INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE ENGENHARIA ELETRÔNICA

HELLEN ÁVILA ROSA

ESTUDO DE CASO: PROCESSOS DE DESENVOLVIMENTO DE *SOFTWARE* EMBARCADO UTILIZANDO FERRAMENTAS DE AUTOMATIZAÇÃO

Florianópolis, 2021

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE ENGENHARIA ELETRÔNICA

HELLEN ÁVILA ROSA

ESTUDO DE CASO: PROCESSOS DE DESENVOLVIMENTO DE *SOFTWARE* EMBARCADO UTILIZANDO FERRAMENTAS DE AUTOMATIZAÇÃO

Trabalho de conclusão de curso submetido ao Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina como parte dos requisitos para obtenção do título de Engenheira Eletrônica

Orientador:
Prof. Dr. Renan Augusto Starke

Florianópolis, 2021

Ficha de identificação da obra elaborada pelo autor.

Rosa, Hellen
Estudo de caso: Processos de desenvolvimento de software embarcado utilizando ferramentas de automatização / Hellen Rosa ; orientação de Renan Augusto Starke.
- Florianópolis, SC, 2021.
50 p.

Trabalho de Conclusão de Curso (TCC) - Instituto Federal de Santa Catarina, Câmpus Florianópolis. Bacharelado em Engenharia Eletrônica. Departamento Acadêmico de Eletrônica.
Inclui Referências.

1. Automatização de processos. 2. Software embarcado . 3. Métodos Ágeis. I. Augusto Starke, Renan . II. Instituto Federal de Santa Catarina. Departamento Acadêmico de Eletrônica. III. Título.

ESTUDO DE CASO: PROCESSOS DE DESENVOLVIMENTO DE *SOFTWARE* EMBARCADO UTILIZANDO FERRAMENTAS DE AUTOMATIZAÇÃO

HELLEN ÁVILA ROSA

Este Trabalho foi julgado adequado para obtenção do Título de Engenheira Eletrônica em abril de 2021 e aprovado na sua forma final pela banca examinadora do Curso de Engenharia Eletrônica do instituto Federal de Educação Ciência, e Tecnologia de Santa Catarina.

Florianópolis, 19 de abril, 2021.

Banca Examinadora:

Renan Augusto Starke, Dr. Eng.

Matheus Leitzke Pinto, Msc.

Gustavo Martins de Araújo Silvano, Eng.

AGRADECIMENTOS

Em primeiro lugar a Deus, que foi a minha força durante todos esses anos.

Aos meus pais Giselle e Fabiano, meu irmão Hector e meu avós Rosalina e João, que investiram e confiaram no meu potencial a vida inteira. Eles que sempre foram minha base e meu socorro, que me incentivaram nos momentos difíceis e compreenderam a minha ausência enquanto eu me dedicava a realizar este trabalho.

Ao meu marido Jonathan, que acima de tudo é meu melhor amigo, que sempre esteve ao meu lado me ajudando a enfrentar as dificuldades e nunca me deixou desistir dos meus sonhos. Obrigada por todo apoio!

Meu orientador Renan, que além de ter sido um orientador paciente e prestativo, também foi um professor que marcou minha jornada acadêmica.

Aos meus amigos que essa jornada me trouxe, Letícia, Ana Claúdia, Diesson, Mariele, Felipe e Vítor. Obrigada por todo companheirismo, risadas e apoio durante todos esses anos.

A GE (General Electric) por ter me proporcionado um ambiente de estágio de muito aprendizado e desafios. Um agradecimento especial ao meu *manager* Gustavo, que sempre me ensinou com muita paciência e esteve ao meu lado dando suporte na realização deste trabalho.

Por fim, agradeço ao IFSC por proporcionar um ensino de qualidade. E em especial a todos os professores do departamento de eletrônica do IFSC, que compartilharam seus conhecimento e ensinaram com maestria, sem vocês isso não seria possível.

Obrigada a todos!

“Nossa maior fraqueza é a desistência.
O caminho mais certeiro para o sucesso
é sempre tentar apenas uma vez mais.”
(Thomas Edison)

RESUMO

Atualmente os métodos tradicionais de desenvolvimento de *software* já não atendem às demandas geradas pelos clientes, que cada vez mais exigem atualizações, novas funcionalidades e novos produtos de forma cada vez mais rápida. Por esse motivo, as empresas estão buscando alternativas para acelerar o *time-to-market*, aumentar qualidade do código, aprimorar o processo de desenvolvimento e melhorar comunicação entre os times. Assim, este trabalho aborda um estudo de caso sobre os impactos da automatização de processos de desenvolvimento de um *software* embarcado com o intuito de analisar seus benefícios. Utilizaram-se algumas ferramentas como: *Jenkins*, *GitHub*, *software* Jira e *Docker* para realizar este estudo de caso, que possui uma metodologia do tipo exploratória com abordagem qualitativa. Por fim, com a automatização de processos em conjunto com métodos ágeis e a cultura *Devops*, obtiveram-se os seguintes resultados: melhora na qualidade do código, testes automatizados, ambiente de desenvolvimento padronizado, *releases* mais frequentes, automatização da publicação e notificação de novas versões.

Palavras-chave: Automatização de processos. *Software* embarcado. Métodos Ágeis.

ABSTRACT

Nowadays traditional software development methods no longer meet the customer demands with increasingly updates, new features and new products. For this reason, companies are looking for alternatives to speed up the time to market, to increase the quality of the code, to improve development process and to improve communication of teams. Therefore, this work addresses a case study on the impacts development process automation in embedded software, in order to analyze its benefits. We used some tools such: Jenkins, GitHub, Jira software and Docker. This work has an exploratory-type methodology with a qualitative approach. Finally, the automation of processes with agile methods and the DevOps culture obtained the following results: code quality improvement, automated tests creation, standardized development environment, frequent releases development, increasing of automated publications and new versions notification.

Keywords: Processes automation. Embedded software. Agile Methods.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CI/CD	*Continuous Integration and Continuous Delivery* - Integração contínua e Entrega contínua

SUMÁRIO

1 INTRODUÇÃO

Os métodos tradicionais de desenvolvimento não atendem às demandas geradas pelos clientes atualmente, que cada vez mais exigem atualizações, novas funcionalidades e produtos novos de forma cada vez mais rápida. As empresas estão buscando cada vez mais alternativas para acelerar o *time-to-market*, diminuir custos dos seus produtos e entregar um *software* com qualidade. Frequentemente um processo de desenvolvimento de *software* possui muitas tarefas repetitivas e manuais que vão além do desenvolvimento de código.

Como solução para diminuir o *time-to-market*, melhorar a qualidade e acelerar o processo de desenvolvimento, as empresas estão investindo na cultura *DevOps* (LWAKATARE et al., 2016), aplicando os métodos ágeis e automatizando tarefas manuais. Segundo (MORO, 2008), as organizações de *software* estão cada vez mais preocupadas com a qualidade dos produtos que desenvolvem. Essa preocupação se dá pelo fato de estarem em um mercado bastante competitivo.

De acordo com (FUGGETTA, 2000), o campo de pesquisa de processo de desenvolvimento de *software* cresceu durante os anos 80 para lidar com a crescente complexidade e criticidade das atividades de desenvolvimento de *software*. A suposição é que existe uma correlação direta entre a qualidade do processo e a qualidade do *software* desenvolvido. Por isso, busca-se ter um processo organizado e de qualidade.

Segundo (COHN, 2000), as equipes ágeis, ou seja, aquelas que aplicam os métodos ágeis, estão produzindo *software* com maior qualidade, que atendem melhor as necessidades dos clientes, com mais rapidez e a um custo menor do que as equipes tradicionais. Posto isto, esta pesquisa trata de um estudo de caso sobre os resultados da automatização de processos de desenvolvimento de um produto da GE (*General Electric*), realizado de janeiro a dezembro de 2020.

