UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

RANGEL CRISTIANO TONIN

ESTRATÉGIAS DE CÓDIGO-FONTE CENTRALIZADO PARA A GESTÃO EFICIENTE DE VARIANTES DE PRODUTOS DE SOFTWARE

CAXIAS DO SUL

2024

RANGEL CRISTIANO TONIN

ESTRATÉGIAS DE CÓDIGO-FONTE CENTRALIZADO PARA A GESTÃO EFICIENTE DE VARIANTES DE PRODUTOS DE SOFTWARE

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação da Universidade de Caxias do Sul.

Orientador(a): Profª. Drª. Elisa Boff

CAXIAS DO SUL

2024

RANGEL CRISTIANO TONIN

ESTRATÉGIAS DE CÓDIGO-FONTE CENTRALIZADO PARA A GESTÃO EFICIENTE DE VARIANTES DE PRODUTOS DE SOFTWARE

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação da Universidade de Caxias do Sul.

BANCA EXAMINADORA

Profª. Drª. Elisa Boff
Universidade de Caxias do Sul - UCS

Prof. Dr. Daniel Luis Notari
Universidade de Caxias do Sul - UCS

Profª. Drª. Iraci Cristina da Silveira de Carli
Universidade de Caxias do Sul - UCS

RESUMO

Em um mercado cada vez mais competitivo, a capacidade de uma empresa de adaptar e diversificar seus produtos é crucial para o sucesso e sustentabilidade a longo prazo. A empresa Eterno Software enfrentava desafios significativos relacionados ao custo e à agilidade na criação de variantes de software. O objetivo deste estudo é avaliar e comparar as principais alternativas disponíveis. Além disso, foi desenvolvida uma solução que se propõe a gerenciar eficientemente a diversificação de produtos de software através de repositórios centralizados, combinando aspectos de estratégias de branching adaptadas especificamente para as necessidades da empresa. Um componente chave dessa solução é a automação implementada para facilitar o gerenciamento e atualização dessas branches. Foi conduzida uma comparação entre o desenvolvimento de variantes de software com diferentes metodologias em cenários de testes controlados. Os resultados dessa comparação indicaram que a solução automatizada proposta foi satisfatória em relação às outras abordagens conhecidas no mercado, reduzindo o tempo necessário para desenvolver e testar novas variantes e melhorando a consistência e a qualidade dos produtos resultantes. Essa conclusão reforça a eficácia da solução desenvolvida, demonstrando que ela não apenas otimiza o processo de diversificação de produtos, mas também contribui significativamente para a sustentabilidade e competitividade da Eterno Software no mercado.

Palavras-chave: Software. Diversificação. Variantes. Repositórios Centralizados. Branching.

ABSTRACT

In an increasingly competitive market, a company's ability to adapt and diversify its products is crucial to its long-term success and sustainability. The company Eterno Software faced significant challenges related to cost and agility in creating software variants. The aim of this study is to evaluate and compare the main alternatives available. In addition, a solution was developed that proposes to efficiently manage the diversification of software products through centralized repositories, combining aspects of branching strategies tailored specifically to the company's needs. A key component of this solution is the automation implemented to facilitate the management and updating of these branches. A comparison was made between the development of software variants with different methodologies in controlled test scenarios. The results of this comparison indicated that the proposed automated solution was satisfactory compared to other approaches known on the market, reducing the time needed to develop and test new variants and improving the consistency and quality of the resulting products. This conclusion reinforces the effectiveness of the solution developed, demonstrating that it not only optimizes the product diversification process, but also contributes significantly to Eterno Software's sustainability and competitiveness in the market.

Keywords: Monorepo. Branching. Version Control. Diversification. Variants. Centralized Repositories.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

CI	Integração contínua
CD	Entrega contínua
SVN	Subversion
VCS	Sistema de controle de versão

SUMÁRIO

1 INTRODUÇÃO

A capacidade de uma empresa de diversificar seus produtos é fundamental para manter sua competitividade em um mercado em constante evolução. A diversificação não apenas ajuda a atingir uma base de clientes mais ampla, mas também minimizar riscos, explorando oportunidades em diferentes segmentos de mercado (PORTER, 1985). No contexto do desenvolvimento de software, essa diversificação na maioria das vezes significa criar variantes de um produto principal, cada uma adaptada a requisitos específicos ou a diferentes nichos de mercado.

Essa variedade de produtos, embora possa ser vantajosa comercialmente, pode conter enorme complexidade no gerenciamento do desenvolvimento e manutenção do software. Nesse trabalho será apresentada uma solução utilizando técnicas para centralização do código-fonte, com a finalidade de proporcionar um meio mais robusto e eficaz de desenvolver essas variantes de produtos de forma paralela.

1.1 OBJETIVOS

Propor e comparar a utilização de centralização do código-fonte com outras alternativas do mercado para facilitar a gestão das diversas variantes de um mesmo projeto que podem existir.

1.2 ESTRUTURA DO TRABALHO

O presente trabalho está organizado da seguinte forma:

- No Capítulo 2 é feita a análise das características normalmente encontradas nas variantes de produtos de softwares. Além disso, serão definidos fatores a serem considerados na estratégia de utilização de monorepos, ao mesmo tempo em que serão estudadas outras abordagens para a implementação desse tipo de cenário.
- No Capítulo 3 foram traçados os resultados e desafios encontrados, através de estudo de casos reais. Também foi criado um tutorial base para criação de diversos produtos em um mesmo repositório via ferramenta de versionamento de código.
- O Capítulo 4 foi implementado a automatização para facilitar os processos de atualização do código-fonte centralizado. Além disso, foi realizado a implementação de um projeto teste em um ambiente controlado para efetuar análises e comparações entre as outras abordagens estudadas.
- Por fim, no Capítulo 5 são apresentadas as considerações finais do trabalho.

2 DIVERSIFICAÇÃO DE PRODUTOS DE SOFTWARE

A diversificação de produtos é uma estratégia chave para empresas que buscam crescimento sustentável e vantagem competitiva no mercado. Sohl, Vroom e McCann (2020) afirmam que empresas de software que diversificam seus produtos possuem a capacidade de explorar novos mercados e se adaptar melhor às mudanças no ambiente de negócio.

A principal característica das variantes de produto é a personalização. Essa personalização pode variar desde a interface do usuário até regras de negócio complexas, permitindo que as empresas ofereçam soluções mais alinhadas com as expectativas e necessidades de cada cliente. Santana (2023) afirma a importância dessas práticas no contexto de software, onde a capacidade de personalizar produtos sem reinventar o código base é fundamental.

Sohl, Vroom e McCann (2020) relatam que a diversificação no desenvolvimento de software não apenas ajuda a empresa a minimizar riscos associados à dependência de um único produto, mas também estimula a inovação. Ao desenvolver variantes de um produto, as empresas podem atender a um maior número de clientes, ampliando assim, sua posição no mercado.

No entanto, a gestão eficaz de múltiplos produtos de software apresenta desafios significativos, principalmente relacionados ao gerenciamento desses projetos e à manutenção da qualidade e consistência entre as suas diferentes adaptações. Brito, Terra e Valente (2018) relatam que a utilização de um único repositório (monorepo) para armazenar vários projetos, atrelado a estratégias bem definidas de controle, são fundamentais para contornar esse tipo de desafios, permitindo que as equipes trabalhem de maneira mais eficiente e consistente.

Além da utilização de monorepos existem abordagens alternativas, como o uso de múltiplos repositórios (multirepo), criação de bibliotecas e alternância de recursos (feature flags), que também podem ser utilizados na criação e manutenção das variantes de um mesmo sistema, cada uma com seus benefícios e desafios. Uma análise comparativa entre essas metodologias é essencial para compreendermos como elas podem ser aplicadas no desenvolvimento de software, influenciando não apenas a qualidade técnica, mas também o seu sucesso comercial.

Este capítulo tem como objetivo aprofundar essas abordagens, buscando compreender como uma gestão eficiente de diferentes variantes de um sistema pode ser um diferencial competitivo no desenvolvimento de software.

2.1 REPOSITÓRIOS DE CÓDIGO-FONTE

Repositórios podem ser definidos como uma plataforma para armazenamento e gerenciamento de código-fonte, que têm como principal objetivo facilitar a colaboração entre os programadores durante o desenvolvimento dos projetos (DECAN *et al.*, 2022). As plataformas

de repositório mais populares são GitHub, GitLab e Bitbucket, cada uma oferecendo diferentes funcionalidades para suportar o desenvolvimento de software.

Segundo Kinsman *et al.* (2021), algumas das principais funcionalidades de repositórios de código-fonte são:

- **Controle de Versão:** Permite aos desenvolvedores manter um histórico detalhado das mudanças no código, essencial para o gerenciamento de projetos complexos.
- **Colaboração:** Ferramentas para revisões de código facilitam a colaboração entre desenvolvedores, mesmo que de diferentes localidades.
- **Segurança:** Mantem histórico das mudanças no código para rastrear o que foi alterado. Além de possuir controles de acesso e permissões.
- **Integração e Entrega Contínua:** A integração de CI/CD automatiza o teste e a distribuição de software, agilizando o ciclo de lançamentos.

Pode-se concluir que repositórios são fundamentais no desenvolvimento de sistemas, sendo utilizados tanto por projetos abertos quanto privados em uma escala global. Eles não apenas facilitam a gestão dos projetos de software, mas também promovem uma colaboração mais eficaz e um desenvolvimento mais rápido e seguro.

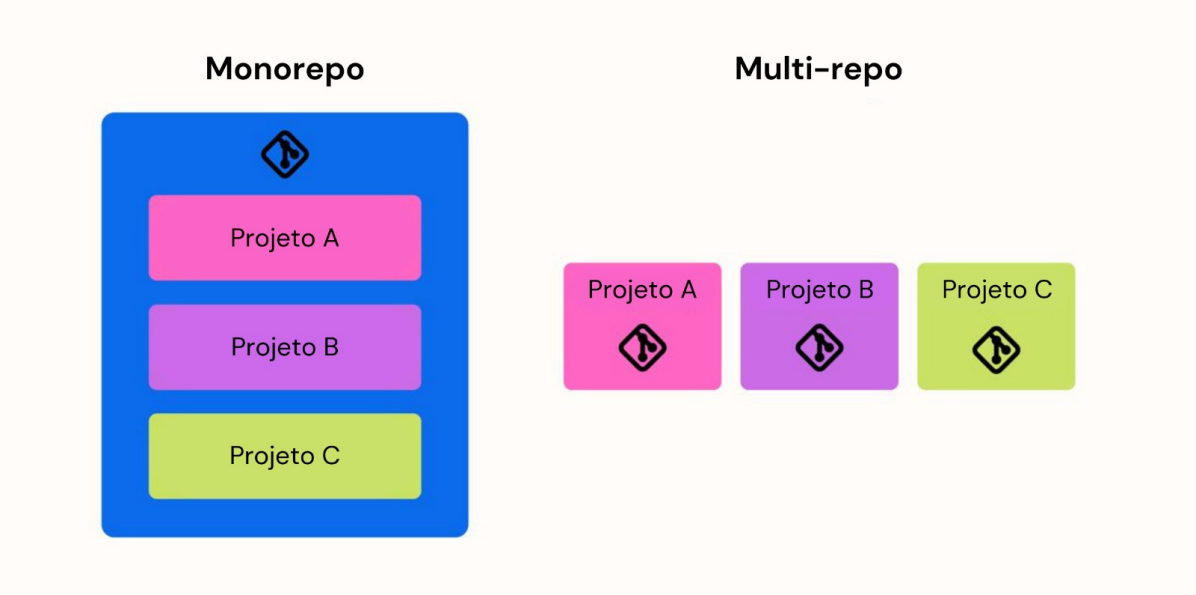
2.2 MONOREPO VS MULTI-REPO

ThoughtWorks (2020) caracterizam monorepo, ou repositório monolítico, como a prática de armazenar todo o código fonte de diferentes projetos ou componentes em um único repositório de controle de versão. Sua definição, como representado pela Figura 1, difere da definição de múltiplos repositórios, também conhecido popularmente como multi-repo, na qual, como afirma Log (2022), consiste em uma estrutura em que cada componente, serviço ou produto é mantido em seu próprio repositório. O autor ainda relata que essa abordagem aumenta a independência entre os componentes, permitindo que as equipes desenvolvam, testem e lancem atualizações para partes específicas do produto sem impactar outras. Porém, quando colocamos no contexto de produtos variantes, podemos concluir que tal definição seria uma desvantagem, pois muito do código entre os repositórios seria repetido e por isso, de difícil manutenção.

DOAN (2023) compara monorepo com a utilização de multi-repo e enfatiza como principais vantagens os seguintes aspectos:

- Colaboração e compartilhamento de código mais fáceis: com todo o código em um só lugar, é mais fácil compartilhar e colaborar no código entre projetos e equipes. Os desenvolvedores podem facilmente fazer alterações que afetam vários projetos ao mesmo tempo.

Figura 1 – Monorepo x Multi-repo



Fonte: Raftt (2022)

- Construções e implantações simplificadas: por ter um único repositório, as compilações e as implantações podem ser simplificadas, reduzindo a complexidade do processo de desenvolvimento.
- Maior qualidade e consistência do código: Monorepos incentivam a reutilização e compartilhamento de código, o que pode resultar em um código mais consistente em projetos.
- Melhores testes e correção de erros: com todo o código em um só lugar, é mais fácil testar mudanças em vários projetos e identificar e corrigir erros que afetam vários componentes sem precisar replicar em outro lugar.
- Melhor gerenciamento de projetos: Monorepos permitem um melhor gerenciamento do projeto, ao possuir um local central para todo o código, reduz o tempo e esforço necessários para coordenar e gerenciar vários repositórios. Isso também permite uma alocação e planejamento de recursos mais eficientes.

Além disso, os autores Brito, Terra e Valente (2018) ainda citam algumas das grandes empresas, como Google, Facebook e Microsoft, que utilizam monorepos em seus projetos e provam na prática que manter vários projetos em um único repositório pode ser uma estratégia viável e bem sucedida. Segundo Brousse (2019), um estudo de caso realizado na Google estudou as compensações entre o uso de arquiteturas monorepo e multi-repo. Os resultados indicam que, embora ambos os modelos apresentem vantagens e desvantagens específicas, os monorepos tendem a promover a consistência e a qualidade do código. Além disso, essa estrutura facilita o engajamento dos engenheiros de software com o conhecimento institucional da em-

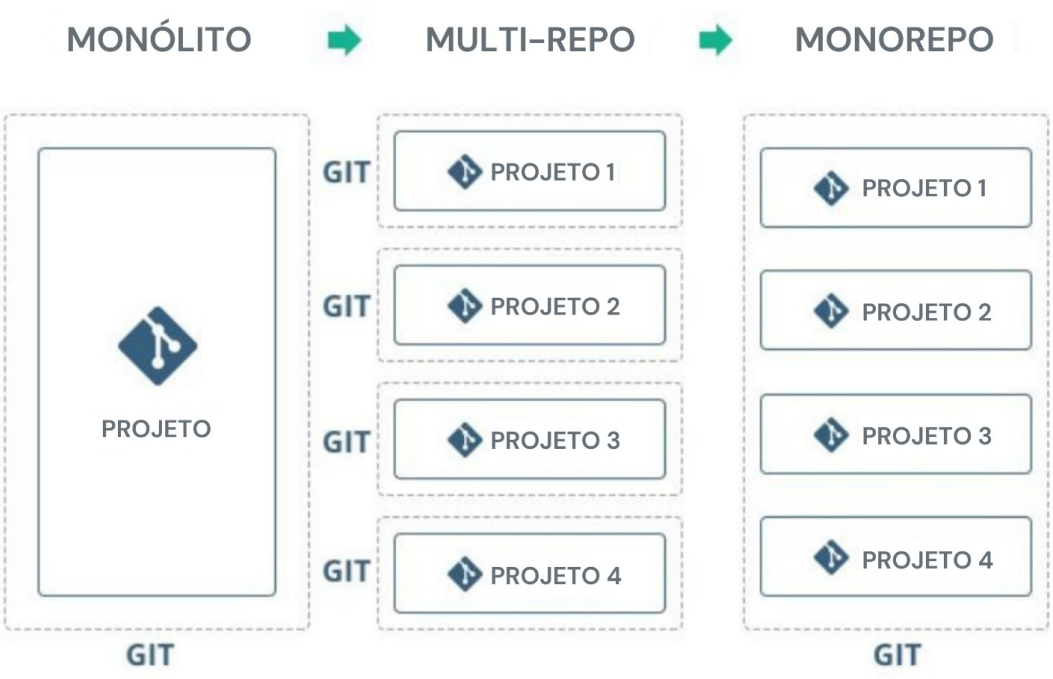
presa, permitindo que aprendam e se desenvolvam por meio do acesso direto ao código-fonte completo.

2.2.1 MONOREPO NÃO É MONÓLITO

Apesar de parecerem similares devido aos nomes e da ideia de centralização de código, é importante ressaltar que monorepo e monólito são conceitos distintos na engenharia de software, cada um com suas características e propósitos específicos.

Um monorepo (repositório monolítico), conforme é possível observar na Figura 2, armazena todo o código de vários projetos, serviços ou pacotes em um único repositório de controle de versão (BRITO; TERRA; VALENTE, 2018). Por outro lado, como explicam os autores Blinowski, Ojdowska e Przybyłek (2022), um monólito integra todas as funcionalidades de uma aplicação em uma única base de código, tornando mais fácil de se inicar um novo projeto, porém, à medida que esse sistema cresce, um monólito tende a se tornar difícil de escalar e manter, pois qualquer pequena alteração requer a reimplementação de todo o projeto.

Figura 2 – Monolítico x Monorepo x Multi-repo



Fonte: Pessôa (2022)

2.2.2 INFLUÊNCIA DA CULTURA ORGANIZACIONAL NA ESCOLHA DA ESTRUTURA DE REPOSITÓRIO

Segundo Brousse (2019) a decisão sobre a estrutura de repositório nas empresas vai muito além da simples escolha entre monorepo e multi-repo, sendo profundamente influenciada

pela própria cultura organizacional de cada uma.

A Netflix, por exemplo, optou pelo modelo de multi-repo, pois está mais inclinada com a sua cultura de "liberdade e responsabilidade"(BROUSSE, 2019). Esse modelo permite aos engenheiros de software autonomia para escolher as ferramentas mais adequadas para seus projetos. De maneira semelhante, a Amazon adota o multi-repo, apoiando uma cultura que fomenta a competição interna entre equipes trabalhando em projetos similares, o que estimula a inovação e a eficiência (BROUSSE, 2019).

