contexto, provendo recursos para o acionamento e a execução das operações de CRUD implementadas. Já a interface web era fornecida pela ferramenta

Swagger[8] em conjunto com o pacote Swashbuckle[9] para ASP.NET Core, que possibilitam a documentação de recursos da aplicação e a disponibilização de cliente HTTP para acioná-los mediante requisições.

O desenvolvimento da nova funcionalidade prosseguiu com a implementação das operações de atualização e remoção. Após sua conclusão, iniciou-se o processo de revisão, as ações de integração, a implantação em ambiente de homologação e os testes manuais. Os testes manuais foram realizados com a inserção de entradas específicas por meio da interface web, buscando assim validar o comportamento da funcionalidade em todos os cenários possíveis. A Figura 3.11 exemplifica a realização de um teste para validar a operação de listagem de produtos, previamente cadastrados na base de dados.

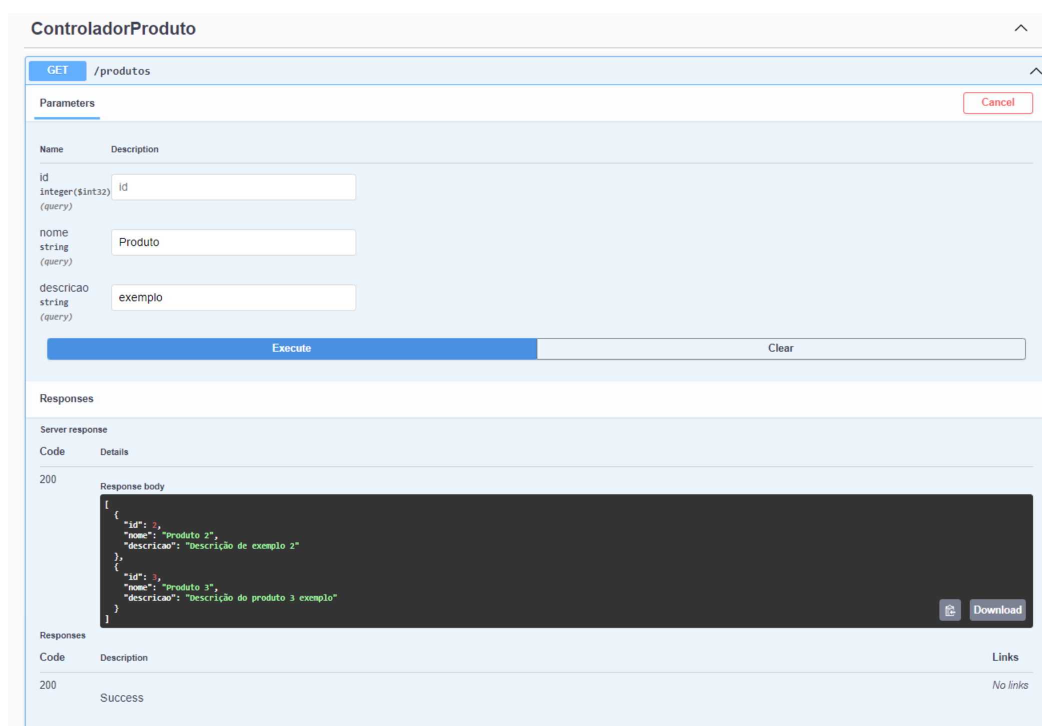


Figura 3.11 – Teste de listagem de produtos por meio da interface fornecida pelo Swagger.
Fonte: Do autor (2022).

O objetivo do teste era confirmar se a filtragem na listagem de produtos estava conforme as expectativas. Para isso, o parâmetro “nome” foi definido como “Produto” e “descricao” como “exemplo”, para que fossem retornados todos os produtos correspondentes. Como resultado, foram listados todos os produtos que possuíam “Produto” em seu nome e “exemplo” em sua descrição.

[8] Disponível em <https://swagger.io/>.
[9] Disponível em <https://github.com/domaindrivendev/Swashbuckle.AspNetCore>.

Após a conclusão dos testes efetuados pela equipe, a aplicação incrementada com a nova funcionalidade foi implantada em ambiente de produção, iniciando a etapa de monitoramento e concluindo o fluxo de desenvolvimento ágil.

3.2.3 Considerações finais

A realização das atividades descritas nesta seção proporcionou ao estagiário um contato ainda mais próximo com as práticas de desenvolvimento ágil de software no contexto da DTI e cliente. Usufruiu-se bastante da oportunidade de vivenciar o fluxo com maior abrangência e de lidar com novos desafios circunstanciais referentes a cada etapa de desenvolvimento. Um exemplo disso foi a realização de refatorações de código, após constatada a sua necessidade durante o processo de revisão de código na etapa de Integração Contínua.