Para desenvolver a automatização de processos deste estudo de caso, utilizam-se algumas ferramentas como o Jenkins (JENKINS, 2021) para criar um *pipeline* (KIM et al., 2018) de automatização; *Docker* (DOCKER, 2021), para unificar o ambiente de desenvolvimento; e Jira (ATLASSIAN, 2020a), para gerenciar o projeto utilizando os métodos ágeis. Assim, aplicaram-se a cultura, as práticas e as ferramentas necessárias para atingir uma maturidade *DevOps*.

Este trabalho apresenta-se estruturado da seguinte maneira: no Capítulo 2, será abordado os conceitos e ferramentas para a realização deste trabalho; no Capítulo 3, a metodologia aplicada; no Capítulo 4, será descrito o estudo de caso da GE e por fim os resultados obtidos; no Capítulo 5, as considerações finais.

1.1 Justificativa

Para atender às expectativas dos clientes, atualmente a criação de um novo produto requer um desenvolvimento ágil, visando acelerar o lançamento de novas versões e diminuir os custos desnecessários.

No processo de desenvolvimento de *software*, encontram-se muitas dificuldades como: falta de padronização de ambientes de desenvolvimento, falta de comunicação entre os times, barreiras entre os times de teste e desenvolvimento, além de processos manuais e repetitivos. Todas essas dificuldades têm como consequência a demora no lançamento de novos produtos e desperdício de recursos.

A importância deste trabalho se justifica em solucionar estes problemas citados, utilizando primeiramente um estudo de caso de um único produto para então analisar os benefícios e implantar para os demais produtos.

Implantando um processo de desenvolvimento automatizado, unifica-se o ambiente de desenvolvimento tanto para o time de desenvolvimento quanto para o time de testes e validações. Isso evita a intervenção humana e possíveis erros durante a realização destes testes e consequentemente ocorre uma melhoria significativa nas entregas de novas versões e novos produtos.

Quando se tem um processo de desenvolvimento de *software* automatizado, torna-se mais viável realizar pequenas entregas de funcionalidades mais frequentemente, pois cada novo recurso estará devidamente testado e com uma qualidade de código necessário para disponibilização. Isso permite que o cliente possa testar algumas funcionalidades e expor *feedbacks* para futuras melhorias de forma mais frequente e ágil.

Segundo (KIM et al., 2018), aplicando o *DevOps* é possível reduzir o tamanho das entregas e os intervalos de trabalho, incorporar qualidade, evitar que defeitos passem adiante, acelerar a fluidez, reduzir o tempo de execução exigida para atender pedidos de clientes internos e externo, aumentar a qualidade do trabalho, tornar o trabalho mais ágil.

Analisando todos esses benefícios, as empresas têm investido cada vez mais na automatização de processos baseados no conceitos de *DevOps, Integração contínua e Entrega contínua (Continuous Integration and Continuous Delivery - CI/CD), Agile*, que tem como objetivo agregar valor, reduzir custos, aumentar o retorno financeiro, velocidade na entrega e também propor um maior fluxo entre as equipes envolvidas em um projeto.

1.2 Definição do problema

De acordo com (HUMBLE; MOLESKY, 2011), a redução do risco de erros no processo de entrega é melhor alcançada por meio do uso da automatização dos processos, ao invés de processos isolados e manuais. Em um *pipeline* de implantação totalmente automatizado, cada comando necessário para construir, testar ou implantar uma parte de *software* é registrado, junto com sua saída, permitindo um registro de eventos para analises posteriores. A automação também permite testes frequentes, antecipados e abrangentes.

Desta forma, esta pesquisa questiona: é viável a automatização de processos? Ela possui um resultado satisfatório? Quais são os benefícios?

1.3 Objetivo geral

Este trabalho tem como objetivo geral descrever os processos de desenvolvimento de um *software* embarcado tradicional, a fim de compará-los com o processo de desenvolvimento de um *software* embarcado com automatização de tarefas, e analisar seus resultados.

1.4 Objetivos específicos

Para a construção da proposta, dividiu-se o trabalho de acordo com os seguintes objetivos específicos:

a) apresentar os conceitos teóricos e culturas associadas à automatização de processos;

b) descrever os processos de desenvolvimento tradicional de um software embarcado;

c) descrever os processos e etapas de automatização realizados em um sistema embarcado;

d) analisar os resultados obtidos com a automatização de um *software* embarcado.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo visa apresentar alguns conceitos de automatização de processos de desenvolvimento de *software* embarcado em que este trabalho baseou-se e abordar algumas ferramentas utilizadas.

2.1 *DevOps*

DevOps é a junção de dois termos em inglês *Development* e *Operation*, ou seja, é uma relação entre a equipe de desenvolvimento e equipe de operação (infraestrutura e sistemas), com o objetivo de melhorar o fluxo de trabalho. Ambas as equipes possuem um único objetivo: entregar um *software* funcionando com qualidade e de forma ágil.

"*DevOps* constitui em práticas técnicas e não técnicas que ajudam as empresas de software a aumentar a capacidade de resposta às necessidades do cliente por meio de lançamentos de novas versões de software frequentes e automatizadas." (LWAKATARE et al., 2016).

Atualmente o *DevOps* tem sido um assunto muito abordado e investido dentro das empresas que almejam uma rapidez para entregar novas funcionalidades ou novos produtos aos seus clientes, principalmente quando se fala em empresas que trabalham com *software as a service (SaaS)*, nas quais são soluções que não dependem de *hardware* e que oferecem serviços por meio da internet. Nessas empresas a cultura do *DevOps* é muito utilizada, pois é possível automatizar quase todas as etapas. Já no *software* embarcado, por envolver *hardware*, sempre há etapas que não possam ser automatizadas, por exemplo os testes em *hardware* (LWAKATARE et al., 2016).

O termo *DevOps* surgiu para minimizar os problemas entre as equipes de desenvolvimento, testes e infraestrutura. Um exemplo deste conflito é quando o desenvolvedor utiliza um ambiente local para codificar e, após o desenvolvimento, este programa é submetido para equipe de testes. Um dos possíveis problemas entre essa interação é a falta de padronização entre estes ambientes. Desta forma, um dos objetivos do *DevOps* é facilitar essa comunicação entre as equipes e ter um ambiente de desenvolvimento, operação e testes padronizados.

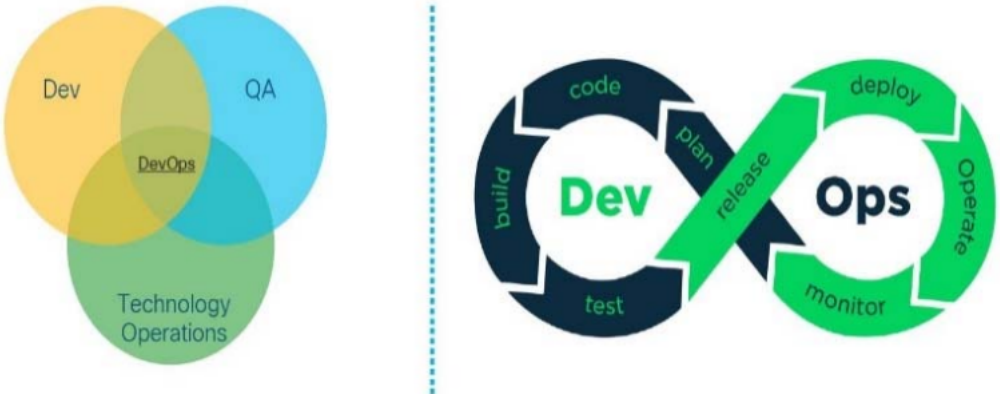
DevOps é uma cultura aplicada a todos os times que estão envolvidos em um produto: equipe de desenvolvimento, infraestrutura/operação e testes. Essa cultura incentiva o trabalho em equipe e a comunicação clara entre os times envolvidos. Um dos princípios do *DevOps* é automatizar o maior número de tarefas possíveis entre essas equipes e padronizar o ambiente de desenvolvimento.