Como podemos observar na Tabela 1, grandes empresas como Google e Facebook, optaram pela estratégia de monorepo, segundo Brousse (2019) não apenas por questões técnicas, mas como uma extensão de suas culturas que enfatizam colaboração, rapidez e eficiência. Já a Microsoft fez a transição de multi-repo para monorepo como parte de uma reformulação cultural que visa promover uma constante renovação e inovação (BROUSSE, 2019). A empresa realizou diversas melhorias no Git, capacitando-o para gerenciar o que é hoje o maior monorepo Git do mundo. Essa mudança estratégica foi impulsionada pela necessidade de superar a fragmentação causada pela estrutura de multi-repo, promovendo um ambiente de trabalho mais coeso e colaborativo, o que é crucial para a dinâmica de desenvolvimento na Microsoft (BROUSSE, 2019).

Tabela 1 – Estrura de Repositório Utilizadas por Grandes Empresas

Estrutura	**Empresas**
Monorepo	Google, Facebook, Microsoft, Uber, Twitter, React, Angular, Babel, Kubernetes
Multi-repo	Amazon, Netflix, Lyft

É possível concluir que a escolha pela estruturação de repositório está ligada a cultura da empresa, assim como, a cultura pode sofrer impactos de acordo com a estratégia escolhida. Um ambiente mais colaborativo e integrado tende a utilizar estratégias de monorepo, por outro lado, empresas que valorizam autonomia individual e a capacidade de inovação rápida e independente podem preferir a flexibilidade de multi-repos.

2.3 ESTRATÉGIAS E FERRAMENTAS NA UTILIZAÇÃO DE MONOREPO EM EMPRESAS LÍDERES DE MERCADO

A empresa Google, conhecida por sua vasta variedade de produtos e tecnologias inovadoras, adota uma abordagem única para o gerenciamento de seu código-fonte através de um monorepo. Segundo GEHMAN (2021), a estratégia de manter todo o código em um único repositório centralizado demonstra ser uma abordagem eficiente para lidar com os desafios de escalabilidade e colaboração em seu imenso e dinâmico ambiente de desenvolvimento.

Na Figura 3, percebe-se a enorme quantidade de dados e projetos localizados no repositório da Google já no ano de 2015. Com milhões de arquivos criados e bilhões de linhas de código-fonte escritos em um único lugar, este repositório foi organizado em diretórios que correspondem a diferentes projetos e componentes, facilitando a navegação e a organização do código. Entretanto, reconhecendo as limitações das ferramentas no mercado para gerenciar seu tamanho e complexidade, a empresa acabou desenvolvendo sistemas internos como Piper para controle de versão e Bazel para compilação (build) e testes automatizados (MATUTE; CHEUNG; CHASINS, 2022).

Figura 3 – Estatísticas do repositório da Google até Janeiro de 2015

Fonte: Potvin e Levenberg (2016)

Piper suporta um modelo de desenvolvimento em que milhares de desenvolvedores podem trabalhar simultaneamente no mesmo código, sem significativa degradação do desempenho (FRÖMMGEN *et al.*, 2024). Bazel, por outro lado, permite realizar builds e testes escaláveis, garantindo que apenas as partes do sistema que foram alteradas sejam reconstruídas e testadas novamente (ALFADEL; MCINTOSH, 2024). Além disso, ferramentas internas como o Code Search permitem aos desenvolvedores pesquisar e navegar facilmente pelo código, essencial em um repositório de grande escala. As dependências são gerenciadas centralmente, o que significa que as atualizações em bibliotecas compartilhadas são propagadas automaticamente para todos os projetos que as utilizam (MATUTE; CHEUNG; CHASINS, 2022). É através dessas e outras ferramentas que a Google consegue gerenciar todos os seus projetos e bilhões de linhas de código em somente um único repositório.

Conforme relata Elijah (2023), outra grande empresa que adotou a utilização de monorepos foi o Facebook. O autor descreve como essa abordagem também mostrou ser um desafio devido ao tamanho do seu repositório, a complexidade da integração e a necessidade de um sistema robusto de gerenciamento de permissões. Para lidar com esses problemas, a empresa utiliza uma variedade de ferramentas internas e de código aberto, como Buck para realizar os builds, Watchman para monitoramento de alterações de arquivos e Phabricator para revisão de código. Essas ferramentas desempenham um papel crucial na facilitação do desenvolvimento em larga escala dentro do contexto do monorepo do Facebook.

Já a Microsoft, outra conhecida no cenário mundial, usa a ferramenta Rush para orquestrar seu monorepo com suas centenas de projetos. Além disso, a empresa também desenvolveu outra ferramenta popular chamada Lage, que é amplamente utilizada em seus processos de desenvolvimento. Lage foi projetada para acelerar a execução de scripts e tarefas em monorepos grandes, utilizando cache inteligente e paralelismo para otimizar o desempenho. Isso permite que as equipes de desenvolvimento da Microsoft gerenciem de forma eficiente a construção, os testes e outras operações em seus numerosos projetos dentro do mesmo repositório (ELIJAH, 2023).

Ferramentas personalizadas ajudam a simplificar a colaboração entre equipes, a garantir a consistência do código e a melhorar a escalabilidade dos projetos. Esse desenvolvimento de ferramentas por parte dessas grandes empresas não apenas resolve desafios internos, mas também atrai outras empresas a adotar monorepos ao perceberem os benefícios dessa infraestrutura. Como resultado, um ecossistema de monorepos robusto e bem suportado está se formando, incentivando mais empresas a migrar para essa abordagem devido às vantagens demonstradas e às soluções disponíveis no mercado.

2.4 FERRAMENTAS DE VERSIONAMENTO

Ponuthorai e Loeliger (2022) definem uma ferramenta de versionamento, ou sistema de controle de versão (VCS), como essencial no gerenciamento de software das diferentes versões de códigos, permitindo que os desenvolvedores trabalhem ao mesmo tempo sem sobrescrever o trabalho um do outro. Além disso o VCS também mantém um histórico completo de cada arquivo, com detalhes sobre as alterações, como por exemplo, o autor e a data em que foram feitas. Essa funcionalidade não só facilita o rastreamento de alterações, mas também possibilita reverter arquivos para estados anteriores, o que é crucial para resolver erros ou conflitos que possam vir a surgir.

No mercado atual, Git e Subversion (SVN) se destacam como as ferramentas de versionamento mais populares. A Tabela 2 apresenta um comparativo, com base nos argumentos apresentados pelos autores Khleel e Károly (2020) sobre Git e SVN.

O Mercurial é outra ferramenta de controle de versão relativamente recente que ganhou

Tabela 2 – Comparação entre Git e SVN

Critério	**Git**	**SVN**
Tipo	Distribuído	Centralizado
Armazenamento	Cada cópia do repositório contém todo o histórico de versões	O histórico é mantido em um servidor central; as cópias de trabalho contêm apenas a versão mais recente
Operações Offline	A maioria das operações pode ser realizada offline	Requer conexão com o servidor para a maioria das operações
Branching e Merging	Facilita branching e merging com operações rápidas e eficientes	Suporta branching e merging, mas as operações podem ser mais complexas e pesadas
Curva de Aprendizado	Considerado mais complexo, com uma curva de aprendizado mais acentuada	Geralmente considerado mais simples e fácil de aprender
Popularidade	Muito popular, especialmente em projetos open source	Amplamente utilizado, especialmente em empresas com infraestrutura existente baseada em SVN

popularidade nos últimos anos. A Tabela 3 foi desenvolvida através dos conceitos e comparativos apresentados pelos autores Khleel e Károly (2020), destacando as principais características e diferenças entre Git e Mercurial.

Tabela 3 – Comparação entre Git e Mercurial (O'SULLIVAN, 2009)

Característica	**Git**	**Mercurial**
Modelo de Dados	Distribuído	Distribuído
Ramo (Branching)	Leve, flexível, fácil de criar e mesclar (segundo o autor)	Semelhante ao Git, mas com menos flexibilidade
Performance	Mais Rápido que seu concorrente	Mais lento em algumas operações
Comunidade e Suporte	Maior comunidade e ampla adoção	Comunidade menor e menos popular
Ferramentas de Interface	Interface de linha de comando (CLI) e várias interfaces gráficas disponíveis	Menos opções de interface gráfica
Ecossistema e Integração	Integração com uma ampla gama de ferramentas e serviços	Integração com menos ferramentas e serviços
Complexidade de Aprendizado	Mais complexo para iniciantes	Menor curva de aprendizagem para iniciantes

Através destes comparativos é possível concluir que o Git possui muitas características que o tornam uma opção mais atrativa do que o Mercurial e o SVN (Subversion). Em comparação com o Mercurial, o Git oferece maior flexibilidade e eficiência no gerenciamento de branches, permitindo a criação e fusão de branches de maneira rápida e sem complicações. O Git também se destaca por sua comunidade de usuários e desenvolvedores significativamente

maior, resultando em uma abundância de recursos, tutoriais e suporte técnico. Além disso, a integração do Git com uma ampla variedade de ferramentas e serviços, como GitHub, GitLab e Bitbucket, torna-o uma escolha mais versátil e robusta para projetos de diferentes escalas e complexidades.

Quando comparado ao SVN, o Git proporciona uma arquitetura distribuída, permitindo que cada desenvolvedor tenha uma cópia local completa do repositório, melhorando assim, a colaboração e a agilidade no desenvolvimento do sistema. O SVN, por ser um sistema de controle de versão centralizado, não oferece essa vantagem, o que pode limitar a flexibilidade e a independência dos desenvolvedores. Além disso, o desempenho do Git em operações comuns, como commits e merges, é geralmente mais rápido do que no SVN e o Mercurial, proporcionando uma experiência mais ágil e eficiente para os desenvolvedores. Esses fatores combinados fazem do Git a ferramenta de controle de versão preferida por muitas equipes de desenvolvimento ao redor do mundo.

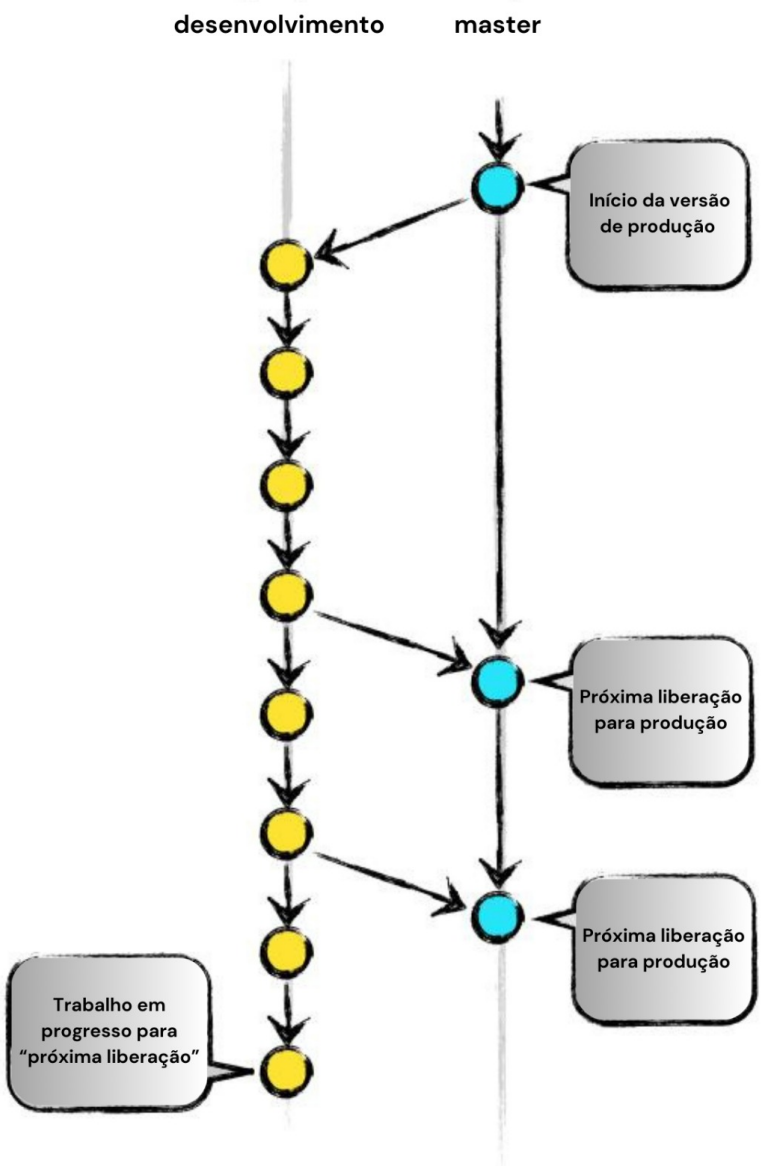
2.5 ESTRATÉGIAS DE BRANCHING

Branching, no contexto de controle de versão, refere-se à prática de criar ramificações (branches) a partir do código principal (main ou master) para desenvolver funcionalidades ou corrigir erros de maneira isolada (AGGARWAL; SINGH; KUMAR, 2022). Conforme é possível ser analisado na Figura 4, essas ramificações possibilitam que diferentes tarefas de desenvolvimento sejam realizadas ao mesmo tempo, facilitando a colaboração entre os membros da equipe e a integração contínua de novas alterações, reduzindo assim o risco de conflitos no código-fonte. Conforme apontado por Aggarwal, Singh e Kumar (2022) em seu modelo Git Flow, uma estratégia de branching bem definida pode melhorar significativamente o processo de entrega, proporcionando um melhor gerenciamento de novas funcionalidades e correções de erros no sistema. É possível concluir que utilizar de estratégias de branching é uma prática essencial na engenharia de software atual, facilitando uma abordagem mais estruturada e organizada para o desenvolvimento de projetos complexos.

2.6 MONOREPO ALIADO AO USO DE BRANCHING

A combinação de monorepo com estratégias de *branching* oferece uma abordagem eficiente para a criação e gerenciamento de variantes de um produto de software. Em um monorepo, todos os componentes do projeto são armazenados em um único repositório de controle de versão, simplificando assim, o compartilhamento de código e a coordenação entre diferentes equipes. Ao utilizar estratégias de branching dentro deste contexto, as empresas conseguem isolar o desenvolvimento de diferentes variantes de um produto em branches separadas. Isso é ilustrado na Figura 5, onde observamos um nó principal denominado master. Esta branch principal do repositório contém o código-fonte base comum às branches secundárias, tais como

Figura 4 – Estrutura de funcionamento da branch Master/Main

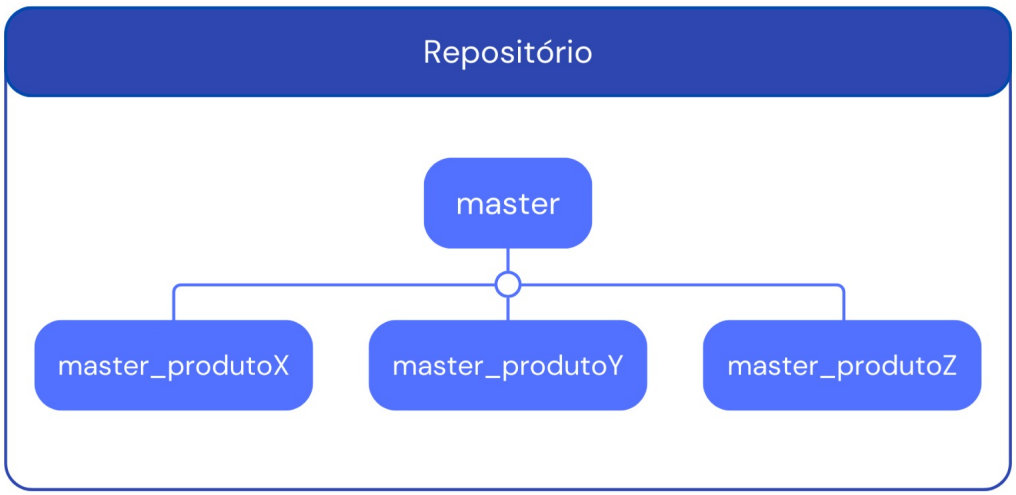


Fonte: Driessen (2010)

master_produtoX, master_produtoY e master_produtoZ. Cada uma dessas branches secundárias herda o código da branch master, mas também inclui suas próprias características e funcionalidades exclusivas, permitindo que equipes trabalhem em paralelo sem interferir umas nas outras e ainda consigam manter a compatibilidade e consistência do código-fonte.

Essa metodologia facilita a criação de produtos personalizados, com o objetivo principal de atender a requisitos específicos dos diferentes segmentos de mercado ou clientes, ao mesmo tempo em que mantém o código centralizado e de fácil gerenciamento. A utilização de branches em um monorepo assegura que qualquer mudança feita em componentes compartilhados possa ser imediatamente refletida em todas as suas variantes (branches), melhorando a consistência e a eficiência na manutenção ou desenvolvimento de novas funcionalidades.

Figura 5 – Exemplo de estrutura de um monorepo utilizando branches para variantes de produto



Fonte: O Autor

Também vale ressaltar que utilizar as branches como variantes de produtos em um monorepo, ao invés de depender exclusivamente de ferramentas especializadas, oferece uma abordagem mais direta e acessível para o gerenciamento de diferentes versões de um produto. As branches permitem um isolamento claro de funcionalidades e desenvolvimento específico para cada variante, facilitando a colaboração entre equipes sem a necessidade de configurar e aprender ferramentas adicionais. Esse método é particularmente útil quando se busca simplicidade e rapidez na implementação, aproveitando os recursos nativos de controle de versão oferecidos por plataformas como Git.

Entretanto, essa estratégia requer uma avaliação cuidadosa do núcleo de código compartilhado entre os projetos. Essa abordagem é particularmente eficaz quando as variantes do produto possuem uma base de código substancialmente semelhante, permitindo que as mudanças no código principal sejam facilmente propagadas para todas as branches. Se o núcleo de código é altamente compartilhado e consistente entre os projetos, o uso de branches pode simplificar significativamente o gerenciamento e a integração contínua, promovendo um desenvolvimento mais coeso e eficiente. No entanto, em situações onde os projetos possuem diferenças significativas ou a necessidade de isolamento é crucial, o uso de ferramentas especializadas pode oferecer maior flexibilidade e controle. Portanto, a decisão de utilizar branches deve ser baseada na análise da similaridade do núcleo de código e nas necessidades específicas de cada projeto.