Durante as atividades, também se adquiriu novas experiências práticas com o emprego de *Feature Toggle* e TDD, importantes técnicas adotadas durante o desenvolvimento ágil. Além disso, o estagiário esteve inserido no processo de desenvolvimento de um novo microsserviço, solução bastante almejada em virtude de seu objetivo de padronização e centralização de informações. A criação da funcionalidade de CRUD foi de grande importância, pois se tratava de um requisito obrigatório para permitir a satisfatória operação do microsserviço e a manipulação dos dados de seu domínio.

Obteve-se também um importante conhecimento em relação a padrões arquiteturais, visto que foi necessário o estudo da arquitetura de uma das aplicações para o correto desenvolvimento de sua nova funcionalidade.

4 CONCLUSÃO

Durante o curso de Bacharelado em Ciência da Computação, tem-se a oportunidade de adquirir conhecimento de importantes fundamentos e técnicas da área de Engenharia de Software, dentre os quais se incluem a gestão de requisitos, arquitetura de software, testes de software, gestão de projetos de software, processos de software, entre outros. Consonante a isso, as atividades realizadas durante o estágio proporcionaram ao estagiário um real contato com um vasto fluxo de desenvolvimento de software, acrescido de um contexto onde o desenvolvimento é regido por uma cultura de práticas ágeis. Logo, tornou-se favorável consolidar na prática o conhecimento obtido durante o curso e adquirir experiência na execução do processo de desenvolvimento ágil de software.

O exercício dessas atividades submetia-se ao legítimo entendimento e à satisfatória observância às práticas de cada etapa do fluxo de desenvolvimento ágil experienciado no estágio. Como abordado na Seção 2, o estagiário pôde contemplá-lo transitando entre *framework* Scrum, procedimentos de controle de versões e criação de testes automatizados durante a codificação, práticas de integração e entrega contínua e monitoramento de sistemas. Além de experienciá-los em situações e desafios próprios às atividades realizadas.

A criação do SPA e o desenvolvimento de suas primeiras interfaces, descritos na Seção 3.1, forneceu aprendizado não apenas com a utilização da biblioteca React junto ao MUI, mas também com os procedimentos de configuração da nova aplicação para conformá-la aos padrões exigidos nas execuções das ações de CI e CD. Além disso, possibilitou ao estagiário a melhoria na interação e comunicação com os integrantes de outras equipes, visando a resolução de problemas e a realização de repasses de conhecimentos referentes à atividade. Apesar de a configuração realizada não ter incluído implementações de infraestrutura ou preparação de ferramentas de CI e CD, adequar a aplicação a uma sofisticada infraestrutura já mantida permitiu ao estagiário um contato inicial, proporcionando um importante conhecimento sobre o seu funcionamento no contexto da DTI e cliente.

Na Seção 3.2, relatou-se o processo de criação de funcionalidades para microsserviços, atividades que foram muito importantes para a aquisição de experiência com o *framework* ASP.NET Core e com a arquitetura de microsserviços, além de proporcionar a consolidação da compreensão e participação nos ritos do Scrum realizados pela equipe, incluindo os processos de refinamento dos itens e o uso do BDD. Durante essas atividades, também se adquiriu experiência com a criação de testes de unidade, com a prática TDD e com a aplicabilidade da *Feature*

Toggle, buscando observar seus benefícios para o desenvolvimento do software. Outro importante estudo realizado foi em relação a padrões arquiteturais, visto que houve a necessidade de entender uma arquitetura de um sistema baseada no *Onion Architecture*.

O desenvolvimento das novas funcionalidades também forneceu experiências quanto às práticas de revisão de código e homologação, permitindo ao estagiário compreender a execução do processo seguido pelas equipes para realizá-las adequadamente e sua importância no contexto de desenvolvimento ágil em que se atuou.

Como resultado das atividades realizadas, teve-se a geração de valor para o cliente, oriunda da entrega dos novos incrementos em suas aplicações. Obteve-se um positivo parecer retornado pelo cliente, como também pelas equipes nas quais o estagiário atuou, insinuando a efetiva absorção da cultura e das práticas de desenvolvimento ágil por parte do estagiário.

Outra observação é que, como mencionado, a criação e a observância da representação do fluxo de desenvolvimento ágil foi de grande importância para a realização das atividades durante o estágio. Sendo assim, pode ser interessante a utilização desse método para contribuir com o processo de adaptação de novos integrantes às equipes que mantêm um fluxo de desenvolvimento ágil, permitindo a rápida identificação das especifidades de cada contexto.

Com base nas atividades descritas, torna-se evidente a enorme contribuição do estágio não somente para o conhecimento técnico do estagiário, como também para o seu crescimento pessoal. Diante de todos esses benefícios obtidos durante a vivência no contexto descrito, também se torna importante recordar sobre a importância de manter e valorizar meios que possibilitam a aproximação da universidade a contextos externos, pois oferecem oportunidades que enriquecem a formação das pessoas.