"Este conceito oferece vantagens para ambos os lados, pois fornece atualizações contínuas, resolve erros mais rapidamente e reduz as falhas de comunicação que

podem ocorrer dentro da equipe."(WIJAYA; ROSYADI; TARYANA, 2019, p. 2, tradução nossa).

O símbolo do infinito, mostrado na Figura 1 é utilizado para representar os processos do *DevOps*. Analisando a Figura 1, também é possível observar o principal conceito do *DevOps*: unir a equipe de DEV (desenvolvimento), QA (Qualidade/testes) e *Technology Operation* (operação). A equipe de desenvolvimento é responsável por desenvolver a aplicação, a equipe QA faz os testes necessários para validar uma nova funcionalidade desenvolvida e a de operação é responsável pela infraestrutura (computadores, servidores, máquinas, por exemplo).

Figura 1 – *DevOps lifecycle*



Fonte: (SHAH; DUBARIA, 2019).

Observa-se que as tarefas de código, compilação e testes são tarefas da equipe de DEV. Já as tarefas de publicação, operação e monitoramento são da equipe de operação.

Analisando a Figura 1, a interseção no símbolo de infinito representa as etapas de planejamento e lançamento de novas versões, que é comum entre as equipes de operação e desenvolvimento. Com base na Figura 1 as etapas do *DevOps* são:

a) Planejar (*Plan*) - planejar atividades e metas.
b) Codificar (*Code*) - desenvolver a aplicação em pequenas partes.
c) Construir (*Build*) - geração do código binário.
d) Testar (*Test*) - testar código para identificar falhas e *bugs* no sistema.
e) Lançar (*Release*) - lançar uma funcionalidade.
f) Publicar (*Deploy*) - publicar em um repositório.
g) Implementar (*operate*) - validar a aplicação já testada com produto final.

h) Monitorar (*monitor*) - monitorar os membros e o desempenho do projeto. Nessa etapa é onde se encontra problemas e gargalos no processo de desenvolvimento.

Esse ciclo repete-se continuamente, por isso é representado por um infinito. Sempre se inicia pela etapa de planejamento.

Para aplicar o *DevOps*, é possível utilizar algumas ferramentas que auxiliarão neste processo como: *containers*, versionadores de código, repositórios de binário, ferramentas de gerenciamento de projetos, ferramenta para automação de processos de desenvolvimento.

2.2 Ferramenta *GitHub*

Segundo o site oficial do GitHub (2021), esta é uma plataforma de hospedagem de código e controle de versão e colaboração, permitindo que mais de uma pessoa trabalhem juntas em um mesmo projeto, de qualquer lugar.

Quando se utiliza o *GitHub* é mais simples acompanhar as mudanças feitas no código, pois ele registra quem fez a modificação e permite que se restaurem versões anteriores. Grandes e pequenas empresas têm utilizado cada vez mais a ferramenta, principalmente nas equipes de desenvolvimento.

O *GitHub* utiliza alguns conceitos que serão abordados nesta seção, como conceito de repositórios, *branch*, *commit*, *pull-request* e TAGs.

O **repositório** é utilizado para organizar um projeto, semelhante a uma pasta que contém tudo o que um projeto precisa. Segundo o site do GitHub (2021), os repositórios podem conter pastas e arquivos, imagens, vídeos, planilhas e conjuntos de dados, e tudo que o projeto precisar. Geralmente cada repositório contém um *README* que é um arquivo no formato *Markdown* que descrever sobre o que o projeto se trata e como utilizar os códigos que estão no repositório.

Outro conceito muito importante é o conceito de ***branch***, sendo a maneira de trabalhar em diferentes versões de um repositório ao mesmo tempo (GITHUB, 2021). Todo repositório possui uma *branch* chamada *main*, que é a *branch* principal e definitiva. Sempre que se deseja fazer uma alteração no código ou desenvolver uma nova funcionalidade, é utilizada uma nova *branch*, a qual será uma cópia da *branch* principal. Caso algum outro desenvolvedor faça uma alteração na *branch* principal, é possível atualizar a nova *branch* para a versão mais atual.

Cada novo código incluído ou alteração salva no *GitHub* possui o nome de ***commit***, o qual é reconhecido não pelo nome do arquivo, mas sim pelo *commit hash*, que é uma sequência de caracteres a qual mapeia os dados; cada *commit* possui um *hash* único. Cada *commit* tem uma mensagem associada, na qual é uma descrição que

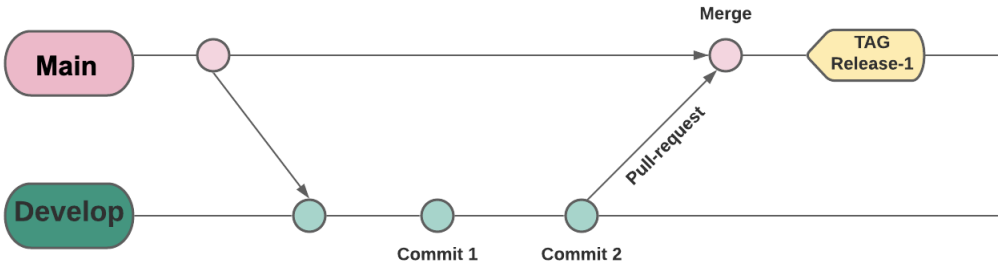
explica por que uma mudança foi feita. As mensagens de *commit* mostram o histórico de suas alterações, para que outros desenvolvedores possam entender o que foi feito (GITHUB, 2021).

Após uma alteração ter sido feita, ela estará sempre em uma *branch* diferente da *branch main*. Para levar esta nova alteração ou inclusão de código para a *branch main*, é necessário abrir um ***pull-request***, que é uma solicitação para mesclar sua *branch* com a *branch main*. Antes disso, outro desenvolvedor deve analisar o código e deixar seus comentário de alteração ou aprovar este *pull-request*. Conforme o site do GitHub (2021), os *pull-requests* mostram diferenças do conteúdo de ambas as *branchs* e as alterações, adições e subtrações são mostradas em verde e vermelho, respectivamente. Após um *pull-request* ser aprovado, é feito o *merge*, que ocorre quando a *branch* criada é mesclada com a *branch main*.

Uma **TAG** é uma marca em um determinado *commit* de uma *branch*, geralmente a TAG é utilizada para marcar uma mudança significativa, frequentemente usada quando possui uma nova versão de *release*. As TAGs são imutáveis, mesmo que ocorram mais *commits* nessa *branch* ela sempre apontará apenas para um determinado *commit*. Com a TAG é possível voltar a alguma versão de um projeto caso seja necessário.

A Figura 2 ilustra os conceitos abordados. É possível observar que inicialmente cria-se uma nova *branch* a partir da *main* e são feitas as alterações e os *commits* necessários. Após isso, é aberto um *pull-request* que, depois de ser aprovado, é feito o *merge* para a *branch main*, e se necessário for, cria-se uma TAG para marcar uma nova release.

Figura 2 – Exemplo utilização do *GitHub*



Fonte: Elaboração própria (2021)..

Utilizar a ferramenta *GitHub* traz inúmeros benefícios como: controle de versão, ambiente de desenvolvimento colaborativo e histórico de *releases* utilizando as TAGs. Com isso, cada vez mais as empresas têm investido em ferramentas como o *GitHub*.