2.7 OUTRAS ALTERNATIVAS

Enquanto a utilização de branching oferece vantagens como facilitação na gestão de dependências e na unificação do código, existem outras estratégias que devem ser levadas em consideração e que podem ser até mesmo mais adequadas dependendo das necessidades específicas do projeto ou da organização. Cada uma dessas abordagens tem suas vantagens e desvantagens e pode ser escolhida com base em fatores como: estrutura da equipe, natureza do produto, práticas de integração e entrega contínuas, e os objetivos estratégicos da organização.

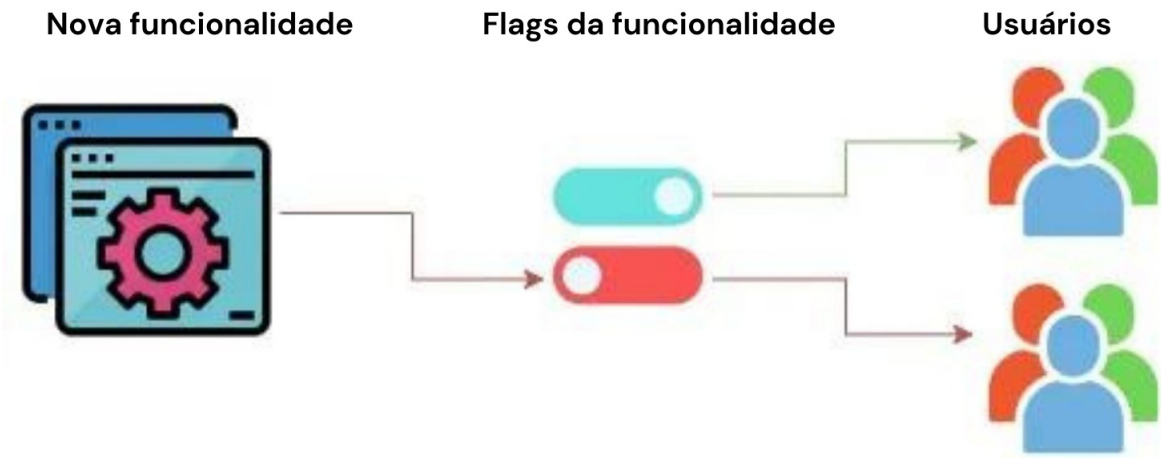
Nesta seção serão abordadas estratégias de feature flags e criação de bibliotecas compartilhadas, buscando compará-las individualmente com a utilização de branching.

2.7.1 FEATURE FLAGS

Feature Flags, também conhecidas como *Feature Toggles*, é uma técnica de programação que consiste em ativar ou desativar uma funcionalidade em tempo real pelo próprio adiministrador do sistema (ROSU; TOGAN, 2023).

Observando a Figura 6, nota-se que uma forma de implementação das Feature Flags é através da criação de um meio de consulta que retorna se o sistema possui ou não determinada funcionalidade para o usuário que está acessando o sistema.

Figura 6 – Feature Flag



Fonte: Atlassiann (2024)

2.7.2 FEATURE FLAGS VS BRANCHING

A partir das definições apresentadas por Rosu e Togan (2023), é possível levantarmos alguns pontos de observação em comparação a utilização de ramificações na criação de variantes de produto.

- Complexidade de Gerenciamento: Conforme o número de feature flags aumenta, também cresce a complexidade para gerenciá-las, pois torna-se difícil rastrear onde e como as flags estão sendo utilizadas devido a grande quantidade de código e condicionais necessárias para implementá-las.
- Performance: O uso de feature flags gera verificações adicionais no fluxo de execução do software. Em casos extremos, se não forem bem implementadas, podem afetar drasticamente a performance da aplicação.
- Segurança: Podem acidentalmente expor funcionalidades que não pertencem a determinado cliente, o que pode representar riscos de segurança.
- Testes: Testar todas as combinações possíveis de feature flags ativadas e desativadas pode se tornar um desafio, aumentando o risco de fluxos inesperados e erros no sistema.
- Dependência de Serviços Externos: Se a gestão de feature flags for feita por meio de um serviço de terceiros ou API, existe a dependência desse serviço. Problemas de disponibilidade ou desempenho desses serviços irão impactar a aplicação.

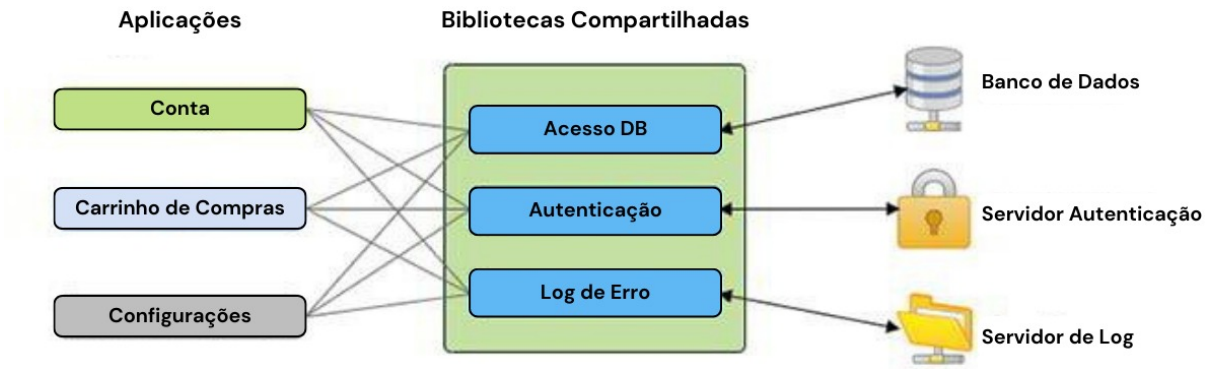
É possível concluir que as feature flags oferecem flexibilidade para testar e lançar funcionalidades de maneira controlada e em tempo real, porém elas também podem aumentar a complexidade, performance e comprometer até mesmo a segurança se não forem gerenciadas adequadamente, sendo necessário um esforço muito maior no planejamento e preparo da equipe para que tudo ocorra da maneira desejada.

2.7.3 BIBLIOTECAS COMPARTILHADAS

Bibliotecas de software são conjuntos de funcionalidades e código pré-escritos que podem ser reutilizados em diferentes partes de uma mesma aplicação ou entre diferentes projetos, sem a necessidade de reescrita do código. O uso de bibliotecas é uma prática comum para promover a reutilização de código, evitar redundâncias e facilitar a manutenção (DIALECTICA, 2023).

No contexto da criação de variantes de produto, as bibliotecas podem ser utilizadas para permitir que os programadores compartilhem funcionalidades comuns entre os projetos, enquanto personalizam e adaptam aspectos específicos para cada variante. Por exemplo, como podemos observar na Figura 7, uma biblioteca pode fornecer um conjunto padrão de operações de entrada que é utilizado em várias variantes, porém a lógica de negócios pode ser desenvolvida separadamente para cada uma. Ao isolar funcionalidades em bibliotecas dessa forma, como é apontado por Batory *et al.* (1993), os desenvolvedores podem atualizar uma única funcionalidade em todos os produtos que dependem dessa biblioteca, melhorando a consistência e reduzindo o esforço de manutenção.

Figura 7 – Exemplo de fluxo de uso de bibliotecas compartilhadas



Fonte: Morgan (2019)

2.7.4 BIBLIOTECAS VS BRANCHING

Por meio das definições apresentadas por Dialectica (2023), é possível levantarmos algumas desvantagens em relação ao uso de branching na criação de variantes de produto.

- Gestão de Dependências: Ao usar bibliotecas, a gestão de dependências pode se tornar complexa, principalmente quando as diferentes variantes do produto necessitam de diferentes versões de uma mesma biblioteca. Isso pode acarretar em problemas de compatibilidade e dificuldades na manutenção do código.
- Acoplamento e Modularidade: Bibliotecas tendem a promover um acoplamento mais forte entre diferentes partes do código, o que pode dificultar a modularidade. Isso pode tornar o processo de desenvolvimento mais rígido, especialmente se diferentes variantes do produto tiverem requisitos divergentes que exijam modificações na biblioteca.
- Reutilização de Código: Embora as bibliotecas tenham como principal objetivo a reutilização de código, em alguns casos, pode ser desafiador adaptar uma biblioteca existente para atender às necessidades específicas de uma variante de produto sem modificar a biblioteca. Isso pode acarretar em um código complexo para tentar atender a todas as demandas em um mesmo lugar ou até mesmo resultando na duplicação do código.
- Integração e Teste: Testar diferentes variantes do produto pode se tornar mais complicado. Cada um dos sistemas pode exigir uma configuração específica das bibliotecas, resultando em possíveis complicações na automação dos testes e até mesmo na integração contínua.
- Atualizações e Manutenção: Manter e atualizar bibliotecas pode ser mais trabalhoso do que gerenciar branches em um sistema de controle de versão. Pois é necesário garantir que as atualizações não quebrem outras funcionalidades em softwares que as utilizam.

- Curva de Aprendizado: Embora as bibliotecas sejam projetadas para simplificar o desenvolvimento, às vezes é necessário investir tempo para aprender a desenvolver e projetar corretamente.

Pode-se concluir que embora bibliotecas compartilhadas ofereçam uma boa maneira de reutilização de código para diferentes projetos, ela pode se tornar extremamente difícil de se gerenciar e demanda uma equipe altamente qualificada para implementá-las. Já a utilização de ramificações de versionamento de código, proporciona um meio mais estável, controlado e prático para a evolução em paralelo dos múltiplos sistemas.

2.8 CUIDADOS COM O USO DE BRANCHING

Através dos estudo realizados das alternativas e de suas principais características, é possível citar alguns pontos na utilização de ramificações que merecem atenção em relação aos seus concorrentes:

- Complexidade de Gerenciamento: O gerenciamento de múltiplas branches pode se tornar complexo, especialmente em equipes grandes. É necessário um esforço e coordenação para sempre manter as branches sincronizadas.
- Complexidade do Merge: Se as variantes de produtos se distanciarem muito em relação as suas funcionalidades principais e exigirem alterações constantes, o processo de merge fica propenso a conflitos, exigindo um maior esforço para resolver essas divergências.
- Equipe: O gerenciamento de versões pode se tornar complexo com múltiplas branches, exigindo uma clara definição no processo de integração de novas funcionalidades e correção de problemas. A equipe deve estar ciente de quando e quais branches devem ser modificadas de acordo com cada tarefa.

2.9 DEVOPS

Mishra e Otaiwi (2020) definem DevOps como uma metodologia que auxilia na comunicação entre os desenvolvedores de software e profissionais de TI operacionais, permitindo integrar automação, testes, infraestrutura, liberação de versão e qualquer outra etapa que vise dar suporte na construção de um sistema. O autor ainda pontua que os DevOps têm como principal objetivo facilitar as operações de desenvolvimento, resultando em uma maior frequência e qualidade das entregas.

Ao relacionar DevOps com a ideia do trabalho de criar branches para cada variante de produto, podemos ver uma aplicação direta dos seus princípios. Essa operação pode melhorar significativamente o fluxo de desenvolvimento, conforme destacado por Ebert *et al.* (2016),

segue algumas das principais características do DevOps que se relacionam com as vantagens na utilização de branching:

- Flexibilidade na Implementação: Branching permite que diferentes equipes desenvolvam e testem suas variantes em paralelo, sem interferir umas nas outras, o que é alinhado com os princípios de DevOps de colaboração e eficiência operacional.
- Automação de Testes: DevOps promove a automação de testes, que pode ser facilmente implementada para cada branch, garantindo que cada variante atenda as suas específicas regras de negócio e funcionalidades únicas.
- Entrega Contínua: Ao utilizar branching para cada variante de produto, a entrega contínua pode ser configurada para implementar automaticamente cada uma dessas variantes em ambientes de teste ou de produção, facilitando assim, o lançamento rápido e controlado de cada branch.
- Gestão de Configuração e Versionamento: O uso de branching permite uma gestão mais eficiente e granular das configurações e versões de cada variante de produto.
- Resiliência e Tolerância a Falhas: O DevOps busca promover a resiliência e a tolerância a falhas nos sistemas de software. Ao separar as variantes de produto em branches distintas, é possível isolar falhas e problemas em uma única variante, garantindo que outras variantes não sejam afetadas e facilitando a recuperação rápida em caso de falhas.
- Escalabilidade de Projetos: À medida que os projetos crescem e mais variantes são desenvolvidas, o branching permite escalar o desenvolvimento e estrutura de servidores de forma paralela para cada produto.

2.10 INTEGRAÇÃO CONTÍNUA (CI) E ENTREGA CONTÍNUA (CD)

Integração Contínua (CI) e Entrega Contínua (CD) são práticas essenciais utilizadas pelos DevOps no desenvolvimento moderno de software que visam melhorar a qualidade do software e a eficiência do processo de desenvolvimento.

Soares *et al.* (2022) define que a Integração Contínua (CI) ocorre quando um membro de uma equipe de desenvolvimento integra as suas alterações numa base de código juntamente com as alterações dos seus colegas. Cada uma destas integrações é verificada por uma construção automatizada para detectar erros de integração o mais rapidamente possível. Esta abordagem têm como objetivo reduzir o risco de atrasos na entrega, facilitar o esforço na integração e permitir práticas que tornem a base de código mais sucetível para um rápido aperfeiçoamento com novas funcionalidades.

Por outro lado, a Entrega Contínua (CD) é definida por Humble e Farley (2010) como uma extensão da CI que busca assegurar que o software esteja sempre em um estado que pode

ser lançado em produção. A CD automatiza o processo de liberação de software, incluindo build, testes e publicação (deploy), permitindo que o código validado pela CI seja implementado rapidamente em ambientes de produção ou pré-produção. O principal objetivo da CD é permitir que os desenvolvedores liberem alterações de código de forma segura e frequente, proporcionando um ciclo de feedbacks rápidos e entregando valor contínuo ao cliente. A automação do processo de deploy também reduz possíveis erros manuais, além da redução do tempo de recuperação, ou seja, se um problema for detectado em produção, as ferramentas de CD permitem voltar rápido para uma versão anterior estável (CHEN, 2015).

2.10.1 DESAFIOS DA INTEGRAÇÃO E ENTREGA CONTÍNUA NA UTILIZAÇÃO DE BRANCHES

Um dos principais desafios é o gerenciamento de múltiplas branches ativas ao mesmo tempo. Cada variante de produto pode ter suas próprias especificações e requisitos de teste, o que pode complicar a configuração e a manutenção dos pipelines de CI e CD. Garantir que cada branch esteja sempre em um estado de aprovada, com os testes passando consistentemente, pode exigir um certo esforço da equipe para resolver conflitos de merge e corrigir problemas específicos de cada variante. No entanto, com uma boa estratégia de gerenciamento de configuração, automação adequada e práticas de desenvolvimento colaborativas, é possível superar esses desafios e implementar pipelines de CI/CD eficazes em diferentes branches para suportar variantes de produto.

Um dos aspectos que facilita o gerenciamento de CI/CD em múltiplas branches é o fato de que o núcleo do código permanece similar entre as variantes. Isso significa que muitas das funcionalidades e correções desenvolvidas podem ser replicadas entre as diferentes branches, reduzindo assim, o esforço necessário para desenvolver e manter características comuns. Ter um núcleo de código similar também simplifica o processo de integração contínua, pois as alterações que funcionam em uma variante têm uma alta probabilidade de serem compatíveis com outras variantes, desde que respeitem os limites definidos pelas especificações individuais de cada variante.

A flexibilidade na liberação de versões também é um fator benéfico na utilização de branches para variantes de produto. Cada variante pode ter seu próprio ciclo de liberação, permitindo que funcionalidades sejam liberadas para diferentes clientes ou mercados de acordo com suas necessidades e cronogramas específicos. Essa abordagem pode aumentar a satisfação do cliente e a competitividade no mercado, ao proporcionar um produto mais adaptado às demandas específicas de cada segmento. Além disso, a separação de branches facilita o isolamento de problemas, ou seja, se um erro estiver ocorrendo em uma branch específica, ele pode ser resolvido sem impactar nas outras versões do produto.

2.11 CONSIDERAÇÕES SOBRE O CAPÍTULO

Este capítulo abordou os conceitos e estratégias no uso dos monorepos em comparação com outras abordagens populares no mercado. Foi possível constatar que são utilizadas até mesmo em grandes empresas, como a Google, Facebook e Microsoft, que optaram por estratégias de centralização do código-fonte mesmo precisando criar ferramentas próprias que se adaptassem melhor a sua realizade. Com base nisso, é possível afirmar que a abordagem sugerida neste trabalho, que combina o uso de branching com monorepos, alinha-se de maneira muito próxima às práticas e benefícios observados na estrutura interna de gerenciamento de projetos dessas empresas líderes de mercado. Essa semelhança evidencia que integrar branching em um sistema de monorepo, para determinados cenários, pode oferecer vantagens significativas em termos de eficiência operacional e flexibilidade no desenvolvimento de software, semelhantes às alcançadas por algumas das empresas mais inovadoras do mundo.

3 ESTUDO DE CASO: IMPLEMENTAÇÃO DE MONOREPO NAS EMPRESAS ETERNO SOFTWARE E DIVINO TICKETS

Este capítulo explora a forma como foram utilizadas as estratégias de monorepos e *branching* nas empresas Eterno Software e Divino Tickets, que enfrentaram o desafio de gerenciar diferentes variantes de produto e utilizaram dessa abordagem para resolver seus problemas de gerenciamento. Também será discutido o contexto operacional de cada empresa, a estrutura adotada e os resultados alcançados com essa abordagem.

3.1 ETERNO SOFTWARE

A Eterno Software é uma empresa que oferece soluções de software para gestão empresarial, atendendo a diversos setores, incluindo saúde e jurídico. Diante da complexidade e das particularidades dessas áreas, a empresa enfrentava o desafio de desenvolver e manter sistemas de gestão que atendessem às necessidades únicas de cada área, ao mesmo tempo em que compartilhava funcionalidades comuns e genéricas entre elas.