REFERÊNCIAS

AHMED, T. M. et al. Studying the effectiveness of application performance management (APM) tools for detecting performance regressions for web applications: An experience report. In: **13th International Conference on Mining Software Repositories**. [S.l.: s.n.], 2016. p. 1–12.

AL-SAQQA, S.; SAWALHA, S.; ABDELNABI, H. Agile software development: Methodologies and trends. **International Journal of Interactive Mobile Technologies**, v. 14, 2020.

ALLIANCE, A. **What is Agile?** 2023. Disponível em: <https://www.agilealliance.org/agile101>.

AMARADRI, A. S.; NUTALAPATI, S. B. **Continuous Integration, Deployment and Testing in DevOps Environment**. 2016. Disponível em: <https://www.diva-portal.org/smash/get/diva2:1044691/FULLTEXT02.pdf>. Acesso em: 17 fev. 2023.

AMRIT, C.; MEIJBERG, Y. Effectiveness of Test-Driven Development and Continuous Integration: A case study. **IT professional**, v. 20, p. 27–35, 2018.

ARACHCHI, S.; PERERA, I. Continuous Integration and Continuous Delivery pipeline automation for agile software project management. In: **4th International Multidisciplinary Engineering Research Conference**. [S.l.: s.n.], 2018. p. 156–161.

BECK, K. **Test-Driven Development: By example**. [S.l.]: Addison-Wesley, 2003.

CHACON, S.; STRAUB, B. **Pro Git**. 2. ed. [S.l.]: Springer Nature, 2014.

CHEN, L. Continuous Delivery: Huge benefits, but challenges too. **IEEE Software**, v. 32, n. 2, p. 50–54, 2015.

DIGITAL.AI. **15th Annual State Of Agile Report**. [S.l.], 2021. Disponível em: <https://info.digital.ai/rs/981-LQX-968/images/RE-SA-15th-Annual-State-Of-Agile-Report.pdf>. Acesso em: 17 fev. 2023.

DRAGONI, N. et al. **Microservices: Yesterday, Today, and Tomorrow**. [S.l.]: Springer International Publishing, 2017.

DRIESSEN, V. **A successful Git branching model**. 2010. Disponível em: <https://nvie.com/posts/a-successful-git-branching-model>. Acesso em: 17 fev. 2023.

EBERT, C. et al. DevOps. **IEEE Software**, v. 33, n. 3, p. 94–100, 2016.

FITZGERALD, B.; STOL, K.-J. Continuous software engineering: A roadmap and agenda. **Journal of Systems and Software**, v. 123, p. 176–189, 2017.

FOWLER, M.; FOEMMEL, M. **Continuous Integration**. 2006. Disponível em: <https://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 17 fev. 2023.

FYBISH, R. **Implementing Microservices in NodeJS**. 2021. Disponível em: <https://frontegg.com/blog/implementing-microservices-in-nodejs>. Acesso em: 17 fev. 2023.

GOMES, P. **Unit Testing and the Arrange, Act and Assert (AAA) Pattern**. 2017. Disponível em: <https://medium.com/@pjbgf/title-testing-code-ocd-and-the-aaa-pattern-df453975ab80>. Acesso em: 17 fev. 2023.

GOTTESHEIM, W. Challenges, benefits and best practices of performance focused DevOps. In: **4th International Workshop on Large-Scale Testing**. [S.l.: s.n.], 2015. p. 3–3.

GRENNING, J. **Planning Poker or how to avoid analysis paralysis while release planning**. 2002. Disponível em: <https://www.scrummaster.dk/lib/AgileLeanLibrary/Topics/Estimation/PlanningPoker-v1.1.pdf>. Acesso em: 17 fev. 2023.

HODGSON, P. **Feature Toggles (aka Feature Flags)**. 2017. Disponível em: <https://martinfowler.com/articles/feature-toggles.html>. Acesso em: 17 fev. 2023.

HOORN, A. van et al. **Continuous Monitoring of Software Services: Design and Application of the Kieker Framework**. Kiel: Selbstverlag des Instituts für Informatik, Kiel, 2009. Disponível em: <https://macau.uni-kiel.de/receive/macau_mods_00001840>. Acesso em: 17 fev. 2023.

HUMBLE, J.; FARLEY, D. **Entrega Contínua: Como entregar software**. 1. ed. [S.l.]: Bookman Editora, 2014.

JADHAV, M. A.; SAWANT, B. R.; DESHMUKH, A. Single Page Application using AngularJS. **International Journal of Computer Science and Information Technologies**, v. 6, p. 2876–2879, 2015.