2.3 Métodos Ágeis

Os métodos ágeis estão ganhando cada vez mais popularidade e grandes multinacionais estão aplicando-o.

O Manifesto Ágil é a base das Metodologias Ágeis, e segundo Agile Manifest (2021) possuem 4 valores:

a) **Indivíduos e interação** mais que processos e ferramentas.

b) ***Software* em funcionamento** mais que documentação extensa.

c) **Foco no cliente** mais que negociação de contrato.

d) **Responder a mudanças de requisitos** mais que seguir um plano.

O método ágil tem um foco grande no cliente e no que o cliente espera do produto final, por isso é um método que responde bem a mudanças de requisitos, que acontecem com frequência.

De acordo com Gomes (2014) pelo fato do método ágil assumir a imprevisibilidade envolvida no desenvolvimento de *software*, eles baseiam-se no ciclos de *feedback* constante, que permite que o cliente e a equipe de desenvolvimento possam adaptar-se a mudanças, aumentando a chance de sucesso do projeto e satisfação do cliente final.

Segundo Gomes (2014), uma das motivações para aplicar o método ágil são os benefícios que este método gera:

a) Melhora o *time to market*.

b) Maior satisfação do cliente - proximidade com o cliente resulta em um melhor alinhamento dos objetivos.

c) Melhor visibilidade do projeto pela equipe de gestão.

d) Maior produtividade do time e melhor qualidade das entregas.

e) Redução de riscos - entrega frequente permite ajustar constantemente os requisitos.

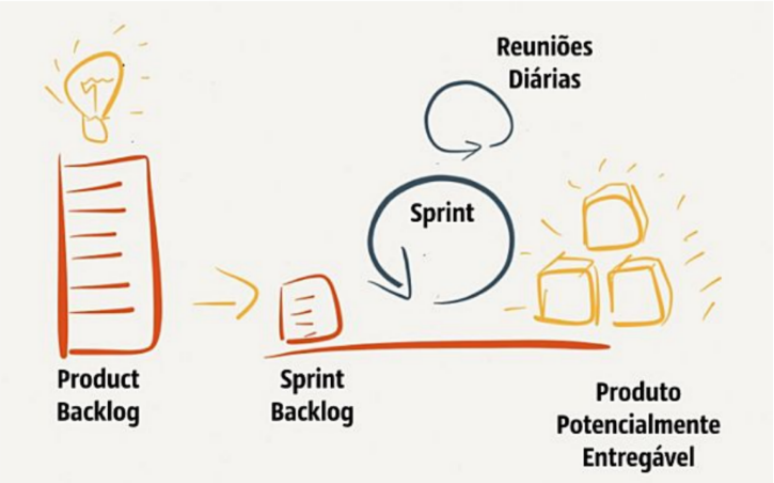
f) Redução de custos - menor chance de desenvolver funcionalidades desnecessárias.

Nas Subseções 2.3.1 e 2.3.2 serão abordados dois métodos ágeis muito utilizados, o *Scrum* e o *Kanban*, respectivamente.

2.3.1 *Scrum*

O funcionamento do *scrum* é ilustrado pela Figura 3.

Figura 3 – Scrum funcionamento



Fonte: (GOMES, 2014).

Um dos mais aplicados métodos ágeis é o *Scrum*, que é dividido por ciclos, os quais recebem o nome de *sprint*, que possuem geralmente possuem duração de 2 semanas. A cada começo de *sprint* é feita uma reunião, que se chama *Sprint Backlog*, em que é feito o levantamento das funcionalidades que serão implementadas na *sprint* e serão escritas no *Product Backlog*.

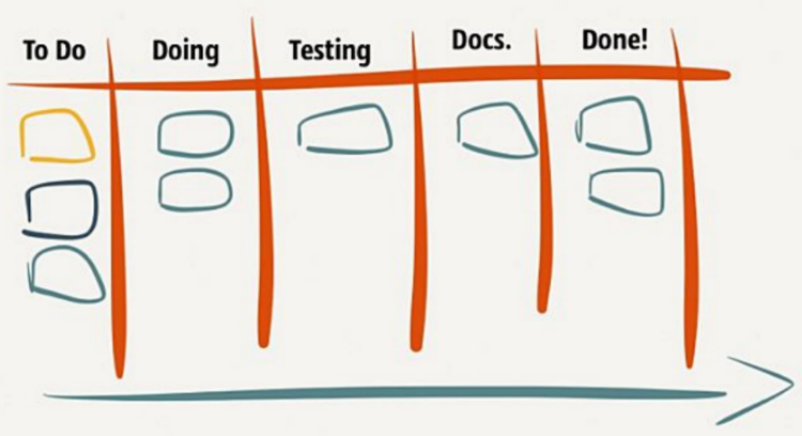
Todos os dias, a equipe faz uma rápida reunião para conversar sobre os avanços de cada integrante do time, o que foi feito no dia anterior, o que será feito no dia atual e analisar se as prioridades do projeto estão corretas. Essa reunião chama-se *Daily Meeting* ou reuniões diárias.

Ao final de uma *sprint*, é realizada a reunião de *Review Meeting* e *Sprint retrospective*, em que são apresentadas as funcionalidades implementadas e se planeja a próxima *sprint*, com isso, o ciclo recomeça.

2.3.2 *Kanban*

O *Kanban* (ARBULU; BALLARD; HARPER, 2003) é um método ágil que visa à organização, produtividade e otimização da gestão. Ele utiliza um quadro ilustrado pela Figura 4, que é composto pelas seguintes etapas: *To do* (a fazer), *Doing* (Fazendo), *Testing* (Testando), *Docs* (documentação), *Done* (feito). Os desenvolvedores e os gestores colocam as tarefas que precisam ser feitas neste quadro *Kanban* e, conforme as tarefas vão sendo realizadas elas mudam de coluna. Este quadro facilita a gestão e análise visual do projeto.

Figura 4 – Quadro *Kanban*



Fonte: (GOMES, 2014).

O *Kanban* sugere um limite de 3 tarefas em progresso para cada integrante da equipe visando um bom fluxo de trabalho e prevenindo sobrecarga.

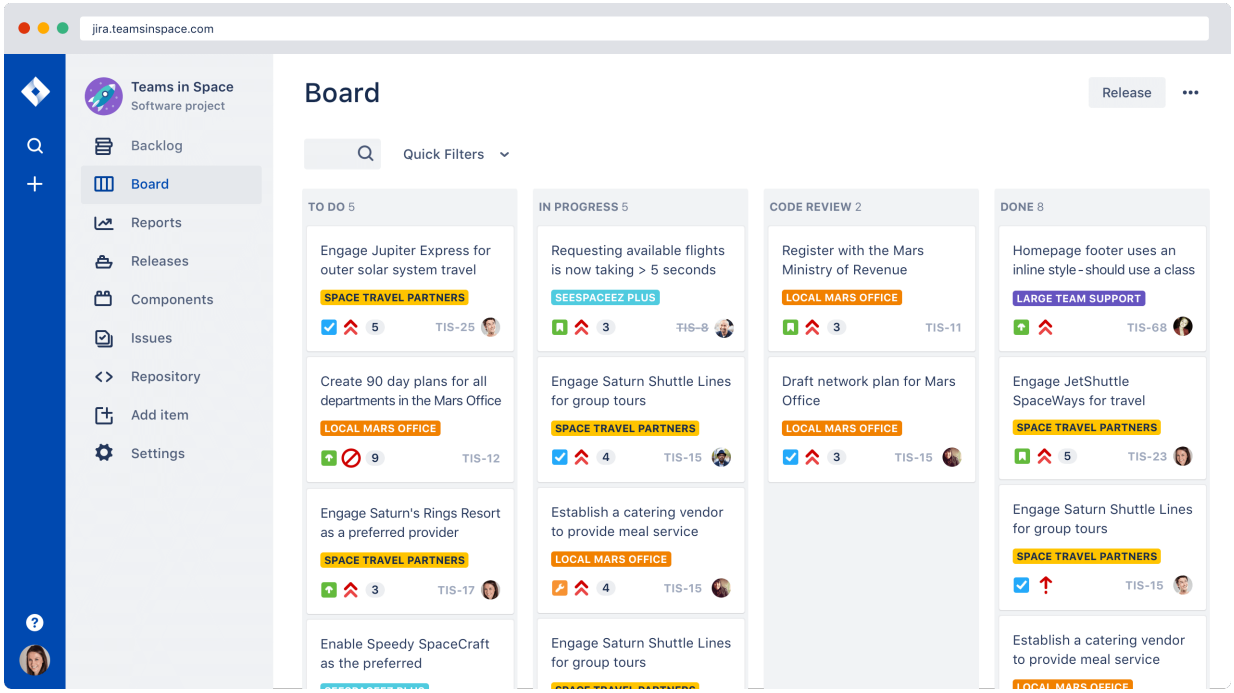
De acordo com Gomes (2014), *kanban* é um fluxo contínuo e não é preciso que todo trabalho planejado seja concluído para fazer uma entrega ao cliente ou a produção. Neste caso, as tarefas que estão prontas fazem parte da entrega e as que estão em progresso ficam para a próxima entrega.