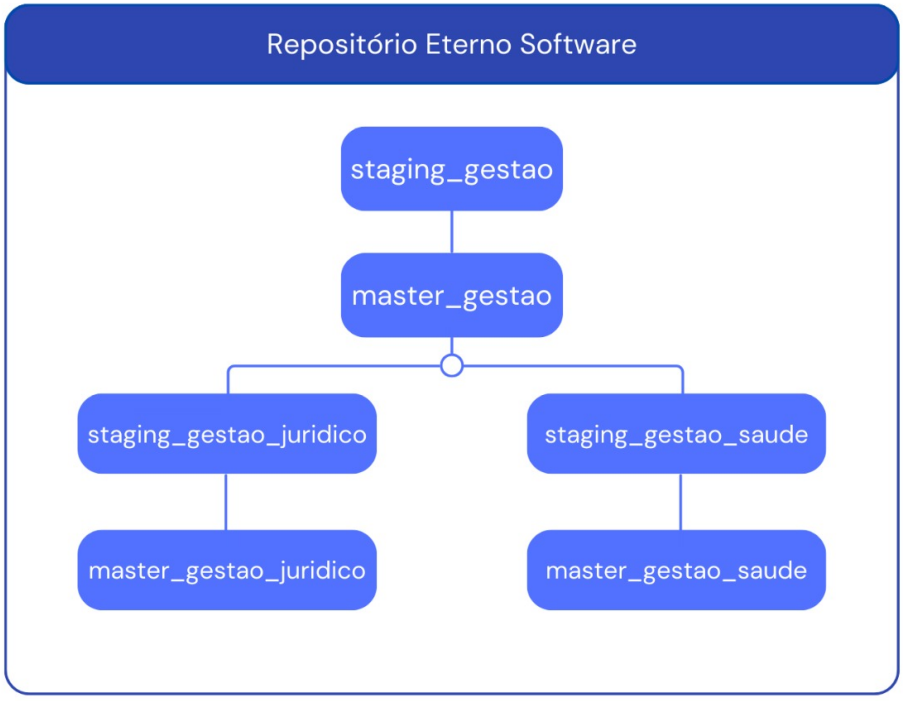
A empresa possui uma arquitetura em camadas utilizando .NET Core no backend e Vue.js no frontend. No backend, a estrutura de pastas é organizada em Domain (entidades, interfaces de repositórios, serviços e exceções), Infrastructure (repositórios, configuração de banco de dados e serviços externos), Application (casos de uso, DTOs e mapeamentos) e API (controladores, modelos de requisição/resposta e filtros). Utiliza algumas boas práticas como injeção de dependência, configuração centralizada e tratamento de erros. No frontend, a estrutura inclui Components (componentes reutilizáveis), Views (componentes de páginas), Store (gerenciamento de estado com Vuex), Router (configuração de rotas), Services (comunicação com a API) e Assets (arquivos estáticos). A integração entre backend e frontend envolve autenticação JWT, configuração CORS e documentação de API com Swagger.

Para lidar com a complexidade das diferentes áreas e garantir uma melhor organização e manutenção dos projetos de gestão, a empresa optou por utilizar monorepo e implementar estratégias de *branching*. Foram criadas branches separadas para os projetos de gestão da área da saúde e da área jurídica. Cada uma representa um conjunto específico de funcionalidades e customizações relacionadas à área de negócio correspondente.

Conforme é possível observar na Figura 8, a branch principal ("master_gestao") foi definida como o projeto de gestão mais genérico, contendo as funcionalidades comuns e compartilhadas entre as áreas da saúde e jurídica. Os ramos filhos foram criados a partir da branch principal para representar os projetos específicos de gestão de cada área, mantendo a hierarquia e a integridade do código-fonte. Essa mesma estrutura foi utilizada para separar backend e frontend, cada uma em seu devido repositório. Tambem vale ressaltar a utilização de bran-

ches staging, que são utilizadas como base de testes antes de serem mescladas (merge) as suas respectivas branches masters, correspondentes a versão final de produção.

Figura 8 – Estrutura do repositório da empresa Eterno Software



Fonte: O Autor

3.1.1 RESULTADOS E BENEFÍCIOS

Segundo a equipe de desenvolvimento da empresa Eterno, a implementação de *branching* proporcionou diversos benefícios, incluindo:

- Isolamento de Desenvolvimento: Cada branch funciona como um ambiente isolado, permitindo que equipes distintas desenvolvam e testem suas funcionalidades específicas sem interferir no trabalho uma das outras. Isso permitiu que além do código-fonte, a empresa também dividisse as equipes de desenvolvimento, comercial e suporte. Essa segmentação é fundamental para focar os esforços e habilidades nas áreas que demandam conhecimento específico.
- Particularidades: A separação em branches permite que cada equipe se concentre exclusivamente nas demandas específicas de seu respectivo campo. Por exemplo, a área da saúde preza pela confidencialidade e acompanhamento de seus pacientes, enquanto no jurídico existe uma maior preocupação em manter as movimentações dos processos eletrônicos dos tribunais sempre atualizados.

- Reutilização de Código: A branch master gestão contém o código genérico que é comum a ambos os projetos, facilitando a reutilização de código. Isso não apenas reduz o esforço de desenvolvimento, mas também minimiza a duplicação de código e potenciais erros.

- Expansão Facilitada no Mercado: A branch principal também é utilizada pela empresa para atender clientes que procuram sistemas de gestão fora das áreas de saúde e jurídica. Isso não só fortalece a posição da empresa no mercado, como também facilita a criação de novas variantes de produtos. Para adaptar o sistema a um novo segmento de mercado, basta derivar uma nova branch da principal e incorporar funcionalidades específicas à nova área de atuação. Esta abordagem permite uma expansão rápida, segura e eficiente, adaptando-se às necessidades do mercado com agilidade.

- Facilidade de Integração e Testes: O uso de branches facilita a integração contínua e testes automatizados em cada contexto específico. Alterações podem ser integradas e testadas na branch relevante antes de serem atualizadas na sua devida master, garantindo que todas as funcionalidades sejam adequadamente validadas de maneira isolada e mais segura.

3.1.2 DESAFIOS

Abaixo são listados alguns dos desafios encontrados no dia a dia com a utilização de branches, segundo a equipe de desenvolvido da Eterno Software:

- Treinamento e Conhecimento da Equipe: Um dos principais desafios é garantir que todos os membros da equipe estejam adequadamente treinados e informados sobre como utilizar o sistema de branches efetivamente. Cada desenvolvedor deve saber exatamente em qual branch trabalhar, dependendo da funcionalidade ou do erro que está sendo tratado.

- Gestão e Coordenação de Merges: Outro desafio é a necessidade de definir um responsável para gerenciar o processo de merge das branches. Em determinados momentos pode exigir um certo nível de coordenação e supervisão para garantir que as integrações sejam feitas sem erros e que todas as funcionalidades e correções sejam incorporadas de forma correta.

- Conflitos Durante Merges: Os conflitos de merge são inevitáveis, especialmente em um ambiente onde múltiplas branches estão ativas ao mesmo tempo. Conflitos podem ocorrer quando duas branches modificaram a mesma parte do código de maneira diferente. Resolver esses conflitos pode ser um processo que exige uma boa compreensão das alterações feitas.

- Configuração Individual de Integração Contínua: A configuração de integração contínua (CI) e testes automatizados para cada branch específica representou outro desafio para a empresa Eterno. Implementar uma estratégia robusta de CI que seja eficiente e eficaz

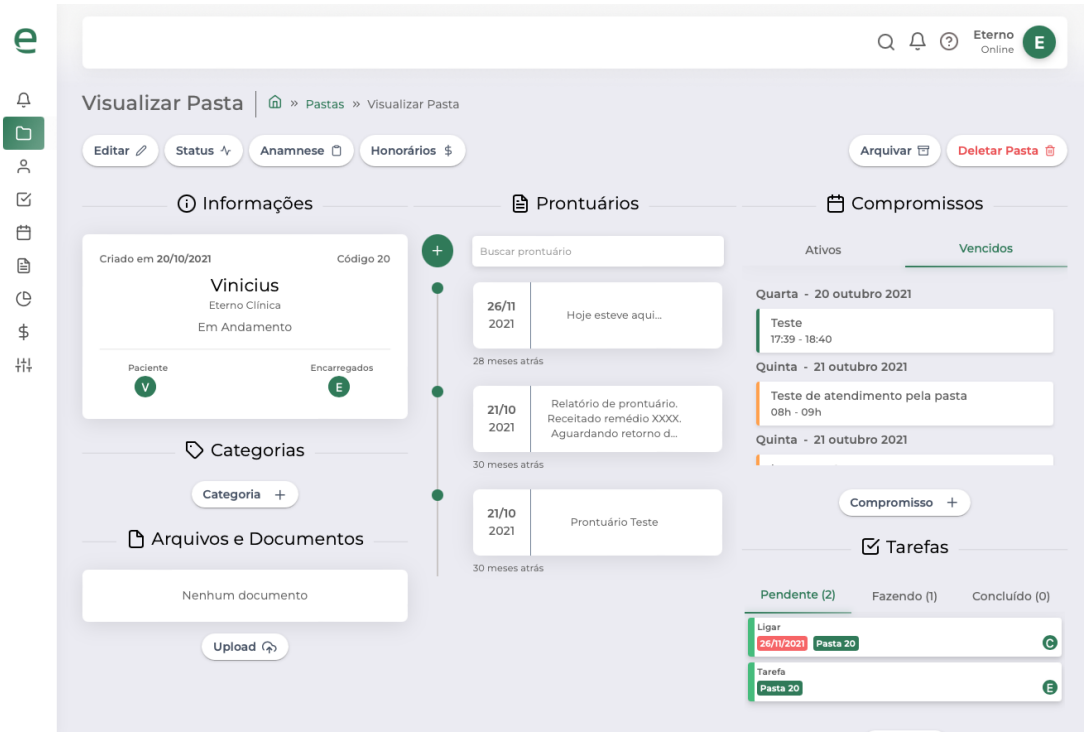
para cada variante de produto em suas respectivas branches exigiu um certo nível de planejamento. Porém o resultado final garantiu que após o merge entre as branches, as mudanças sejam automaticamente construídas, testadas e prontas para rodar de forma rápida e eficiente.

3.1.3 EXEMPLOS PRÁTICOS

Nesta seção, demonstraremos visualmente como as estratégias de *branching* permitiram a customização efetiva das interfaces de usuário para três variantes do produto da empresa Eterno, cada uma atendendo a necessidades distintas de segmentos de mercado: Eterno Saúde, Eterno Jurídico e Eterno Gestão.

A interface do Eterno Saúde, que refere-se a branch "master_gestao_saude", é projetada com foco na gestão de pacientes. Na tela da pasta de um cliente, conforme pode-se observar na Figura 9, é incorporado um painel dedicado ao prontuário do paciente, que inclui informações médicas, histórico de consultas e tratamentos. Esta interface utiliza uma paleta de cores verde, que são escolhidas por sua associação com saúde e tranquilidade. Os termos utilizados são específicos da área médica, como "Prontuário", "Anamnese"e "Paciente".

Figura 9 – Tela do módulo pasta na branch master saúde

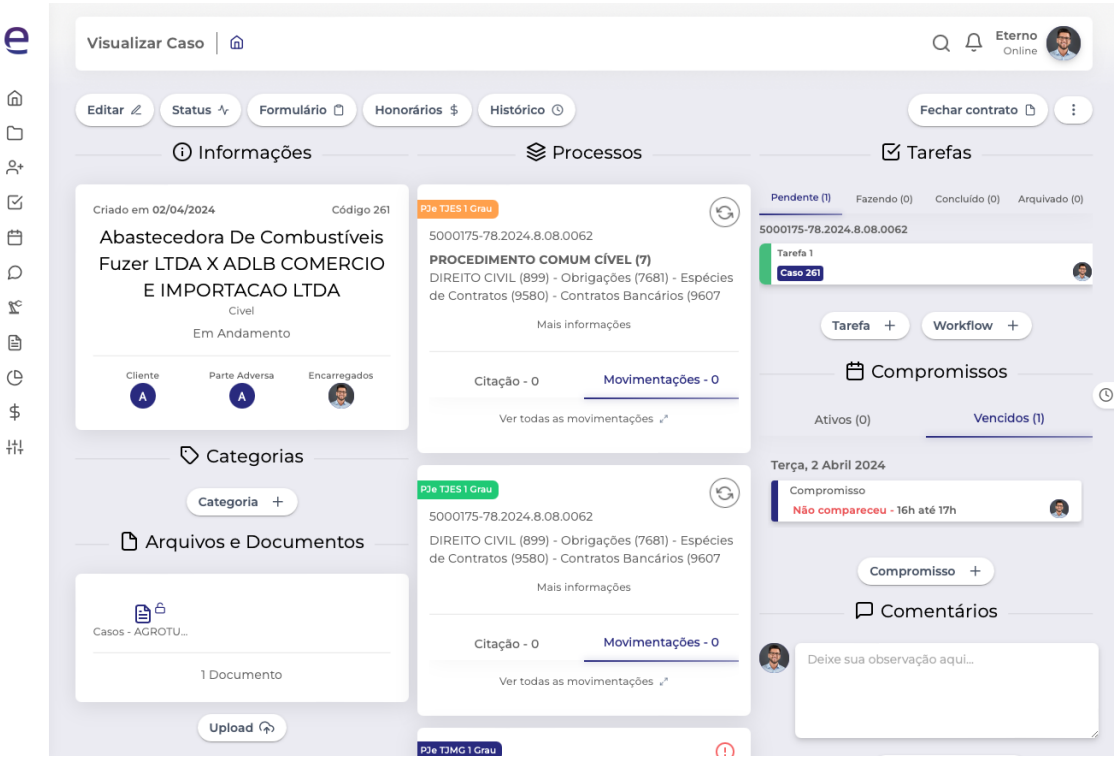


Fonte: Eterno Software Saúde

Para o Eterno Jurídico, a tela da Figura 10 da pasta do cliente (chamada de caso nessa ramificação) apresenta um painel detalhado com processos e movimentações eletrônicas dos tribunais, funcionalidades que só existem nessa branch. Este design é adaptado para facilitar o

acesso rápido a informações críticas legais, usando uma paleta de cores azul escuro, que transmitem seriedade e profissionalismo. A nomenclatura é igualmente especializada, com termos como "Processos", "Movimentações Judiciais", "Caso"e "Honorário".

Figura 10 – Tela do módulo pasta na branch master jurídico



Fonte: Eterno Software Jurídico

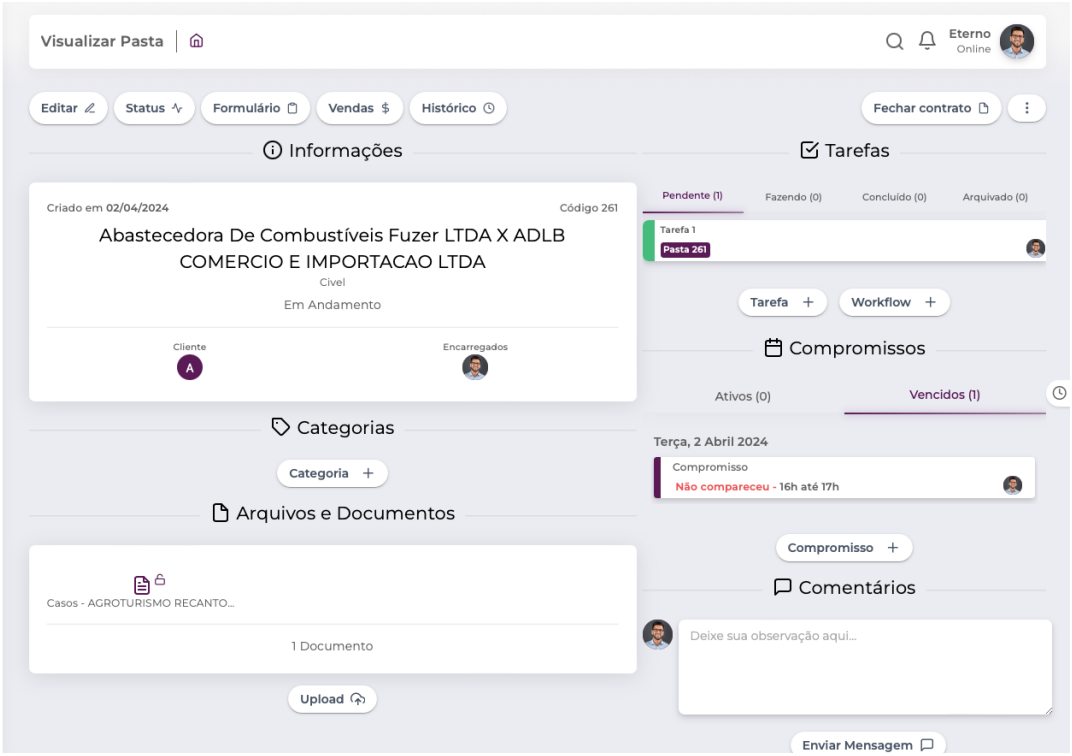
A variante Eterno Gestão, conforme apresentado na Figura 11, é voltada para a administração e gestão empresarial, possui uma interface mais limpa na tela da pasta do cliente, sem painéis específicos de prontuários ou processos judiciais. A interface é visualmente simplificada para evitar a sobrecarga de informações e torná-lo um sistema mais flexível ao uso, utilizando uma paleta de cores roxa. Os termos aqui são genéricos e focados em gestão, como "Formulário", "Venda"e "Pasta".

Esses exemplos práticos mostram como o uso de branches permite à Eterno adaptar suas interfaces para atender de maneira eficiente às expectativas e necessidades dos diferentes tipos de clientes. Esta abordagem não só melhora a usabilidade e a satisfação do usuário, mas também reforça a identidade e a especialização de cada produto do Eterno.

3.1.4 CONCLUSÃO

A estratégia de *branching* adotada pela Eterno Software provou ser uma maneira eficiente de gerenciar os diferentes nichos de sistemas oferecidos, permitindo um desenvolvimento isolado para cada segmento de mercado e ao mesmo tempo permite reaproveitamento do código-fonte em correções e até novas funcionalidades. Embora esta abordagem ofereça

Figura 11 – Tela do módulo pasta na branch master gestão



Fonte: Eterno Software Gestão

melhorias significativas na eficiência operacional e na capacidade de resposta rápida das especificações únicas de cada produto, ela também impõe desafios relacionados à complexidade de gerenciar múltiplas branches ativas e à necessidade de coordenação intensiva entre as equipes para manter a consistência e a integridade do código em todas as variantes. Portanto, enquanto a estratégia de *branching* apresenta vantagens consideráveis, ela requer uma maior atenção por parte da gestão para minimizar os riscos associados.