LAUKKANEN, E.; ITKONEN, J.; LASSENIUS, C. Problems, causes and solutions when adopting continuous delivery — A systematic literature review. **Information and Software Technology**, v. 82, p. 55–79, 2017.

LIU, Y.; LI, C.; LIU, W. Integrated solution for timely delivery of customer change requests: A case study of using DevOps approach. **International Journal of u-and e-Service, Science and Technology**, v. 7, p. 41–50, 2014.

LOELIGER, J.; MCCULLOUGH, M. **Version Control with Git: Powerful tools and techniques for collaborative software development**. [S.l.]: O'Reilly, 2012.

MILLER, J. **Which Version Control System Should Your Developers Use?** 2021. Disponível em: <https://www.bairesdev.com/blog/which-version-control-system-developers-use>. Acesso em: 17 fev. 2023.

NORTH, D. et al. **Introducing BDD**. 2006. Disponível em: <https://dannorth.net/introducing-bdd>. Acesso em: 17 fev. 2023.

PALERMO, J. **The Onion Architecture**. 2008. Disponível em: <https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1>. Acesso em: 17 fev. 2023.

PÉREZ, J. F.; WANG, W.; CASALE, G. Towards a DevOps approach for software quality engineering. In: **1st Workshop on Challenges in Performance Methods for Software Development (WOSP-C)**. [S.l.: s.n.], 2015. p. 5–10.

PRESSMAN, R.; MAXIM, B. **Engenharia De Software: Uma Abordagem Profissional**. [S.l.]: McGraw Hill, 2016.

RIVERO, L. et al. Implementando o Gitflow para Gerência de Configuração em um projeto de desenvolvimento de software ágil: Um relato de experiência. **XII Computer on the Beach em 2021**, v. 12, p. 178–185, 2021.

RUNESON, P. A survey of unit testing practices. **IEEE Software**, v. 23, n. 4, p. 22–29, 2006.

SAKHNIUK, M. **What is state in React?** 2022. Disponível em: <https://iq.js.org/questions/react/what-is-state-in-react>. Acesso em: 17 fev. 2023.

SCHWABER, K.; SUTHERLAND, J. **O Guia do Scrum: O Guia Definitivo para o Scrum: As Regras do Jogo.** 2020. Disponível em: <https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-PortugueseBR-3.0.pdf>. Acesso em: 17 fev. 2023.

SMITH, A. C. **How Design Systems Make Teams Work Smarter**. 2021. Disponível em: <https://envylabs.com/insights/software-development-projects-design-improvements>. Acesso em: 17 fev. 2023.

SOFTEX, M. **Guia de Implementação – Parte 2: Fundamentação para Implementação do Nível F do MR-MPS-SW:2016**. [S.l.], 2016. Disponível em: <https://www.softex.br/wp-content/uploads/2016/04/MPS.BR_Guia_de_Implementacao_Parte_2_2016.pdf>. Acesso em: 17 fev. 2023.

SPINELLIS, D. Git. **IEEE Software**, v. 29, n. 3, p. 100–101, 2012.

SULEMANI, M. **CRUD operations explained: Create, read, update, delete**. 2021. Disponível em: <https://www.educative.io/blog/crud-operations>. Acesso em: 17 fev. 2023.

THÖNES, J. Microservices. **IEEE Software**, v. 32, n. 1, p. 116–116, 2015.

TOTVS. **Kanban: conceito, como funciona, vantagens e implementação**. 2022. Disponível em: <https://www.totvs.com/blog/negocios/kanban>. Acesso em: 8 mar. 2023.

VIETRO, Í. L. de. **Fluxo de desenvolvimento com GitFlow**. 2015. Disponível em: <https://imasters.com.br/agile/fluxo-de-desenvolvimento-com-gitflow>.

WALLER, J.; EHMKE, N. C.; HASSELBRING, W. Including performance benchmarks into Continuous Integration to enable DevOps. **Software Engineering Notes**, v. 40, n. 2, p. 1–4, 2015.

WIEGERS, K. E. **Peer Reviews in Software: A Practical Guide**. 1. ed. [S.l.]: Addison-Wesley, 2002.

ZHAO, J. T.; JING, S. Y.; JIANG, L. Z. Management of API gateway based on micro-service architecture. **Journal of Physics: Conference Series**, v. 1087, n. 3, p. 032032, 2018.

ZIEMOŃSKI, G. **Onion Architecture is Interesting**. 2017. Disponível em: <https://dzone.com/articles/onion-architecture-is-interesting>. Acesso em: 17 fev. 2023.

ZOLKIFLI, N. N.; NGAH, A.; DERAMAN, A. Version control system: A review. **Procedia Computer Science**, v. 135, p. 408–415, 2018.