2.4 Ferramenta Jira *Software*

A ferramenta Jira *software* foi desenvolvida pela empresa *Atlassian*. De acordo com o site oficial do *software* Jira (ATLASSIAN, 2020a), o *software* é uma ferramenta de gerenciamento ágil de projetos que oferece suporte a qualquer metodologia ágil, como o *Scrum* e o *Kanban*.

O *software* Jira possui quadros para aplicar as metodologias ágeis. É muito utilizado para gerenciar tarefas que estão em andamento, tarefas já realizadas e as que precisam ser feitas. Com os quadros é possível ter transparência das tarefas de cada desenvolvedor e do time. A Figura 5 ilustra um exemplo de um quadro no Jira com tarefas.

Figura 5 – Exemplo de quadro no software Jira



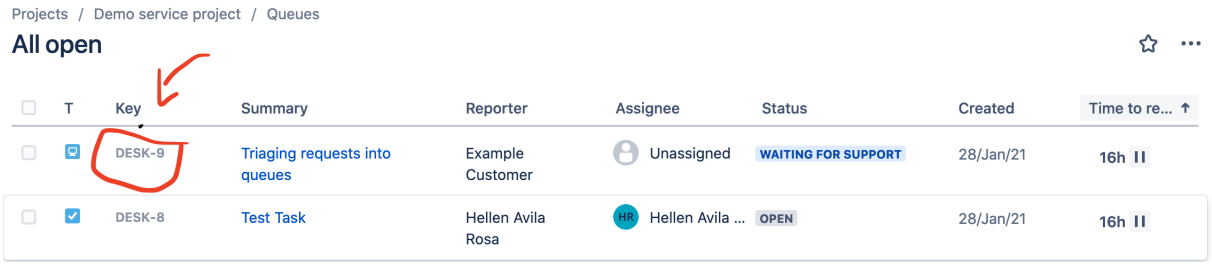
Fonte: (ATLASSIAN, 2020a).

A Figura 5 representa o quadro *kanban*, ele é dividido em: *To do* (Para fazer), *in progress* (Em progresso), *code review* (Revisão de código), *Done* (Feito), cada bloco branco representa uma atividade alocada para um desenvolvedor específico. A partir desse quadro, também é possível gerar relatórios e gráficos para monitorar a produtividade do time ao longo do tempo. É uma ferramenta muito poderosa para a gestão de projetos.

Um gestor de projetos cria as tarefas, que no Jira são chamadas de *issue*; dentro desta tarefa, é preenchido a estimativa de tempo para a conclusão, descrição detalhada da *issue*, prioridade e quem será o responsável por realizar a tarefa.

Para cada *issue* criada, é gerada uma chave única que é composta por letras e números. Com essas informações, é possível ter um detalhamento das tarefas, mesmo que passe um longo período de tempo. A Figura 6 ilustra uma lista de *issues* com suas chaves de identificação.

Figura 6 – Exemplo de quadro no software Jira



Fonte: (ATLASSIAN, 2020a).

O Jira possui integração com uma gama de ferramentas. Uma delas é a ferramenta *GitHub*, em que é possível vincular uma *issue* a uma *branch* do Jira, sendo possível observar quantos *commits* foram feitos e quantos *pull-requests* foram abertos para resolver esta tarefa.

Com essas informações torna-se mais visível o progresso do projeto e se possibilita a análise das *issues* implementadas quando uma nova versão é lançada.

2.5 Ferramenta de Automação

De acordo Seth e Khare (2015), Integração contínua (CI) é a prática mais comum entre desenvolvedores de *software*, que podem integrar seu trabalho em uma linha contínua, ou seja, reunir e testar os códigos desenvolvidos de forma automática. A indústria está enfrentando enormes desafios ao desenvolver *software* em diferentes ambientes e testá-lo em várias plataformas. A melhor maneira de tornar a integração contínua mais rápida e eficiente é automatizar o processo de compilação e teste utilizando ferramentas. Ou seja CI é a prática de sempre integrar o código de todos os desenvolvedores em um único repositório e testa-lo e CD é entregar o *software* para o ambiente de produção de forma automática.

Para *software* embarcado geralmente não possui a etapa de CD, pois é necessário realizar testes em *hardware*, por isso não é possível desenvolver um *software* e envia-lo para produção de forma automática.

2.5.1 Ferramenta Jenkins

Jenkins é um dos poucos servidores de automação gratuito e de código aberto, sendo desenvolvido na linguagem Java. Com ele é possível criar *pipelines CI/CD*, na qual é uma série de etapas que são descritas e que devem ser realizadas até uma nova versão ser disponibilizar.

Segundo Seth e Khare (2015), Jenkins é uma ferramenta de integração contínua que auxilia na automatização de todo o processo, reduzindo o trabalho de um desenvolvedor e checando o desenvolvimento a cada etapa da evolução do *software*.

A implementação do Jenkins *pipeline* é escrita em um arquivo chamado "Jenkinsfile", que geralmente é encontrado em um repositório de versões, como o *GitHub*. O Jenkins tem suporte para uma gama de *plugins*, tornando viável a comunicação entre diversas plataformas.

Com o código a seguir, é possível observar o modelo de um Jenkinsfile e sua sintaxe. Nota-se que o *pipeline* é executado por *stages* (estágios); sempre que um *stage* retorna um erro, a execução do restante dos *stages* é cancelada. Na linha 5, 11 e 17 temos os estágios de compilação, testes e publicação; respectivamente.

```
1   pipeline {
2       agent any
3
4       stages {
5           stage('Compilacao') {
6               steps {
7                   echo 'Compilando...'
8                   sh 'make build'
9               }
10          }
11          stage('Testes') {
12              steps {
13                  echo 'Testando...'
14                  sh 'make check || true'
15              }
16          }
17          stage('Publicacao') {
18              when {
19                  expression {
20                    currentBuild.result == null || currentBuild.result == '
    SUCCESS'
21                  }
22              }
23              steps {
24                  echo 'Publicando...'
25                  sh 'make publish'
26              }
27          }
28      }
29  }
30
```

Código 2.1 – Jenkinsfile

Geralmente, o Jenkins *pipeline* possui os seguintes *stages*: compilação, testes e publicação. Pode ser configurado e adicionado conforme a necessidade do projeto. No Jenkins, é possível adicionar condições para cada *stage*, configurar envio de *email* para notificação e executar *scripts* nos *stages*.