Conclui-se ser uma abordagem superior em comparação com o uso de múltiplos repositórios, feature flags ou bibliotecas compartilhadas para gerenciar variantes de produto. Abordagens como múltiplos repositórios fragmentariam o ambiente de desenvolvimento, complicando a atualização e a consistência do código entre os diferentes produtos. Feature flags, apesar de muito úteis para configurar funcionalidades específicas, resultariam em complexidade e dificuldades na manutenção a longo prazo, pois muitas das telas e regras de negócio exigiriam excessivas condicionais para cada variante. Bibliotecas compartilhadas, embora promovam a reutilização de código e a eficiência, exigem uma gestão meticulosa de versões e dependências, que se tornaria complexo em um ambiente de desenvolvimento em constante mudança como o da Eterno. Além disso, a necessidade de gerenciar essas dependências de forma eficaz necessitaria de mais tempo dedicado, aumentando os custos operacionais e extendendo os cronogramas de desenvolvimento. Portanto, a escolha do *branching*, com seu equilíbrio entre flexibilidade, controle e rapidez, revela-se a melhor estratégia para a empresa Eterno, maximizando a eficá-

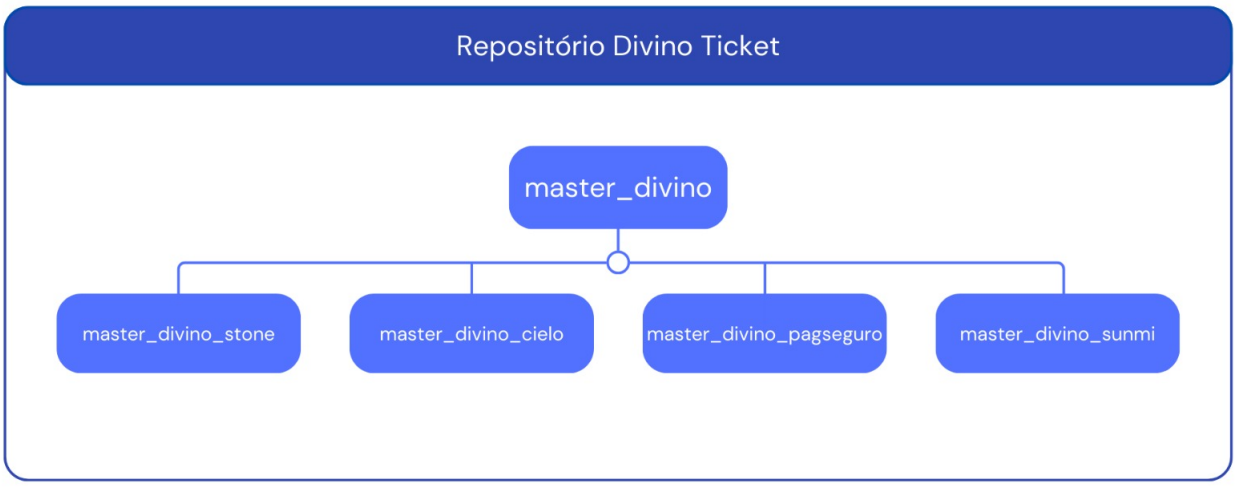
cia do desenvolvimento enquanto minimiza complicações operacionais, características fundamentais para uma empresa relativamente nova no mercado e com constante alteração em seus produtos.

3.2 DIVINO TICKET

A Divino Ticket é um aplicativo para Android de venda de ingressos online que oferece serviços de gestão e vendas de ingressos para eventos. Um dos principais desafios que a empresa enfrentava estava relacionado à integração com as máquinas de cartão de diferentes operadoras, onde as integrações precisavam ser cuidadosamente separadas para garantir a aprovação em cada loja de operadora, que proíbem o uso de bibliotecas ou códigos de máquinas concorrentes.

Para superar essas barreiras e otimizar a integração com diversas máquinas de impressão, a Divino Ticket adotou uma abordagem de *branching* que contém branches específicas para cada máquina de cartão (Stone, Cielo, PagSeguro e Sunmi), conforme é possível observar na Figura 12. Esta seção explora as vantagens e desafios dessa estratégia, destacando como ela facilita a personalização e a manutenção do aplicativo, ao mesmo tempo que cumpre as exigências regulatórias das plataformas de máquinas de cartão.

Figura 12 – Estrutura do repositório da empresa Divino Ticket



Fonte: O Autor

3.2.1 RESULTADOS E BENEFÍCIOS

Segundo a equipe de desenvolvimento da empresa Divino Ticket, a implementação de *branching* proporcionou diversos benefícios, incluindo:

- Aprovação em Lojas de Operadoras: A separação das integrações com as máquinas de cartão permitiu que a empresa adaptasse as integrações de acordo com as exigências específicas de cada loja de operadora, facilitando a aprovação em todas elas.
- Conformidade e Segurança: Cada branch pode ser desenvolvida e testada de forma independente em cada máquina de cartão, garantindo que o aplicativo esteja em conformidade com as diretrizes das lojas de aplicativos e não viole as restrições de uso de bibliotecas concorrentes. Cada máquina possui o seu próprio layout de impressão de ingressos e de relatórios, além disso, o fluxo de pagamento de cartão também precisou ser adaptado para cada uma individualmente.
- Manutenção Individual de cada Máquina: *branching* permite que as atualizações, correções e melhorias sejam implementadas rapidamente e de maneira isolada, sem interferir nas funcionalidades de outras máquinas. Isso não só acelerou o ciclo de desenvolvimento mas também simplificou a manutenção do aplicativo ao longo do tempo.

3.2.2 DESAFIOS

Segundo a equipe de desenvolvimento da empresa Divino Ticket, a implementação de *branching* resultou em alguns desafios:

- Reestruturação do código: Foi necessário para a empresa contratar algumas horas de desenvolvimento para que fosse possível desaclopar as integrações de máquinas já existentes e separá-las em branches.
- Sincronização e Consistência: Por se tratar de uma empresa que não possui uma equipe de desenvolvedores fixa, exigiu um treinamento e explicação detalhada para o responsável da empresa, pois seria ele que repassaria essas informações aos desenvolvedores autonomos e atualizaria nas branches filhas quando necessário.

3.2.3 EXEMPLOS PRÁTICOS

Nesta seção, demonstraremos através de imagens como a utilização de branches permitiram a customização própria no layout de impressão para cada máquina de cartão que foi integrada ao aplicativo da Divino.

A Figura 13 ilustra a integração entre duas ferramentas cruciais para pagamentos na Divino Ticket. No lado esquerdo da imagem, temos a representação da máquina da Sunmi, enquanto no lado direito está a máquina de cartão da Cielo.

Figura 13 – Impressão de ingresso na máquina da Sunmi x Cielo

Fonte: Divino Ticket

Nota-se que houveram adaptações em ambas as máquinas para garantir uma integração harmoniosa e eficiente. Tanto a máquina da Sunmi quanto a da Cielo precisaram passar por ajustes em seu layout de impressão, esses ajustes foram essenciais para que as informações fossem exibidas de maneira clara e legível, conforme as diretrizes e documentação fornecidas por cada empresa. Na máquina Sunmi, até mesmo o tamanho do ingresso precisou ter seu tamanho reduzido e fontes alteradas.

Também é possível observar que a Figura em questão representa não apenas a integração técnica entre a máquina da Sunmi e a da Cielo, mas também as adaptações necessárias tanto no layout de impressão quanto na paleta de cores para garantir uma experiência consistente e alinhada com as diretrizes de cada uma. Essas adaptações foram fundamentais para garantir que

o produto final atendesse às expectativas dos usuários e fosse aprovada na loja de aplicativos de cada uma delas.

Já a Figura 14 apresenta a integração da Divino Ticket com a máquina PagSeguro, especificamente o modelo Moderninha Smart 2. Nesta integração, foram realizadas diversas adaptações, incluindo alterações na fonte e nos espaçamentos entre as informações exibidas. Além disso, foi necessário incluir uma mensagem no rodapé informando ao cliente que "Essa impressão não é responsabilidade da PagSeguro". Todas essas modificações foram essenciais para seguir as diretrizes estabelecidas pela empresa PagSeguro e garantir a aprovação do aplicativo em sua loja virtual.

É importante destacar também que a separação do código em branches foi crucial para evitar qualquer vestígio de importação de bibliotecas de máquinas concorrentes. Essa medida foi necessária para evitar rejeições e impedimentos ao aplicativo durante o processo de submissão nas lojas das empresas de máquinas de cartão.

Essas adaptações e precauções demonstram o cuidado e a atenção necessários para garantir uma integração bem-sucedida com as máquinas de pagamento, respeitando as políticas e diretrizes das empresas envolvidas e garantindo uma experiência fluida e autêntica aos usuários finais.

A Figura 15 retrata a integração com o totem da empresa PagSeguro, uma ferramenta essencial para processamento de pagamentos em diversos estabelecimentos comerciais. Uma característica muito importante desta integração é a necessidade de realizar adaptações no layout devido ao tamanho da tela do totem em comparação as outras máquinas de cartão.

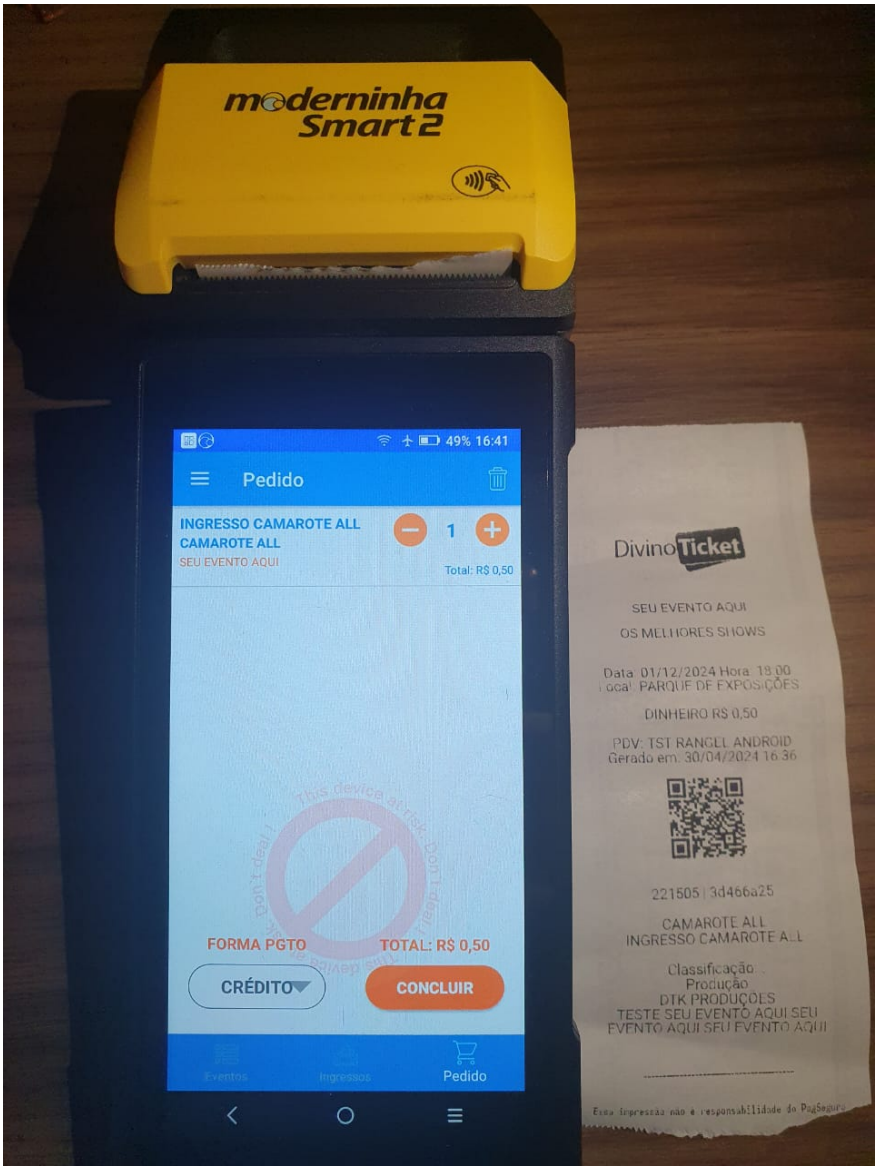
As alterações no layout foram necessárias para garantir que as informações exibidas sejam apresentadas de forma clara e legível, levando em consideração as dimensões específicas da tela do totem. Essas adaptações incluíram ajustes na disposição dos elementos e no tamanho da fonte.

Além disso, é importante mencionar que as alterações foram necessárias principalmente porque o usuário do aplicativo seria o cliente final que comprou o ingresso, e não mais os vendedores treinados para utilizar o aplicativo da Divino Ticket. Portanto, uma série de ajustes precisou ser realizada para facilitar a usabilidade e proporcionar uma experiência intuitiva para os clientes. Como por exemplo: a simplificação de menus, botões e outras interações que garantem que os clientes possam realizar seus pagamentos de forma rápida e sem complicações, independente do seu nível de conhecimento em tecnologia. Essas medidas foram essenciais para proporcionar um serviço de pagamento confiável e conveniente para os usuários finais.

3.2.4 CONCLUSÃO

A implementação de estratégias de *branching* na Divino Ticket, para gerenciar diferentes integrações de máquinas de pagamento, demonstrou grandes vantagens em relação ao uso

Figura 14 – Impressão de ingresso na máquina da Pagseguro

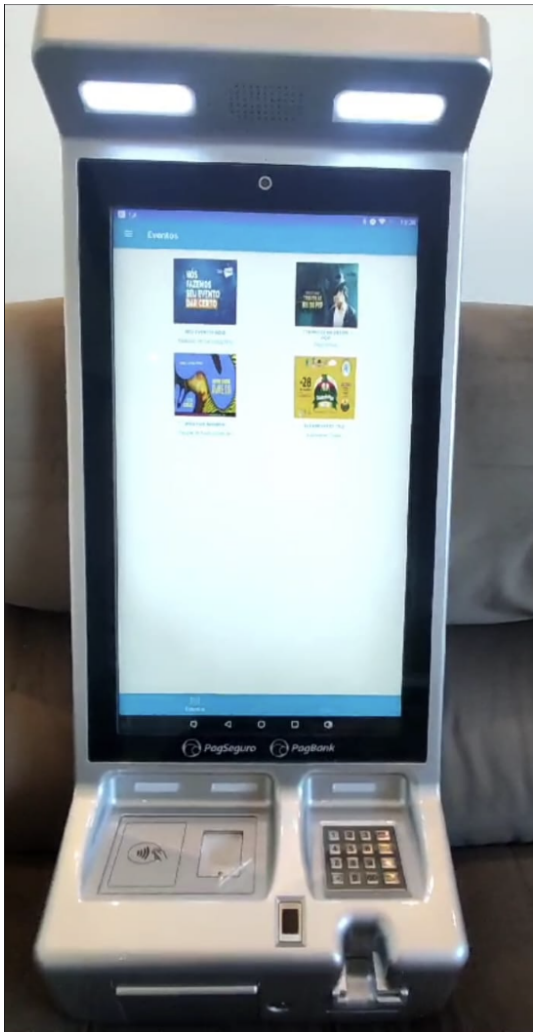


Fonte: Divino Ticket

de bibliotecas compartilhadas, múltiplos repositórios e feature flags. Bibliotecas compartilhadas necessitam de uma gestão complexa de dependências e planejamento para evitar conflitos em um aplicativo que demanda diferentes customizações pra cada máquina, tornando um processo muito custoso e difícil para essa empresa que é composta apenas por profissionais autonomos na área de desenvolvimento. Já múltiplos repositórios tornariam trabalhoso qualquer alteração que precisasse ser replicada em todos os projetos, resultando em um maior custo de horas para a empresa. Por outro lado, feature flags não ofereceriam o isolamento necessário para atender às políticas das lojas de aplicativos que proíbem a inclusão de códigos de concorrentes. O *branching*, por outro lado, permitiu o isolamento e desenvolvimento paralelo, mantendo uma base de código unificada que facilita a atualização e a sincronização das melhorias e correções.

O uso de *branching* proporcionou à Divino Ticket a flexibilidade necessária para cum-

Figura 15 – Totem da Pagseguro



Fonte: Divino Ticket

prir regulamentações e adaptar-se às especificações de cada dispositivo de pagamento de forma segura. Esta abordagem não apenas suporta a customização eficaz de suas impressões e rotinas de pagamento, mas também a rápida implementação de atualizações por desenvolvedores autonomos, garantindo que as integrações entre as máquinas possam ser feita paralelamente por esses profissionais e as melhorias comuns consigam ser propagadas entre todas as branches sem comprometer a integridade do sistema.

3.3 UTILIZAÇÃO DE *BRANCHING* NA PRÁTICA

Esta seção tem como objetivo fornecer um guia passo a passo sobre como criar uma estrutura de branches usando o Git para gerenciar variantes de um mesmo produto.

Para iniciar o projeto, é necessário criar um repositório para o projeto em uma plataforma de hospedagem e seguir os comandos sugeridos após sua criação. Aqui está um exemplo de comandos, extraídos do GitHub:

Algoritmo 1 – Início de um projeto utilizando Git

```
1    git init
2    git add -A
3    git commit -m "codigo inicial para o projeto base"
4    git branch -M master
5    git remote add origin git@github.com:CAMINHO_DO_REPOSITORIO
6    git push -u origin master
```

1. `git init`: Inicializa um novo repositório Git no diretório atual.
2. `git add -A`: Adiciona todas as mudanças (novos arquivos, modificações e exclusões) ao índice.
3. `git commit -m "código inicial para o projeto base"`: Cria um commit ou armazenamento com a mensagem especificada, salvando as mudanças no repositório.
4. `git branch -M master`: Renomeia a branch atual para "master".
5. `git remote add origin git@github.com:CAMINHO_DO_REPOSITORIO`: Adiciona e faz a relação entre o repositório remoto e o local.
6. `git push -u origin master`: Envia as mudanças para o repositório remoto

Após este procedimento, a branch principal (geralmente denominada de master ou main) será criada e conterá o código base do projeto. O próximo passo é criar as branches para as variantes dos produtos e atualizá-las no repositório remoto. Nessas ramificações serão mantidas as mesmas linhas de código da branch principal, com personalizações aplicadas diretamente conforme necessário.

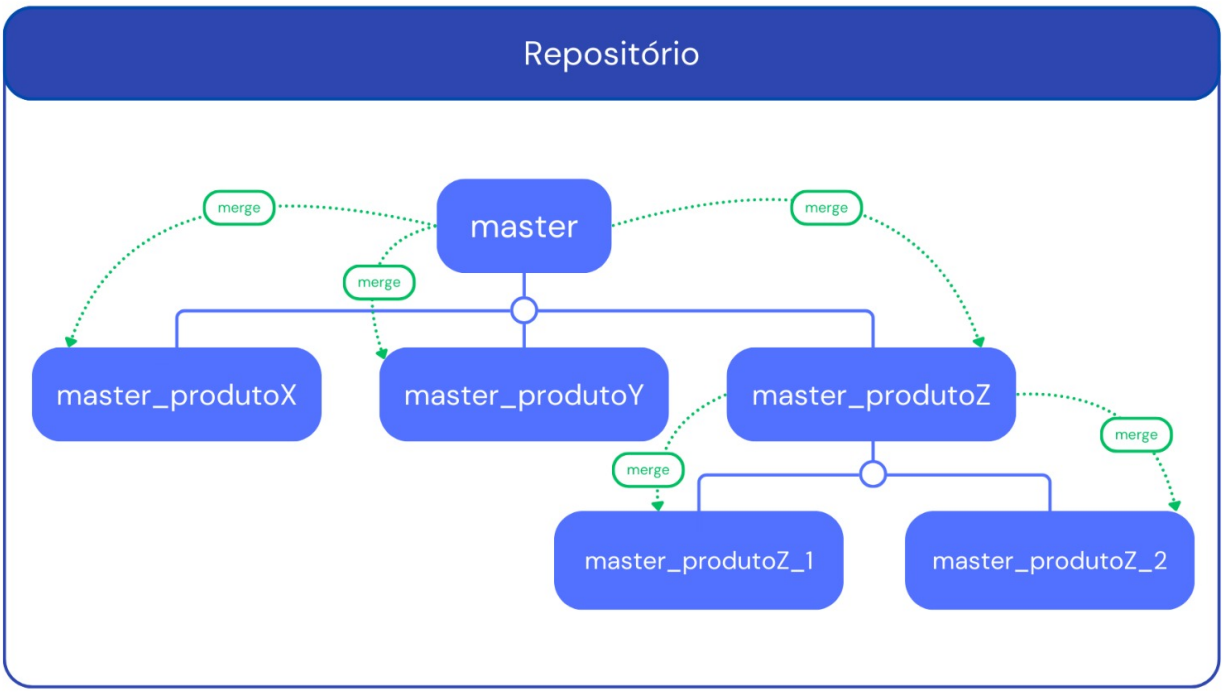
Algoritmo 2 – Alteração de branch de um projeto utilizando Git

```
1  git checkout -b master_produtoX
2  git push origin master_produtoX
```

1. `git checkout -b master_produtoX`: Cria uma nova branch chamada "master_produtoX"e altera sua localização local para essa branch.
2. `git push origin master_produtoX`: Envia a nova branch para o repositório remoto.

A partir daí, é crucial seguir o fluxo correspondente para manter as variações sempre atualizadas. Na Figura 16, é possível observar como essa estrutura funciona, onde as alterações devem ser replicadas nas branches filhas, através de operações como merge ou push no Git.

Figura 16 – Fluxo de atualização das branches



Fonte: O Autor

Após realizar alterações em uma branch, é necessário inserir os comandos Git para atualizar o repositório remoto. Por exemplo, na branch master:

Algoritmo 3 – Enviar atualização de uma branch utilizando Git

```
1 git add -A
2 git commit -m "atualizado tarefa X"
3 git push origin master
```

1. `git add -A`: Adiciona todas as mudanças (novos arquivos, modificações e exclusões) ao índice.
2. `git commit -m "atualizado tarefa X"`: Cria um commit ou armazenamento com a mensagem especificada.
3. `git push origin master`: Envia as mudanças da branch "master"para o repositório remoto.

Em seguida, os comandos abaixo são utilizados para atualizar os projetos filhos. É essencial replicá-los em todas as ramificações descendentes da branch originalmente alterada, garantindo que todos os produtos estejam com a última versão do sistema:

Algoritmo 4 – Receber atualização de uma branch utilizando Git

```
1 git checkout master_produtoX
2 git pull origin master
3 git push origin master_produtoX
```

1. `git checkout master_produtoX`: Altera a localização local para a branch "master_produtoX".
2. `git pull origin master`: Faz um merge das mudanças da branch "master"para "master_produtoX".
3. `git push origin master_produtoX`: Envia as mudanças atualizadas da branch "master_produtoX"para o repositório remoto.

Uma vez que as branches estejam atualizadas, é possível configurá-las para integração e entrega contínua, automatizando a execução de testes para garantir a confiabilidade dos projetos. Com cada branch possuindo suas próprias customizações e regras de negócio, as atualizações podem ser liberadas ou não para os clientes de forma automática e customizável, garantindo que cada variante mantenha seu fluxo de funcionamento independente e correto.

Embora este procedimento de gerenciamento de branches seja eficaz para manter variações de um produto, ele pode causar lentidão nas entregas e atualizações devido à necessidade de replicar manualmente as alterações em cada branch. Esse processo manual pode ser propenso a erros e exige um esforço significativo da equipe de desenvolvimento para garantir que todas as branches estejam sincronizadas corretamente. Além disso, o tempo necessário para realizar essas operações manualmente pode atrasar a implementação de novas funcionalidades e correções de erros, impactando negativamente a eficiência e a agilidade do desenvolvimento.

Para mitigar esses problemas e aumentar a eficiência, no próximo capítulo será apresentada uma implementação que automatiza todo esse processo. A automação garantirá que as atualizações sejam aplicadas de forma consistente e rápida a todas as branches, reduzindo o risco de erros humanos e liberando a equipe de desenvolvimento para focar em tarefas mais estratégicas.

4 AUTOMATIZANDO A ATUALIZAÇÃO DAS BRANCHES COM MONOREPO

A proposta deste trabalho consiste em desenvolver uma solução de monorepo com branches automáticas, com o intuito de otimizar e simplificar o processo de criação de variantes de produto de software de empresas como a Eterno Sofware, que necessitam de agilidade para adentrar novos nichos de mercado.

Conforme apontado na seção 2.8, uma das principais desvantagens na estratégia de utilização de branches para variantes de produto é a necessidade de gerenciar e atualizar todas elas. Com o objetivo de otimizar esse fluxo de trabalho, foi desenvolvido uma solução de automação utilizando a linguagem Python e a API oficial do GitHub. Este capítulo detalha o processo de criação e implementação dessa solução, que visa atualizar todas as branches de um único repositório a partir de uma branch principal, levando em consideração a hierarquia das branches.

4.1 DESENVOLVIMENTO DA SOLUÇÃO AUTOMATIZADA

A complexidade crescente dos projetos demandaram uma solução mais eficiente para gerenciar as múltiplas branches. O processo manual de mesclar alterações da branch principal em cada uma das branches secundárias é suscetível a erros e pode consumir uma quantidade significativa de tempo. Além disso, outro potencial problema que pode ocorrer com o processo manual é que a hierarquia das branches exige uma ordem específica de atualização para garantir a integridade do projeto.

Para efetuar o desenvolvimento da solução automatizada foi realizado um estudo das funcionalidades oferecidas pela API pública do GitHub para interagir com os repositórios e suas branches remotas. Esse estudo permitiu compreender melhor as capacidades e os métodos disponíveis para manipular os dados do GitHub de forma prática e eficiente.

Após finalizado o estudo e levantamento de requisitos, integrou-se a API do GitHub ao código-fonte do projeto, proporcionando uma série de operações-chave. Essas operações incluíram:

- Obtenção da Lista de Branches: Por meio da API, foi possível recuperar a lista completa de branches de um repositório, fornecendo informações essenciais para analisar a hierarquia e as dependências entre as branches.
- Gestão de Merge: Foi possível efetuar merges automáticos entre as branches, aplicando as alterações da branch principal em todas as branches dependentes de forma eficiente e precisa.

- Notificações e Logs: A API possibilitou receber notificações sobre o status dos merges e a geração de logs detalhados, fornecendo dados importantes sobre o andamento e a conclusão das operações. Como por exemplo, possíveis conflitos ou erros que poderiam ocorrer no processo de merge.

Entretanto, para utilizar da API do GitHub no projeto de automação desenvolvido, é necessário coletar e inserir alguns dados de cadastro que são essenciais para a autenticação com a plataforma:

- Nome do Proprietário do Repositório: Este é o nome de usuário ou organização que é o proprietário do repositório no GitHub. Este dado é crucial para identificar de forma única o local do repositório na plataforma.
- Nome do Repositório: É o nome específico do repositório no GitHub. Este identificador é necessário para direcionar as operações de API para o repositório específico em que se deseja trabalhar.
- Token de Acesso para Autenticação na API do GitHub: A autenticação é necessária para interagir com a API do GitHub. Um token de acesso fornece um método seguro para identificar e autorizar solicitações, garantindo que apenas usuários autorizados possam realizar as operações nos repositórios.
- Nome da Branch Principal: O sistema de automação requer o nome da branch que originou o projeto (geralmente chamadas de master ou main), pois as operações de merge e outras interações dependem desta informação para que executem os processos das branches na ordem hierárquica correta.

Ao reunir esses dados necessários, os desenvolvedores já estão aptos para executar o sistema desenvolvido na automatização de atualizações das branches, somente é necessário atribuí-las aos seus respectivos campos no momento da execução do programa.

Após executado o programa, o Algoritmo 5 é o primeiro a ser requisitado, com o propósito de percorrer a hierarquia das branches do projeto e determinar a sequência de sua atualização. Esse código têm como base uma nomenclatura padronizada para as branches, em que cada uma é composta pelo nome da sua branch pai seguido do caractere "_"(indicativo de extensão) e do próprio nome da branch. Por exemplo: *main* e *main_projeto1*.

Algoritmo 5 – Função em Python para construir a árvore de hierarquia das branches

```
1
2 def build_branch_tree(branches):
3     branch_tree = {}
4
5     for branch in branches:
```

```
6           branch_name = branch['name']
7           parts = branch_name.split('_')
8           current_level = branch_tree
9
10          for part in parts:
11
12              if part not in current_level:
13                  current_level[part] = {}
14
15              current_level = current_level[part]
16
17          #Guarda o nome completo da branch
18          current_level['__branch_name__'] = branch_name
19
20      return branch_tree
```

Já o método apresentado no Algoritmo 6 foi projetado para lidar com o resultado dessa construção de árvore das branches, permitindo a integração automatizada das alterações de uma branch pai em suas sub-branches. Ao receber os parâmetros necessários, como a URL base para operações de merge, os cabeçalhos de autenticação para a API do GitHub e a estrutura de árvore representando as branches, o método entra em ação. Ele itera sobre cada elemento da estrutura de árvore, examinando as relações de dependência entre as branches.

Ao encontrar uma branch pai com sub-branches, o método realiza o merge de forma automática. Para cada sub-branch, uma solicitação POST é enviada para a URL base do GitHub, contendo informações da branch de destino (onde as alterações serão mescladas) e a branch de origem (de onde as alterações serão retiradas). A mensagem de commit também é incluída para fornecer contexto e histórico sobre o merge.

Após a execução do merge, o método analisa a resposta da solicitação para determinar o resultado. Se o procedimento for bem-sucedido, uma mensagem de sucesso é exibida. Caso ocorra um conflito de merge, uma outra mensagem é exibida, alertando sobre a necessidade de resolução manual do conflito. Em caso de falha, uma mensagem de erro é emitida, indicando o código de status e detalhes adicionais sobre a resposta recebida.

Além disso, é importante ressaltar que o método implementado é recursivo, o que significa que ele pode lidar com a estrutura de árvore, criada anteriormente, de forma eficiente, permitindo a navegação entre as sub-branches e realizando merges em cada nível da hierarquia.

Algoritmo 6 – Função em Python que realiza o merge entre as branches

```
1  def merge_branch_tree(base_url, headers, branch_tree, depth=0):
2      for branch, children in branch_tree.items():
3
4          if isinstance(children, dict):
5              parent_branch = children['__branch_name__']
```

```
6
7               for child in children:
8
9                   if child != '__branch_name__':
10                      #Parametros para o merge
11                      into_branch = parent_branch + '_' + child
12                      data = {
13                          "base": into_branch,
14                          "head": parent_branch,
15                          "commit_message": Merge {parent_branch}
16                          into {into_branch}
17                      }
18
19                      #Realiza a requisicao de merge e trata a resposta
20                      response = requests.post(base_url,
21                                               headers=headers,
22                                               json=data)
23
24                      if response.status_code == 201:
25                          print(Merge entre '{parent_branch}' e
26                                  '{into_branch}' realizado com sucesso.)
27                      elif response.status_code == 409:
28                          print(Conflito de merge entre
29                                  '{parent_branch}' e '{into_branch}'.
30                                  Resolva manualmente.)
31                      else:
32                          print(Falha ao realizar o merge entre
33                                  '{parent_branch}' e '{into_branch}'.
34                                  Codigo de status: {response.status_code}
35                                  Resposta: {response.text})
36
37              if isinstance(children, dict) and children:
38                  merge_branch_tree(base_url, headers, children, depth + 1)
```

Em resumo, a solução tem como principal objetivo ser escalável e adaptável a diferentes contextos de desenvolvimento, permitindo uma fácil integração com os fluxos de trabalho existentes. Além disso, a simplicidade de uso e a clareza das mensagens de saída buscam facilitar a utilização da solução para os desenvolvedores.

4.2 TESTES E VALIDAÇÕES

Para garantir sua eficácia e confiabilidade, a solução foi testada com o objetivo de gerenciar dois projetos caracterizados como variantes. Foram testados cenários que simulassem situações reais de desenvolvimento de melhorias de código já existente, correção de erros e implementação de novas funcionalidades.

Será realizada a comparação de maneira qualitativa através da implementação de simulações de alguns algoritmos para cada estratégia apresentada na seção 2.7, utilizando os seguintes indicadores:

- **Quantidade de Código:** Esta métrica avalia a extensão do código fonte em termos de linhas de código, arquivos ou componentes para que um sistema funcione da maneira desejada. Quantidade alta de código é ruim, pois indica complexidade excessiva e difícil manutenção.
- **Impacto na Performance:** Refere-se à velocidade e eficiência operacional do software durante sua execução. Esta métrica avalia como a organização do código influencia diretamente na rapidez com que o sistema responde às solicitações dos usuários, processa dados e realiza operações críticas.
- **Tempo de Desenvolvimento:** Indica a eficiência e a rapidez com que novos recursos e funcionalidades são desenvolvidos e implementados. Um tempo de desenvolvimento alto é ruim, pois sugere que o processo de criação e integração de novas funcionalidades é demorado e potencialmente ineficaz.
- **Manutenção:** Refere-se à facilidade e ao esforço necessários para manter e atualizar o software ao longo do tempo. Esta métrica considera a capacidade de realizar modificações, correções de erros, implementar melhorias e ajustar o sistema sem comprometer sua estabilidade ou segurança.
- **Curva de Aprendizado:** Indica a facilidade ou dificuldade com que novos membros da equipe podem se familiarizar e começar a contribuir efetivamente para o projeto. Esta métrica é influenciada pela complexidade da estrutura do código, pela documentação disponível e pelas ferramentas utilizadas no processo de desenvolvimento.
- **Complexidade de Escalabilidade:** Refere-se à dificuldade do software de crescer e suportar um aumento na demanda por recursos ou funcionalidades. Uma alta complexidade de escalabilidade é negativa, pois indica que a estrutura atual do sistema torna complicado o crescimento e a adaptação do software às novas demandas sem comprometer sua estabilidade ou desempenho.

4.2.1 SIMULAÇÃO COM MONOREPO

Na busca por esses resultados foi criado um projeto de teste com uma estrutura de branches controlada que permitissem realizar as simulações compatíveis com um ambiente real de produção.

Inicialmente é necessário verificar se a construção da hierarquia das branches está de acordo com o previsto. Essa identificação e estruturação do código são fundamentais para o

processo de automação, pois permite que o algoritmo identifique e percorra a hierarquia das branches de forma precisa, garantindo a integridade do projeto durante as operações de merge. É importante ressaltar que é necessário que as branches respeitem a nomenclatura correta do nome para que o algoritmo retorne a árvore corretamente.

No exemplo utilizado, a branch *main* é o nó raiz da árvore. Abaixo dela, são listadas as branches filhas que foram criadas a partir dela: *main_projeto1* e *main_projeto2*. A branch *main_projeto1* ainda possui uma sub-branch chamada *main_projeto1_var1*, indicando uma hierarquia mais profunda de dependência. A estrutura abaixo reflete a relação entre as diferentes variantes do projeto e fornece uma visão clara sobre a ordem em que as atualizações serão aplicadas pelo programa.

Simulação da saída de hierarquia das branches:

```
main
  main_projeto1
    main_projeto1_var1
  main_projeto2
```

A estrutura do projeto de teste começa pela branch principal *main*, que contém o código inicial do projeto. Todas as outras branches foram criadas a partir dela, mantendo o mesmo código base, conforme ilustrado no Algoritmo 7, que possui um código simplificado representando de forma fictícia diferentes funcionalidades em cada linha do comando "print".

Algoritmo 7 – Algoritmo inicial de todas as branches

```
1 if __name__ == '__main__':
2     print(----- INICIO PROJETO -----)
3
4
5
6     print(----- FINAL PROJETO -----)
```

Foram realizadas modificações na branch *main_projeto1*, como é possível observar no Algoritmo 8, na qual foi adicionada a linha "print(FLUXO DO PROJETO 1)"entre as outras duas linhas de código já existentes. Essa alteração têm como objetivo representar as possíveis particularidades de cada variante.

Algoritmo 8 – Algoritmo da branch main_projeto1 com suas devidas particularidades

```
1 if __name__ == '__main__':
2     print(----- INICIO PROJETO -----)
3
4     print(FLUXO DO PROJETO 1)
5
6     print(----- FINAL PROJETO -----)
```

Após realizado essa alteração, o código da branch *main* também foi atualizado. Conforme é possível observar no Algoritmo 9, foram alterados os caracteres especiais de '-' para '*', simbolizando alguma alteração de código pré-existente, seja para arrumar um erro ou aplicar uma melhoria. Também foi inserida a linha "print(FLUXO DE FUNCIONALIDADE GERAL)", representando alguma nova funcionalidade ou regra de negócio que deve ser replicada a todas as outras variantes.