É possível também escolher os *triggers* para executar o Jenkinsfile. Os *triggers* são condições que, se atendidas, executam o *Jenkinsfile*. Alguns exemplos de *triggers* são: executar o Jenkinsfile quando uma nova *branch* do *GitHub* é criada, quando ocorre algum novo *commit* em uma *branch* ou quando uma nova TAG é criada. Essa configuração é feita no Jenkins e pode ser configurada conforme for a necessidade da aplicação.

De acordo com Jenkins (2021), à medida que o *pipeline* é utilizado para mais projetos em uma organização, é provável que surjam padrões comuns. Muitas vezes, é útil compartilhar partes de *pipelines* entre vários projetos para reduzir redundâncias. Com isso, o *pipeline* tem suporte para a criação de *shared libraries*.

As *shared libraries* são bibliotecas compartilhadas e desenvolvidas na linguagem *Groovy* e que podem ser usadas por diferentes Jenkinfiles. Ela é incluída no cabeçalho do Jenkinfile e deve conter apenas funções genéricas; por isso, é desenvolvida em um repositório específico para ela.

O Jenkins possui uma variedade de *plugins* e configurações para automatizar e facilitar o processo de desenvolvimento de uma aplicação. Utilizando o Jenkins é possível otimizar o tempo do desenvolvedor, fazendo com que ele possa focar no desenvolvimento da aplicação e não em tarefas repetitivas e manuais.

2.6 *Docker*

O *Docker* é uma plataforma de código aberto, escrito na linguagem de programação Go, que é desenvolvida na Google. De acordo com o site oficial do *Docker* (DOCKER, 2021), em 2013, o *Docker* apresentou o que se tornaria o padrão da industria para *containers*. Os *containers* são uma unidade padronizada de *software* que permite isolar seu aplicativo de seu ambiente. Para milhões de desenvolvedores hoje, o *Docker* é o padrão para construir e compartilhar aplicativos em *containers*, do computador à nuvem.

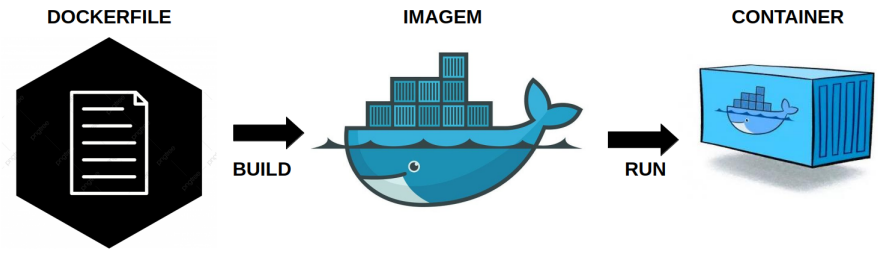
As empresas têm optado por utilizar os *containers* por serem isolados e portáveis. Permite-se que um desenvolvedor empacote a aplicação com as bibliotecas e pacotes necessários e os demais desenvolvedores baixem essa aplicação empacotada para ser facilmente executada em seus computadores. O benefício é extinguir o problema da maioria dos desenvolvedores: compila na máquina do desenvolvedor 1 e não compila na máquina do desenvolvedor 2, por isso o *Docker* facilita o trabalho em

equipe e simplifica a aplicação da metodologia *DevOps* e o desenvolvimento ágil.

Os *containers* compartilham o *kernel* do sistema operacional do *host*, da mesma maneira compartilham das bibliotecas e binários. Geralmente os *containers* possuem um tamanho em torno de MB, diferente de uma Máquina Virtual que possui tamanho na ordem de GB, por isso ele é facilmente portável e inicializa rapidamente.

O processo para utilização de um *Docker Container* está descrito na Figura 7

Figura 7 – Processo para gerar um *Docker Container*



Fonte: Elaboração própria (2020).

Para utilizar a ferramenta *Docker*, é necessário desenvolver um *Dockerfile* que descreve tudo que é necessário para executar a aplicação, as dependências, o sistema operacional que usará como base e os comandos que serão executados. O código a seguir representa um exemplo de um *Dockerfile*.

Nota-se que a primeira linha do Dockerfile é o sistema operacional que esse *container* irá utilizar, nesse exemplo um Linux Debian, posteriormente é feito um comando de *COPY* que permite copiar os arquivos para dentro do *container*, por fim é utilizado o comando *RUN* que atualiza o sistema operacional e instala os pacotes de *python* e *curl* . O Dockerfile é descrito por camadas, cada linha é considerada uma camada. Com isso, é simples fazer atualizações de comandos ou exclui-los, pois eles alteram somente a camada modificada, não tendo necessidade de criar toda a imagem do zero.

```
31  FROM debian
32
33  COPY build/bin/ /usr/local/bin
34
35  RUN set -ex && apt-get update && apt-get install -y \
36  python \
37  curl
38
```

Código 2.2 – Exemplo de um Dockerfile

Após isso, é permitido fazer *upload* dessa imagem *Docker* para um repositório de imagens *Docker* na nuvem. Isso facilita a utilização desta imagem posteriormente, pois qualquer desenvolvedor que tenha permissão para baixá-la, poderá utilizar em seu ambiente.

São inúmeros os benefícios de se utilizar o *container*, pois ele permite uniformizar os ambientes, e consequentemente estimular as práticas de *DevOps*; melhora a comunicação e facilita o trabalho da equipe de infraestrutura, além de ser facilmente portáveis. Os *containers* também são mais leves do que uma máquina virtual e facilitam a arquitetura de micro serviços.

2.7 Testes de *software*

A etapa de testes é muito importante no desenvolvimento de *software*. É nesta etapa que garante-se a qualidade do código e é analisado se a aplicação está cumprindo os requisitos do projeto.

De acordo com Neto e Dias (2007), o teste de *software* é o processo de execução do produto visando analisar se ele atingiu suas especificações e funcionou corretamente no ambiente para o qual foi projetado. O seu objetivo é revelar falhas em um produto e que possam ser corrigidas pela equipe de desenvolvimento antes da entrega final.

Os principais níveis de testes para *software* são: testes unitário, testes de integração, testes de *software* e análise estática de código.

Segundo Neto e Dias (2007), os **Testes unitários** têm por objetivo explorar a menor unidade do projeto, procurando analisar falhas ocasionadas por erros de lógica e de implementação em cada função separadamente. Os testes unitários são desenvolvidos pelo programador e geralmente se utiliza de ferramentas de testes unitários para executá-los. As ferramentas são programadas para uma entrada e saída prevista, caso a saída seja diferente do esperado, o teste falha.

Os **testes de integração**, de acordo com Neto e Dias (2007), visa analisar falhas associadas entre os módulos quando esses são integrados para construir a estrutura do *software* que foi projetado. Normalmente sucedem os testes unitários, visando analisar se suas entradas e saídas estão como o planejado. É um teste que analisa se uma nova funcionalidade modificou o funcionamento de outra funcionalidade. Também utiliza-se ferramentas para este tipo de teste.

De acordo com Neto e Dias (2007), o **teste de sistema**, avalia o *software* em busca de falhas por meio da utilização do mesmo, como um usuário final. Dessa maneira, os testes são executados nos mesmos ambientes, com as mesmas condições e com os mesmos dados de entrada que um usuário utilizaria. Tal teste visa verificar se

o produto satisfaz seus requisitos de projeto.

Já a **análise estática** possui uma abordagem diferente. De acordo com Terra e Bigonha (2008), os verificadores estáticos são ferramentas de *software* que varrem o código fonte e possuem a intenção de ressaltar anomalias no código, como variáveis usadas sem serem inicializadas, variáveis não usadas ou atribuições cujos valores poderiam ficar fora de seus limites.