Algoritmo 9 – Algoritmo com alterações na main para serem replicados em todas as outras

```
1 if __name__ == '__main__':
2     print(***** INICIO PROJETO *****)
3
4
5
6     print(FLUXO DE FUNCIONALIDADE GERAL)
7     print(***** FINAL PROJETO *****)
```

Essas modificações tinham como objetivo atualizar todas as respectivas sub-branches com as alterações realizadas, ao mesmo tempo em que mantinham suas particularidades. As saídas encontradas nas respectivas branches demonstram isso, conforme é possível observar nos Algoritmos 10, 11 e 12

Nota-se que a branch *main_projeto1* (Algoritmo 10) recebeu as atualizações da *main* e manteve a atualização realizada exclusivamente nela. Da mesma forma, *main_projeto1_variavel1* (Algoritmo 11) recebeu as atualizações da *main* e da *main_projeto1*, por ser uma sub-branch de ambas. Já a branch *main_projeto2* (Algoritmo 12) recebeu apenas a atualização da *main*.

Algoritmo 10 – Resultado do merge na branch main_projeto1

```
1 if __name__ == '__main__':
2     print(**** INICIO PROJETO ****)
3
4     print(FLUXO DO PROJETO 1)
5
6     print(FLUXO DE FUNCIONALIDADE GERAL)
7     print(**** FINAL PROJETO ****)
```

Algoritmo 11 – Resultado do merge na branch main_projeto1_variavel1

```
1 if __name__ == '__main__':
2     print(**** INICIO PROJETO ****)
3
4     print(FLUXO DO PROJETO 1)
5
6     print(FLUXO DE FUNCIONALIDADE GERAL)
7     print(**** FINAL PROJETO ****)
```

Algoritmo 12 – Resultado do merge na branch main_projeto2

```
1 if __name__ == '__main__':
2     print(**** INICIO PROJETO ****)
3
4
5
6     print(FLUXO DE FUNCIONALIDADE GERAL)
7     print(**** FINAL PROJETO ****)
```

Além dos testes de integração, também foram simuladas situações de conflitos de merge, nas quais duas ou mais branches apresentavam modificações conflitantes entre si. Esses conflitos geralmente ocorrem quando as mesmas linhas são modificadas de formas diferentes entre as branches.

Para ilustrar isso, observa-se a seguinte situação do Algoritmo 13. Na branch *main*, foi adicionado o código 'print(FLUXO DO PROJETO MAIN)' na mesma linha em que o código 'print(FLUXO DO PROJETO 1)' foi inserido na branch *main_projeto1* (Algoritmo 10).

Algoritmo 13 – Atualização na branch main

```
1 if __name__ == '__main__':
2     print(**** INICIO PROJETO ****)
3
4     print(FLUXO DO PROJETO MAIN)
5
6     print(FLUXO DE FUNCIONALIDADE GERAL)
7     print(**** FINAL PROJETO ****)
```

A execução dessa mesclagem conflitante resultou em um aviso indicando a necessidade de resolver manualmente os conflitos. Segue abaixo o retorno do programa:

Saída da simulação com branches conflitantes:

```
Conflito de merge entre 'main' e 'main_projeto1'. Resolva manualmente.

Merge entre 'main' e 'main_projeto2' realizado com sucesso.
```

Para resolver conflitos é necessário efetuar o merge manualmente. Isso envolve corrigir individualmente as discrepâncias entre as versões de código divergentes apontadas pela ferramenta de versionamento. A ocorrência desses conflitos destacam a importância de uma gestão cuidadosa do código durante o desenvolvimento em um monorepo, para garantir a integridade e a consistência do projeto.

Por fim, para obter o resultado final esperado nos projetos, foram realizadas alterações nas branches *main_projeto1_variavel1* e *main_projeto2*, adicionando respectivamente as linhas de código "print(FLUXO DO PROJETO 1 - VARIANTE)"e "print(FLUXO DO PROJETO 2)",

conforme apresentado nos Algoritmos 14 e 15. Essas modificações não irão alterar nenhuma outra variante, pois trata-se de branches situadas nas extremidades da árvore dos produtos.

Algoritmo 14 – Resultado final da branch main_projeto1_variavel1

```
1 if __name__ == '__main__':
2     print(**** INICIO PROJETO ****)
3
4     print(FLUXO DO PROJETO 1 - VARIANTE)
5
6     print(FLUXO DE FUNCIONALIDADE GERAL)
7     print(**** FINAL PROJETO ****)
```

Algoritmo 15 – Resultado final da branch main_projeto2

```
1 if __name__ == '__main__':
2     print(**** INICIO PROJETO ****)
3
4     print(FLUXO DO PROJETO 2)
5
6     print(FLUXO DE FUNCIONALIDADE GERAL)
7     print(**** FINAL PROJETO ****)
```

4.2.1.1 ANÁLISE E RESULTADOS COM MONOREPO

A Tabela 4 apresenta uma análise qualitativa dos principais indicadores associados a solução de monorepo com branching. Estes resultados foram adquiridos através da observação do comportamento dos cenários simulados na seção anterior.

Tabela 4 – Análise da Solução com Monorepo

Critério	**Análise**
Quantidade de Código	Não afetou
Impacto na Performance	Não afetou
Tempo de Desenvolvimento	Baixo
Manutenção	Moderada
Curva de Aprendizado	Baixa
Complexidade de Escalabilidade	Baixa

- **Quantidade de Código:** A quantidade de código não sofre alteração, já que todas as variantes compartilham o mesmo código-fonte. As novas funcionalidades ou correções são feitas uma única vez na main e replicadas via automatização para todas as outras variantes.
- **Impacto na Performance:** Não há impacto na performance. Uma linha adicional em *main_projeto1* não afeta a eficiência de *main_projeto2*, pois o fluxo de execução de cada uma são independentes e não sofre alterações para suportar os projetos.

- **Tempo de Desenvolvimento:** O tempo de desenvolvimento é baixo, pois as alterações são feitas em um único ambiente integrado. As mudanças no código principal (*main*) são automaticamente propagadas para as variantes (*main_projeto1* e *main_projeto2*), economizando tempo ao não precisar replicar manualmente as modificações.

- **Manutenção:** A manutenção é moderada, exigindo um gerenciamento cuidadoso das branches e das operações de atualização entre elas para garantir a consistência do código e evitar conflitos de merge. Por exemplo, ao corrigir um bug na main, a correção é replicada para todas as variantes, assegurando a consistência do código em todo o monorepo. No entanto, conforme observado nos testes realizados, conflitos de merge podem ocorrer quando diferentes variantes modificam as mesmas linhas de código de forma divergente, exigindo resolução manual e dificultando o processo de atualização.

- **Curva de Aprendizado:** A curva de aprendizado é geralmente baixa, pois presume-se que os desenvolvedores já estejam familiarizados com o uso de ferramentas de controle de versão. A estrutura de branches, como main, *main_projeto1*, *main_projeto1_var1* e *main_projeto2*, é similar ao de árvores, permitindo que novos desenvolvedores rapidamente entendam e trabalhem com o código.

- **Complexidade de Escalabilidade:** A dificuldade de escalar novas variantes é baixa, uma vez que para criar um projeto 3, basta clonar a branch main via comando da ferramenta de versionamento e prosseguir com o desenvolvimento de suas funcionalidades específicas nessa ramificação independente.

A solução de monorepo com branching permite manter um código base centralizado, facilitando a aplicação de mudanças e a consistência entre as variantes. No entanto, requer um gerenciamento cuidadoso das branches para evitar a ocorrência de conflitos durante as atualizações. Isso oferece uma solução equilibrada entre eficiência no desenvolvimento e a gestão de variantes específicas.

4.2.2 SIMULAÇÃO COM MULTIREPO

No cenário de multirepos, cada projeto é mantido em seu próprio repositório. Os exemplos 16, 17 e 18 apresentados representam a ideia de que são mantidos separadamente em seus repositórios ao mesmo tempo em que contém suas próprias particularidades no mesmo código base.

Algoritmo 16 – Repositório do projeto 1

```
1
2 if __name__ == '__main__':
3     print(**** INICIO PROJETO ****)
4
```

```
5       print(FLUXO DO PROJETO 1 )
6
7       print(FLUXO DE FUNCIONALIDADE GERAL)
8       print(**** FINAL PROJETO ****)
```

Algoritmo 17 – Repositório da variante do projeto 1

```
1
2  if __name__ == '__main__':
3       print(**** INICIO PROJETO ****)
4
5       print(FLUXO DO PROJETO 1 - VARIANTE)
6
7       print(FLUXO DE FUNCIONALIDADE GERAL)
8       print(**** FINAL PROJETO ****)
```

Algoritmo 18 – Repositório do projeto 2

```
1
2  if __name__ == '__main__':
3       print(**** INICIO PROJETO ****)
4
5       print(FLUXO DO PROJETO 2)
6
7       print(FLUXO DE FUNCIONALIDADE GERAL)
8       print(**** FINAL PROJETO ****)
```

4.2.2.1 ANÁLISE E RESULTADOS COM MULTIREPOS

A Tabela 5 foi construída com base no estudo da implementação apresentada de multirepos.

Tabela 5 – Análise da Abordagem com Multirepos

Critério	**Análise**
Quantidade de Código	Alta
Impacto na Performance	Não afetou
Tempo de Desenvolvimento	Alto
Manutenção	Alta
Curva de Aprendizado	Baixa
Complexidade de Escalabilidade	Alta

- **Quantidade de Código:** A quantidade de código é alta devido à necessidade de duplicação do código para cada um dos repositórios. Por exemplo, tanto o projeto 1 quanto o projeto 2 possuem arquivos *main.py* que incluem o fluxo "FLUXO DE FUNCIONALIDADE GERAL", resultando em código duplicado em cada repositório.

- **Impacto na Performance:** Não há impacto na performance, pois cada repositório opera de forma independente e o fluxo de execução do sistema não sofre alterações para suportar os outros projetos. Por exemplo, o projeto 2 executa seu próprio fluxo sem depender dos fluxos dos projetos 1 ou sua variante, garantindo que a performance de cada projeto permaneça inalterada.

- **Tempo de Desenvolvimento:** O tempo de desenvolvimento é maior devido à necessidade de gerenciar múltiplos repositórios e sincronizar mudanças comuns entre eles, precisando duplicar códigos em ambientes diferentes. Por exemplo, ao adicionar uma nova funcionalidade geral, essa mudança precisa ser aplicada separadamente em cada repositório, aumentando o tempo necessário para desenvolvimento.

- **Manutenção:** A manutenção é alta, pois é necessário aplicar correções e atualizações em todos os repositórios que compartilham código comum. Por exemplo, se uma alteração for feita na "FUNCIONALIDADE GERAL", essa mudança precisa ser replicada manualmente em todos os repositórios do projeto 1 e sua variante, e projeto 2, o que aumenta o esforço e o risco de inconsistências.

- **Curva de Aprendizado:** A curva de aprendizado é baixa, pois cada repositório é independente, simplificando o entendimento inicial. Novos desenvolvedores podem facilmente se familiarizar com um único repositório e seu fluxo específico sem precisar entender a estrutura completa dos outros projetos.

- **Complexidade de Escalabilidade:** Exige um alto esforço para escalar novas variantes. Por exemplo, para adicionar um projeto 3 seria necessário criar um novo repositório, duplicar o código base necessário e ajustar as configurações específicas, aumentando a complexidade e o esforço.

A abordagem de multirepos oferece uma separação clara do código, reduzindo riscos de conflitos entre diferentes fluxos de projetos. No entanto, a replicação de mudanças em múltiplos repositórios resulta em alta manutenção e duplicação de código comum, complicando a sincronização de atualizações entre as variantes.

4.2.3 SIMULAÇÃO COM FEATURE FLAGS

No Algoritmo 19, utilizamos Feature Flags para gerenciar as funcionalidades dos diferentes fluxos de variantes. Através das variáveis *FLAG_PROJETO_1*, *FLAG_PROJETO_2* e *FLAG_PROJETO_1_VARIANTE* definidas nos arquivos de cada projeto, é possível controlar dinamicamente a inclusão dos fluxos específicas de código em um único projeto. Isso permite ativar ou desativar funcionalidades de forma flexível sem a necessidade de alterar o código principal, que somente precisa identificar qual o arquivo de flags habilitado no momento da execução.

Algoritmo 19 – Resultado do projeto com Feature Flag

```
1
2  # Feature Flags do Projeto 1
3  FLAG_PROJETO_1 = True
4  FLAG_PROJETO_1_VARIANTE = False
5  FLAG_PROJETO_2 = False
6
7  # Feature Flags do Projeto 1 - Variante
8  FLAG_PROJETO_1 = True
9  FLAG_PROJETO_1_VARIANTE = True
10 FLAG_PROJETO_2 = False
11
12 # Feature Flags do Projeto 2
13 FLAG_PROJETO_1 = False
14 FLAG_PROJETO_1_VARIANTE = False
15 FLAG_PROJETO_2 = True
16
17 # arquivo main.py
18 if __name__ == '__main__':
19     print(***** INICIO PROJETO *****)
20
21     if FLAG_PROJETO_1:
22         if FLAG_PROJETO_1_VARIANTE:
23             print(FLUXO DO PROJETO 1 - VARIANTE)
24         else:
25             print(FLUXO DO PROJETO 1)
26
27     if FLAG_PROJETO_2:
28         print(FLUXO DO PROJETO 2)
29
30     print(FLUXO DE FUNCIONALIDADE GERAL)
31     print(***** FINAL PROJETO *****)
```

4.2.3.1 ANÁLISE DE RESULTADOS COM FEATURE FLAGS

A Tabela 6 apresenta o estudo das características e resultados com a utilização de flags no código-fonte.

- **Quantidade de Código:** A quantidade de código é moderada, pois as Feature Flags exigem a inserção de condicionais no código principal para determinar o fluxo que deverá seguir. Por exemplo, o código principal *main.py* inclui verificações condicionais para as flags *FLAG_PROJETO_1*, *FLAG_PROJETO_1_VARIANTE* e *FLAG_PROJETO_2*. Além disso, é necessário adicionar as novas flags na base de dados, o que aumenta a quantidade de código necessário para gerenciá-las.

Tabela 6 – Análise da Abordagem com Feature Flags

Critério	**Análise**
Quantidade de Código	Moderada
Impacto na Performance	Baixo
Tempo de Desenvolvimento	Baixo
Manutenção	Moderada
Curva de Aprendizado	Baixa
Complexidade de Escalabilidade	Baixo

- **Impacto na Performance:** O impacto na performance é baixo, pois as condicionais são validadas pelo programa em tempo de execução e não afetam significativamente o desempenho do código. Por exemplo, a verificação de *FLAG_PROJETO_1* ou *FLAG_PROJETO_2* é uma operação rápida que exige um pré-carregamento mínimo das flags, resultando em um impacto leve de performance na inicialização do programa.

- **Tempo de Desenvolvimento:** O tempo de desenvolvimento é baixo, pois basta adicionar a flag necessária na base de dados já construída, sem necessidade de grandes alterações no código principal. Por exemplo, para adicionar uma nova variante, apenas uma nova flag precisa ser definida e uma condição correspondente adicionada ao código principal.

- **Manutenção:** A manutenção é moderada devido à necessidade de gerenciar múltiplas condições no código base para ativar ou desativar funcionalidades específicas. Por exemplo, a definição das flags *FLAG_PROJETO_1* e *FLAG_PROJETO_1_VARIANTE* e a implementação das condições associadas requerem cuidado para evitar a complexidade excessiva e garantir que as funcionalidades corretas sejam ativadas ou desativadas conforme necessário.

- **Curva de Aprendizado:** A curva de aprendizado pode ser considerada baixa, pois após a estrutura das Feature Flags estar construída, os desenvolvedores precisam apenas continuar adicionando novas flags conforme necessário.

- **Complexidade de Escalabilidade:** A complexidade de escalabilidade é baixa devido à flexibilidade dessa abordagem, permitindo adicionar e testar novas funcionalidades sem alterar o código principal. Por exemplo, novas funcionalidades ou variantes podem ser ativadas simplesmente definindo novas flags e inserindo a condicional no código principal.

A abordagem com Feature Flags proporciona flexibilidade e eficiência na gestão de funcionalidades específicas, permitindo a ativação ou desativação de partes do código sem a necessidade de modificações extensivas. Isso facilita a implementação de novas funcionalidades e a correção de erros, reduzindo o tempo de desenvolvimento e melhorando a manutenção do código. No entanto, requer uma gestão cuidadosa das flags e das condições associadas para evitar complexidade excessiva e garantir a integridade do código.

4.2.4 SIMULAÇÃO COM BIBLIOTECAS

A abordagem com bibliotecas compartilhadas centraliza funcionalidades comuns em um único módulo para que possam ser modificadas em somente um único lugar, como por exemplo, a classe *biblioteca.py* do Algoritmo 20, que pode ser reutilizado pelos diferentes projetos. No exemplo, as funções são definidas na biblioteca e importadas no *main.py* dos Algoritmos 21, 22 e 23 para que possam ser utilizadas conforme necessário de acordo com cada projeto. Além disso, é possível observar que a função "projeto1"aceita um parâmetro para incluir particularidades de projetos específicos, permitindo flexibilidade no fluxo das funcionalidades.

Algoritmo 20 – Bibliotecas compartilhadas

```
1
2  # biblioteca.py
3  def funcionalidade_inicial():
4       print(***** INICIO PROJETO *****)
5
6  def projeto1(projeto=None):
7      if projeto == "VARIANTE":
8          print(FLUXO DO PROJETO 1 - VARIANTE)
9      else:
10         print(FLUXO DO PROJETO 1)
11
12 def projeto2():
13     print(FLUXO DO PROJETO 2)
14
15 def funcionalidade_geral():
16      print(FLUXO DE FUNCIONALIDADE GERAL)
17
18 def funcionalidade_final():
19     print(***** FINAL PROJETO *****)
```

Algoritmo 21 – Exemplo de uso das bibliotecas no projeto 1

```
1
2  # main.py
3  import biblioteca
4
5  if __name__ == '__main__':
6      biblioteca.funcionalidade_inicial()
7      biblioteca.projeto1()
8      biblioteca.funcionalidade_geral()
9      biblioteca.funcionalidade_final()
```

Algoritmo 22 – Exemplo de uso das bibliotecas da variante do projeto 1

```
1
2  # main.py
```

```
3 import biblioteca
4
5 if __name__ == '__main__':
6     biblioteca.funcionalidade_inicial()
7     biblioteca.projeto1(projeto="VARIANTE")
8     biblioteca.funcionalidade_geral()
9     biblioteca.funcionalidade_final()
```

Algoritmo 23 – Exemplo de uso das bibliotecas no projeto 2

```
1
2 # main.py
3 import biblioteca
4
5 if __name__ == '__main__':
6     biblioteca.funcionalidade_inicial()
7     biblioteca.projeto2()
8     biblioteca.funcionalidade_geral()
9     biblioteca.funcionalidade_final()
```

4.2.4.1 ANÁLISE DE RESULTADOS DAS BIBLIOTECAS COMPARTILHADAS

A Tabela 7 contém os resultados da abordagem de bibliotecas compartilhadas no cenário desenvolvido.