2.8 Resumo Geral das ferramentas utilizadas

As ferramentas utilizadas para a automatização dos processos de desenvolvimento estão descritas na Tabela 1.

Tabela 1 – Ferramentas utilizadas no processo de automatização

Ferramenta	Função
GitHub	Hospedagem de código com controle de versões
Jenkins	Ferramenta para automatização de desenvolvimento de *software*.
Jira	Ferramenta de gestão ágil de projetos, ou seja, permite aplicar os métodos ágeis.
Docker	Possibilita empacotamento de aplicações dentro de um *container*, tornando-se portátil.

Fonte: Elaboração própria (2021).

Com isso, para alcançar os objetivos traçados utilizou-se as ferramentas: *GitHub*, Jenkins, *Docker* e Jira.

3 METODOLOGIA

Este trabalho apresenta um estudo de caso sobre processos de desenvolvimento de *software* embarcado utilizando ferramentas de automatização. A pesquisa foi realizada durante o processo de desenvolvimento de um produto na GE (*General Electric*) de Florianópolis e possui o objetivo de automatizar e analisar os benefícios e resultados da automatização dos processos de desenvolvimento.

Desta forma, esta pesquisa pode ser classificada como exploratória com abordagem qualitativa.

Segundo Gil (2002), uma pesquisa exploratória tem como objetivo proporcionar maior familiaridade com o problema, com vistas a torná-lo mais explícito ou a constituir hipóteses. Já a abordagem qualitativa, de acordo com Malhotra (2010), consiste em uma metodologia de pesquisa não estruturada e exploratória baseada em pequenas amostras que proporciona percepções e compreensão do contexto do problema.

Para a fundamentação teórica utilizou-se a pesquisa do tipo bibliográfica, segundo Gil (2002), ela é desenvolvida com base em material já elaborado, principalmente livros e artigos científicos.

Na Seção 3.1 serão abordados os métodos aplicados neste estudo de caso.

3.1 Métodos aplicados

Neste trabalho foram utilizadas algumas ferramentas para que fosse possível alcançar um resultado satisfatório, como o Jenkins, *GitHub*, *Docker* e Jira.

Umas das ferramentas principais para este estudo de caso foi o *software* Jenkins, que é uma ferramenta para automatização do processo de desenvolvimento de um *software*. Com esta ferramenta, foi possível construir um *pipeline* para automatizar o processo de compilação, testes, publicação de uma nova versão do produto e notificação a equipe de testes e validações.

Utilizou-se a ferramenta *GitHub* para hospedar o código, fazer o controle de versões e facilitar o trabalho em equipe, pois ele permite um repositório de códigos colaborativo.

Para padronizar o ambiente de desenvolvimento e testes, utilizou-se a ferramenta *Docker* que permite empacotar a aplicação juntamente com suas dependências. Esta ferramenta foi essencial para alcançar os objetivos do projeto.

Por fim, utilizou-se o *software* Jira para gestão ágil do projeto. Nesta ferramenta é possível organizar quem irá realizar determinada tarefa, em quanto tempo ela precisa ser desenvolvida e gerar gráficos para analisar o andamento do projeto.

Em resumo, as etapas para alcançar os resultados obtidos neste estudo de caso foram:

a) Utilização da ferramenta *Docker* criando um *dockerfile* para empacotar a aplicação com as dependências.

b) Configuração do Jenkins e das ferramentas que possuem integração com o Jenkins, como o *GitHub* e *software* Jira.

c) Criação do *Jenkinfile* para automatizar os processos de desenvolvimento.

d) Desenvolvimento de uma *shared library* com funções como: analisar a TAG criada no *GitHub*, deletar a TAG se ela estiver errada, envio de *email* quando a TAG está errada, envio de *email* para equipe de teste quando ocorre a publicação de uma nova versão, funções para obter as informações das modificações feitas naquela versão.

e) Criação e configuração dos testes unitário e de análise estática de código.

f) Configuração e criação dos *triggers* do *pipeline* do Jenkins;

g) Configuração do *software* Jira para integrar com os repositórios do *GitHub*

Após o desenvolvimento destas etapas, inicia-se o desenvolvimento de uma nova funcionalidade; depois disso, é enviado este código para o *GitHub*, e o Jenkins é acionado, realizando as etapas de compilação e testes. Quando se implementam todas as funcionalidades para lançar uma nova versão, é criado uma TAG no *GitHub*, e com isso, é acionado o Jenkins para realizar a compilação, testes, publicação desta versão e é enviado um *email* de notificação para equipe de testes. Sendo assim as tarefas manuais realizadas pelos desenvolvedores foram automatizadas.

Quanto à escolha destas ferramentas, não foi possível utilizar as que não fossem autorizadas pela GE. Assim, foi possível usar somente as ferramentas liberadas dentro da GE, como o *GitHub*, Jenkins, Jira e *Docker*. Por isso, não foi possível fazer comparações com outras ferramentas ou utilizar outras opções.

4 ESTUDO DE CASO - GE

Nesta seção será abordado o estudo de caso da GE onde foi automatizado o processo de desenvolvimento de um *software* embarcado. Será discutido também o processo de automatização e os resultados obtidos. Na Seção 4.1 será apresentado o estudo de caso da *General Electric* e na 4.2 os resultados obtidos com a automatização de processos para este estudo de caso.

4.1 Estudo de Caso *General Electric* - GE

O estudo de caso abordado é a automatização do processo de desenvolvimento de um produto para subestações digitais, produzido pela GE de Florianópolis. O produto em desenvolvimento, onde se utilizou este estudo de caso, foi o *S20* (GE GRID SOLUTIONS, 2020), um *switch ethernet* utilizado em subestação digital para fazer a comutação de pacotes entre os dispositivos de forma confiável e robusta.

Durante o desenvolvimento deste produto, surgiram as iniciativas de *DevOps*, aplicando métodos ágeis e *pipeline CI/CD*, pois até então, todo o processo era realizado de forma manual. Visando mudar o método de desenvolvimento e automatizar algumas etapas, foram necessárias algumas mudanças de organização, processos e comunicação entre times.