Tabela 7 – Análise da Abordagem com Bibliotecas Compartilhadas

Critério	**Análise**
Quantidade de Código	Baixa
Impacto na Performance	Não afetou
Tempo de Desenvolvimento	Moderada
Manutenção	Moderada
Curva de Aprendizado	Alta
Complexidade de Escalabilidade	Moderada

- **Quantidade de Código:** A quantidade de código é baixa, pois as funcionalidades são centralizadas em uma biblioteca compartilhada, como no exemplo da classe *biblioteca.py*, e reutilizadas pelos diferentes projetos (*main.py*), evitando assim, duplicação de código.
- **Impacto na Performance:** Não há impacto na performance, uma vez que as bibliotecas são importadas e utilizadas como qualquer outro código. Na simulação desenvolvida, as funções *funcionalidade_inicial()* ou *funcionalidade_geral()* não afetam a performance do software.

- **Tempo de Desenvolvimento:** A complexidade é moderada, pois requer a definição clara de interfaces e dependências na classe *biblioteca.py*. No entanto, a reutilização de código acelera o desenvolvimento de novos projetos. Por exemplo, a função *projeto1()* que está sendo utilizada em outro projeto sem a necessidade de reescrever a mesma funcionalidade, economizando tempo e esforço.

- **Manutenção:** A manutenção é moderada. As atualizações e correções precisam ser feitas apenas na biblioteca compartilhada, refletindo automaticamente em todos os projetos que a utilizam. No entanto, é necessário garantir que essas alterações não comprometam outras funcionalidades que dependem delas. Por exemplo, a função *biblioteca.projeto1(projeto="VARIANTE")* utiliza o mesmo fluxo do projeto 1 e pode falhar se ocorrerem alterações inconsistentes nesse fluxo.

- **Curva de Aprendizado:** A curva de aprendizado pode ser considerada alta, pois envolve entender como criar e utilizar funcionalidades comuns em uma biblioteca centralizada. Novos desenvolvedores precisam se familiarizar com a biblioteca compartilhada e suas funções.

- **Complexidade de Escalabilidade:** A escalabilidade é moderada, pois as bibliotecas compartilhadas facilitam a reutilização de código. No entanto, adicionar parametrizações diferentes dentro de uma mesma função, como em *projeto1(projeto="VARIANTE")*, pode exigir um maior esforço para manter a clareza e a coesão do código, garantindo que a biblioteca continue a suportar novos requisitos e funcionalidades sem comprometer sua integridade.

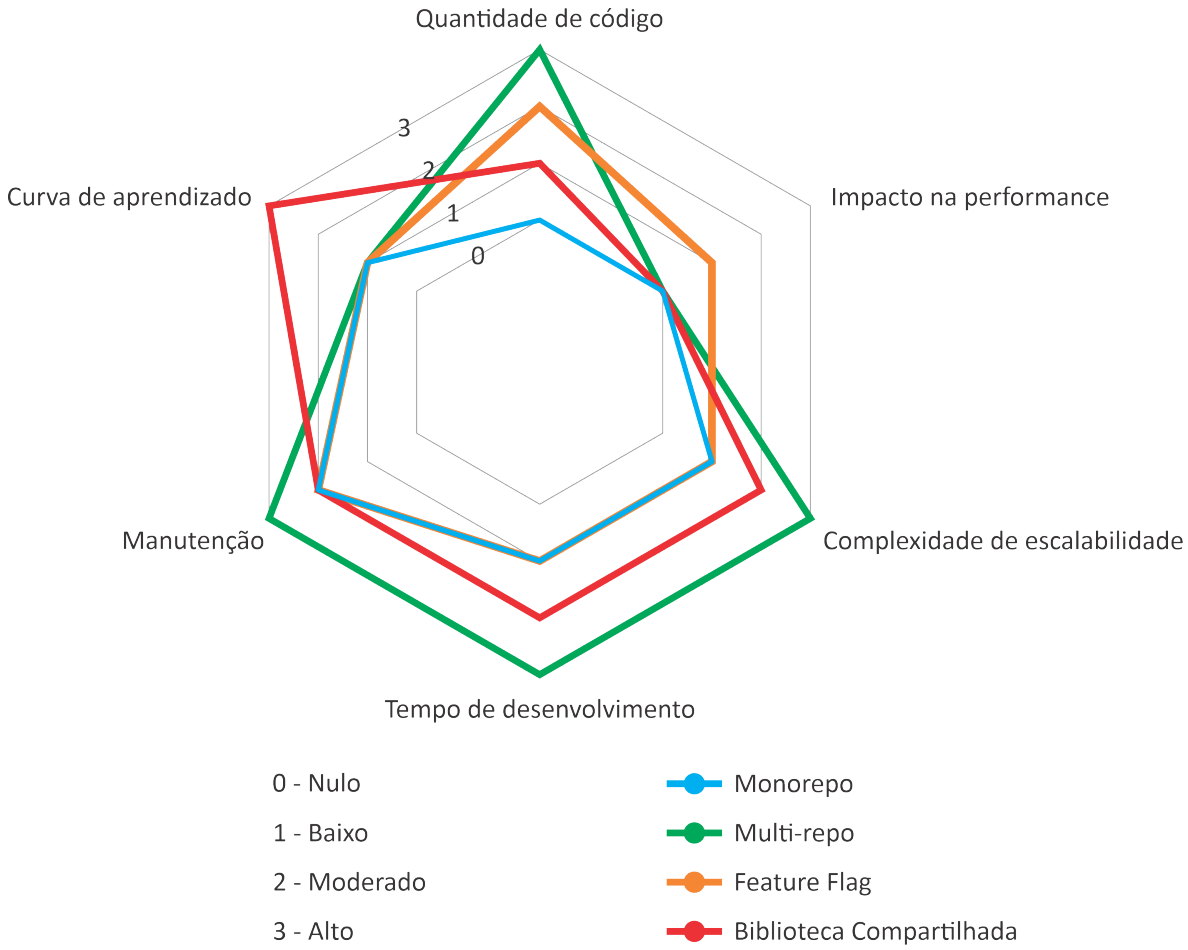
A abordagem de bibliotecas compartilhadas oferece vantagens significativas em termos de reutilização de código e manutenção centralizada. No entanto, requer atenção especial para garantir que mudanças na biblioteca não causem regressões em projetos que dependem dela. A documentação clara e a automação de testes são essenciais para mitigar esses riscos.

4.2.5 ANÁLISE GERAL DOS RESULTADOS

Para uma melhor visualização e comparação das diferentes abordagens analisadas na seção anterior, foi construído o gráfico da Figura 17, que apresenta as avaliações de cada critério considerado para as abordagens de bibliotecas compartilhadas, feature flags, multirepos e monorepo. Esse gráfico permite identificar rapidamente as vantagens e desvantagens de cada abordagem no contexto de gestão de variantes de produto, facilitando a análise integrada dos resultados e ajudando na escolha da solução mais adequada para projetos com um núcleo de código similar.

É possível observar no gráfico que a abordagem de bibliotecas compartilhadas é eficiente em termos de quantidade de código, pois permite a reutilização de componentes. No

Figura 17 – Gráfico de Rede da Análise de Resultados



Fonte: O Autor

entanto, apresenta desafios consideráveis, como uma curva de aprendizado alta e uma manutenção de nível moderado. Embora não tenha impacto negativo na performance, o tempo de desenvolvimento é moderado, o que pode ser um obstáculo em projetos que demandam agilidade.

Por outro lado, a utilização de Feature Flags tem uma quantidade de código moderada e impacta pouco na performance do sistema. O tempo de desenvolvimento e a curva de aprendizado são baixos, tornando essa abordagem atraente para implementações rápidas e de fácil compreensão. No entanto, a manutenção é moderada, sugerindo que, a longo prazo, pode haver desafios na gestão das diversas flags implementadas para diferentes variantes de produto.

A abordagem com multirepos, embora ofereça uma clara separação de responsabilidades entre diferentes repositórios, apresenta desafios significativos. A quantidade de código e a complexidade de escalabilidade são altas, o que pode complicar a manutenção e o tempo de desenvolvimento. A curva de aprendizado é baixa, mas os altos níveis de manutenção e desenvolvimento necessários podem ser desvantagens consideráveis para gerenciar projetos com múltiplas variantes de produto.

Por fim, a abordagem de monorepo sobressai em vários critérios importantes. Não há necessidade de aumentar a quantidade de código principal para manter as variantes, o que simplifica consideravelmente o processo de desenvolvimento. O impacto na performance não foi afetado, e o tempo de desenvolvimento é baixo, permitindo uma rápida implementação de novas funcionalidades e correções. A manutenção, embora moderada, é facilitada pela centralização do código e automação das atualizações entre as branches. Além disso, a curva de aprendizado baixa e a facilidade na escalabilidade tornam esta abordagem particularmente atraente para as equipes de desenvolvimento.

Em uma análise global, a abordagem de monorepo se destaca como a mais eficiente e prática entre as abordagens analisadas. Sua capacidade de centralizar o código sem comprometer a performance, juntamente com a facilidade de desenvolvimento e manutenção, faz dela a escolha ideal para projetos variantes que demandam agilidade e simplicidade. A possibilidade da separação clara do código em branches, ao mesmo tempo em que consegue manter facilmente todas essas variantes atualizadas sem reescrita ou duplicação de código é o fator decisivo que contribui significativamente para sua superioridade.

5 CONSIDERAÇÕES FINAIS

A gestão eficiente da diversificação de produtos, especialmente no contexto das variantes de software, é um desafio crucial para as empresas. Adaptar e diversificar produtos são elementos essenciais para garantir competitividade e sustentabilidade em um mercado dinâmico e em constante evolução.

Empresas como Google, Facebook e Microsoft demonstraram na prática como os repositórios centralizados podem ser uma solução eficiente para lidar com a complexidade associada à diversificação de software. Ao permitir que equipes trabalhem simultaneamente em diferentes projetos dentro do mesmo ambiente, essa solução acelerou o ciclo de desenvolvimento e ampliou a entrega dessas empresas líderes de mercado.

A adoção de monorepos não é apenas uma decisão técnica nessas empresas, mas também reflete profundamente sua cultura organizacional. Essa transição frequentemente requer ajustes significativos nos processos de desenvolvimento e ferramentas utilizadas, exigindo uma avaliação contínua de sua eficácia e adaptações conforme necessário.

Nesse contexto, foi estudada e desenvolvida uma solução que utiliza um repositório centralizado com branches para cada produto, desde que esses produtos sejam variantes de um mesmo núcleo de código. Essa técnica visa simplificar o desenvolvimento e a manutenção, facilitando a construção ágil e eficiente de diferentes versões e personalizações em um ambiente unificado.

Durante a pesquisa, identificaram-se desafios de gestão entre as branches para manter todas as variantes atualizadas e evitar conflitos. Para minimizar esses problemas, foi desenvolvido um sistema de automação para o processo de integração entre as branches, tornando o gerenciamento mais seguro e confiável.

A combinação de repositórios centralizados com o uso de branching demonstrou resultados positivos nos estudos de caso conduzidos na empresa Eterno Software, proporcionando os meios necessários para enfrentar os desafios da diversificação de seus produtos. Considerações de custo também foram cruciais, já que a implementação de um único repositório centralizado com branches para cada variante ofereceu significativa economia de recursos em comparação com outras abordagens estudadas. Dessa forma, a solução não apenas atende às exigências técnicas da Eterno Software, mas também oferece uma abordagem mais econômica e eficiente para sustentar o crescimento e a competitividade da empresa no mercado.

Conclui-se, com base na pesquisa realizada neste trabalho, que o uso de monorepos com branching é satisfatório, tornando-a uma excelente solução para empresas que buscam uma maneira acessível e de alto desempenho para gerenciar a complexidade de múltiplos produtos similares.

Como trabalhos futuros, sugere-se o desenvolvimento de pipelines DevOps integrados para otimizar ainda mais o processo de desenvolvimento e deploy contínuo, aumentando a eficiência e a qualidade das entregas.

REFERÊNCIAS

AGGARWAL, A.; SINGH, V.; KUMAR, N. A rapid transition from subversion to git: time, space, branching, merging, offline commits & offline builds and repository aspects. **Recent Advances in Computer Science and Communications (Formerly: Recent Patents on Computer Science)**, Bentham Science Publishers, v. 15, n. 5, p. 653–660, 2022.

ALFADEL, M.; MCINTOSH, S. The classics never go out of style: An empirical study of downgrades from the bazel build technology. In: **Proceedings of the IEEE/ACM 46th International Conference on Software Engineering**. [S.l.: s.n.], 2024. p. 1–12.

ATLASSIANN. **Feature Flag**. 2024. Disponível em: <https://www.atlassian.com/solutions/devops/integrations/feature-flags>.

BATORY, D. *et al.* Scalable software libraries. **ACM SIGSOFT Software Engineering Notes**, ACM New York, NY, USA, v. 18, n. 5, p. 191–199, 1993.

BLINOWSKI, G.; OJDOWSKA, A.; PRZYBYŁEK, A. Monolithic vs. microservice architecture: A performance and scalability evaluation. **IEEE Access**, v. 10, p. 20357–20374, 2022.

BRITO, G.; TERRA, R.; VALENTE, M. T. Monorepos: a multivocal literature review. **arXiv preprint arXiv:1810.09477**, v. 18, 2018.

BROUSSE, N. The issue of monorepo and polyrepo in large enterprises. In: **Companion proceedings of the 3rd international conference on the art, science, and engineering of programming**. [S.l.: s.n.], 2019. p. 1–4.

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

DECAN, A. *et al.* On the use of github actions in software development repositories. In: **2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2022. p. 235–245.

DIALECTICA. Shared libraries in a microservice architecture. 2023. Acessado em 9 de julho de 2024. Disponível em: <https://www.dialecticanet.com/shared-libraries-in-a-microservice-architecture/>.

DOAN, N. Development of tooling for software development teams. 2023.

DRIESSEN, V. **Main/Master Branch**. 2010. Disponível em: <https://nvie.com/posts/a-successful-git-branching-model/>.

EBERT, C. *et al.* Devops. **IEEE software**, Ieee, v. 33, n. 3, p. 94–100, 2016.

ELIJAH, N. **A Beginner's Guide to Monorepos**. 2023. <https://blog.openreplay.com/a-beginners-guide-to-monorepos/>. Acesso em: 6 jun. 2024.

FRÖMMGEN, A. *et al.* Resolving code review comments with machine learning. In: **Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice**. [S.l.: s.n.], 2024. p. 204–215.

GEHMAN, C. What is a monorepo. **What Is a Monorepo**, 2021.

HUMBLE, J.; FARLEY, D. **Continuous delivery: reliable software releases through build, test, and deployment automation**. [S.l.]: Pearson Education, 2010.

KHLEEL, N. A. A.; KÁROLY, N. Comparison of version control system tools. **Multidiszciplináris Tudományok**, v. 10, n. 3, p. 61–69, 2020.

KINSMAN, T. *et al.* How do software developers use github actions to automate their workflows? In: **2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)**. [S.l.: s.n.], 2021. p. 420–431.

LOG, M. Monorepo vs. multirepo: Pros and cons for managing codebases. **Moments Log**, 2022. Disponível em: <https://www.momentslog.com/articles/monorepo-vs-multirepo-pros-and-cons>.

MATUTE, G.; CHEUNG, A.; CHASINS, S. Changes in software ecosystems. **PLATEAU**, 2022.

MISHRA, A.; OTAIWI, Z. Devops and software quality: A systematic mapping. **Computer Science Review**, Elsevier, v. 38, p. 100308, 2020.

MORGAN, J. **Reusable Code: The Good, The Bad, and The Ugly**. 2019. Accessado em 2024-05-22. Disponível em: <https://dev.to/pluralsight/reusable-code-the-good-the-bad-and-the-ugly-3do6>.

O'SULLIVAN, B. **Mercurial: The Definitive Guide: The Definitive Guide**. [S.l.]: "O'Reilly Media, Inc.", 2009.

PESSôA, C. **Monolítico x Monorepo x Multi-Repo**. 2022. Disponível em: <https://www.alura.com.br/artigos/monorepo-usa-lo-desenvolver-integrar-grandes-projetos>.

PONUTHORAI, P. K.; LOELIGER, J. **Version Control with Git**. [S.l.]: "O'Reilly Media, Inc.", 2022.

PORTER, M. E. **Competitive Advantage: Creating and Sustaining Superior Performance**. [S.l.]: Free Press, 1985.

POTVIN, R.; LEVENBERG, J. Why google stores billions of lines of code in a single repository. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 59, n. 7, p. 78–87, jun 2016. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/2854146>.

RAFTT. **Monolítico x Monorepo x Multi-Repo**. 2022. Disponível em: <https://www.raftt.io/post/development-challenges-of-working-with-monorepos-and-multirepos.html>.

ROSU, C.; TOGAN, M. A modern paradigm for effective software development: Feature toggle systems. In: **2023 15th International Conference on Advanced Software Engineering & Its Applications**. [s.n.], 2023. Disponível em: <https://ieeexplore.ieee.org/abstract/document/10193936/>.

SANTANA, B. N. Diversificação da produção e diferenciação de produto: um estudo de caso de uma empresa do setor de logística para o setor financeiro. Universidade Estadual Paulista (Unesp), 2023.

SOARES, E. *et al.* The effects of continuous integration on software development: a systematic literature review. **Empirical Software Engineering**, Springer, v. 27, n. 3, p. 78, 2022.

SOHL, T.; VROOM, G.; MCCANN, B. T. Business model diversification and firm performance: A demand-side perspective. **Strategic entrepreneurship journal**, Wiley Online Library, v. 14, n. 2, p. 198–223, 2020.

THOUGHTWORKS. Monorepo vs. multi-repo: Different strategies for organizing repositories. **ThoughtWorks Blog**, 2020. Disponível em: <https://www.thoughtworks.com/insights/articles/monorepo-vs-multi-repo>.