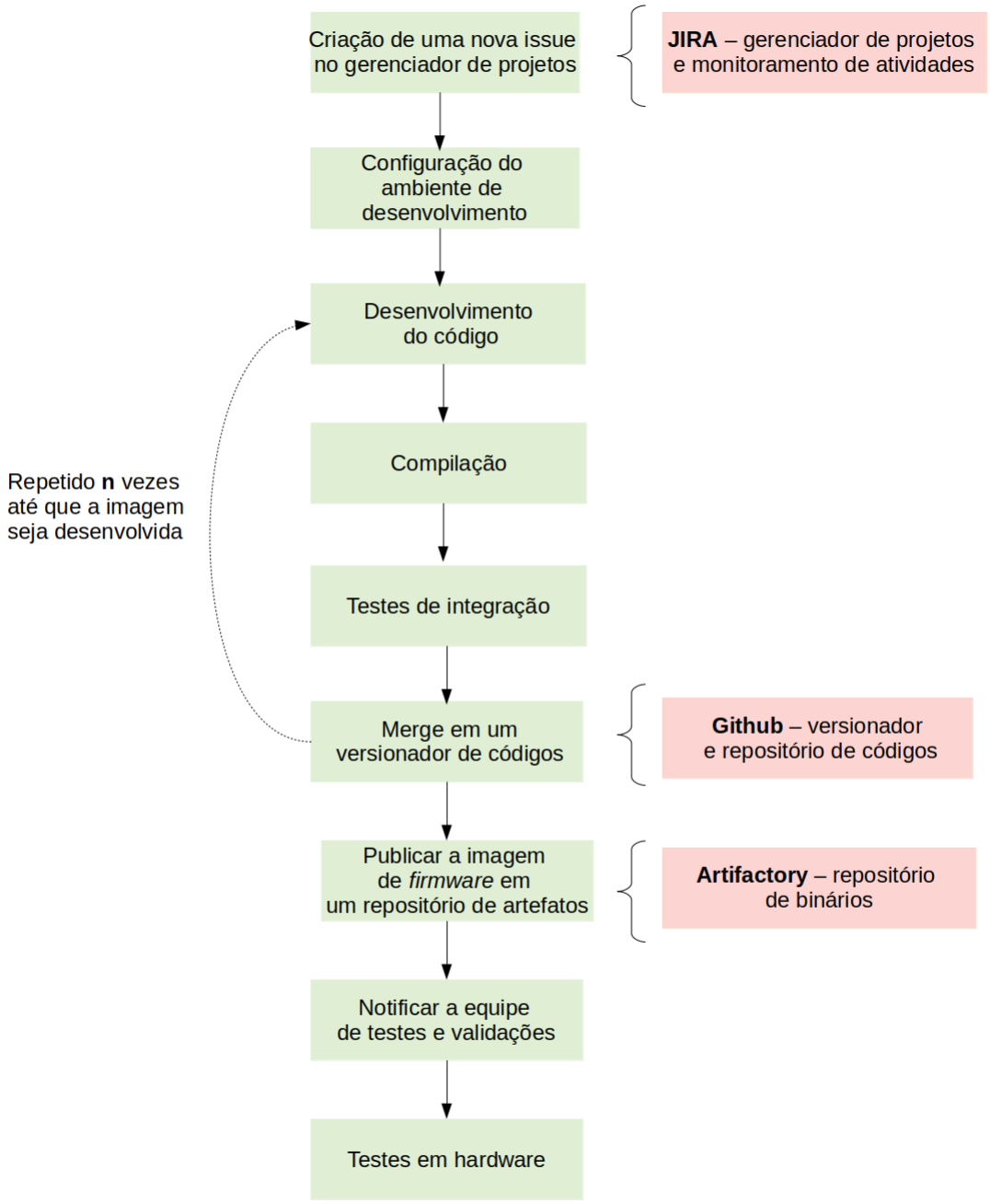
Uma das primeiras iniciativas foi a automatização dos processos de desenvolvimento. Para isso, foi decidido utilizar uma ferramenta que fosse configurável e que pudesse automatizar as etapas do processo. Assim, optou-se por utilizar o *Software Jenkins*, pois é uma ferramenta gratuita, de código aberto e utilizada em outras unidades da GE.

Na Subseção 4.1.1 será abordado o processo manual de desenvolvimento e suas problemáticas e na Subseção 4.1.2, o processo automatizado de desenvolvimento de *software* embarcado.

4.1.1 Processo manual de desenvolvimento de *software* embarcado e suas problemáticas-GE

Analisando o processo de desenvolvimento da imagem de *firmware* dos produtos da GE Florianópolis, foi possível observar as etapas que permitiam ser automatizadas. O diagrama dos processos sem automatização é ilustrado pela Figura 8.

Figura 8 – Processo de desenvolvimento de *software* embarcado tradicional - GE



Fonte: Elaboração própria (2020).

Como ilustra a Figura 8, a primeira etapa do processo de desenvolvimento foi a criação de uma nova *issue* no *software JIRA*, usado para gerenciamento do projeto e acompanhamento das atividades. A cada nova *issue* criada, era preenchido o tempo que levaria para desenvolver, o seu tipo, se era um *bug*, uma melhoria ou uma nova implementação; também eram preenchidas informações detalhadas sobre essa *issue*. Por fim era gerado um código do JIRA referente a essa *issue*; esse código seria

utilizado posteriormente nas mensagens de *commits* na ferramenta do *Github*.

A segunda etapa do processo era configurar o ambiente de desenvolvimento, ou seja, instalar as dependências necessárias para compilar e desenvolver o código, por exemplo: bibliotecas e pacotes. Uma das dificuldades encontradas nessa etapa era a diferença entre os ambientes de desenvolvimento e testes, pois cada desenvolvedor utilizava ferramentas e ambientes diferentes, gerando incompatibilidades, idealmente o código deveria ser compilado em um ambiente limpo e idêntico.

Ao entrar um novo integrante no time de desenvolvimento, era necessário um grande conjunto de configurações no ambiente do desenvolvedor, além de baixar todas as dependências para que fosse possível compilar o código e isso acabou gerando um grande desperdício de recursos. Para mitigar esse problema, utilizou-se o *Docker*.

Após isso, desenvolvia-se o código geralmente utilizando a linguagem de programação C ou C++. Alguns testes foram executados para garantir a performance do código, como testes funcionais e de integração; esses testes eram manuais e executados pelo desenvolvedor. Os testes funcionais são realizados para validar os requisitos do projeto e os testes de integração são executados para garantir que a funcionalidade desenvolvida seja validada em conjunto com as funcionalidades já implementadas anteriormente.

Os testes funcionais e de integração eram sempre executados antes de abrir um novo *pull-request* no GitHub, diferente dos testes unitários e de análise estática de código. Esses geralmente eram executados ao final de uma nova versão, pois são testes custosos. Com isso, a problemática encontrada nesta etapa de testes era descobrir uma quantidade elevada de erros na fase final do projeto, tornando-se mais difícil e demorado para serem solucionados. Dessa forma, observou-se a necessidade de automatizar esta etapa, pois com isso, o testes seriam executados em pequenas partes de código, levando menos tempo e garantindo que os testes fossem executados em todos os novos códigos inseridos no repositório *GitHub*.

Após os testes funcionais e de integração serem feitos, o desenvolvedor analisava o resultado dos testes e abria um *pull request* no *Github*, para que outro desenvolvedor o aprovasse. Posteriormente se realizava o *merge* desse código para a *branch master*. Quando todas as funcionalidades eram desenvolvidas e não havia melhorias a serem feitas, era gerada a imagem de *firmware*.

No final de todo esse processo o desenvolvedor publica a imagem de *firmware* em um repositório de artefatos, neste estudo de caso utilizou-se o *Jfrog Artifactory*. Um artefato é uma parte de um *software*.

Com a imagem publicada, o desenvolvedor notificava a equipe de testes e validações, informando o link para baixar a nova versão e os códigos do JIRA com todas as modificações ou implementações realizadas para que pudesse ser testado

em *hardware*. A equipe de testes utilizava as informações do *software JIRA* que foram preenchidas no início do processo de desenvolvimento.

Grande parte desse processo era realizado de forma manual, fazendo com que o desenvolvedor investisse tempo com processos que poderiam ser automatizados, e permitindo que erros humanos acontecessem como, por exemplo, não executar algum teste, enviar *email* com a lista de *issues* do JIRA incompleta ou com informações equivocadas, não realizar a publicação da imagem de *firmware* ou até mesmo demorar para notificar a equipe de testes.

Analisando as etapas deste processo e o tempo gasto com processos improdutivos e visando diminuir os erros humanos, pois todo erro gera retrabalho, optou-se por automatiar algumas etapas.

4.1.2 Processo automatizado de desenvolvimento de *software* embarcado - GE

Visando diminuir o tempo de entrega de novas *features* e aumentar qualidade e eficiência dos processos, optou-se por automatizar algumas etapas e modificar outras. Para isso, utilizou-se a ferramenta *Jenkins*, que permite criar um *pipeline* para integração contínua e entrega contínua, e a ferramenta *Docker*, para unificar o ambiente de desenvolvimento. A Figura 9 descreve o processo após a automatização, em amarelo está o pipeline CI/CD, na qual é a parte automatizada, todas as demais tarefas são manuais.

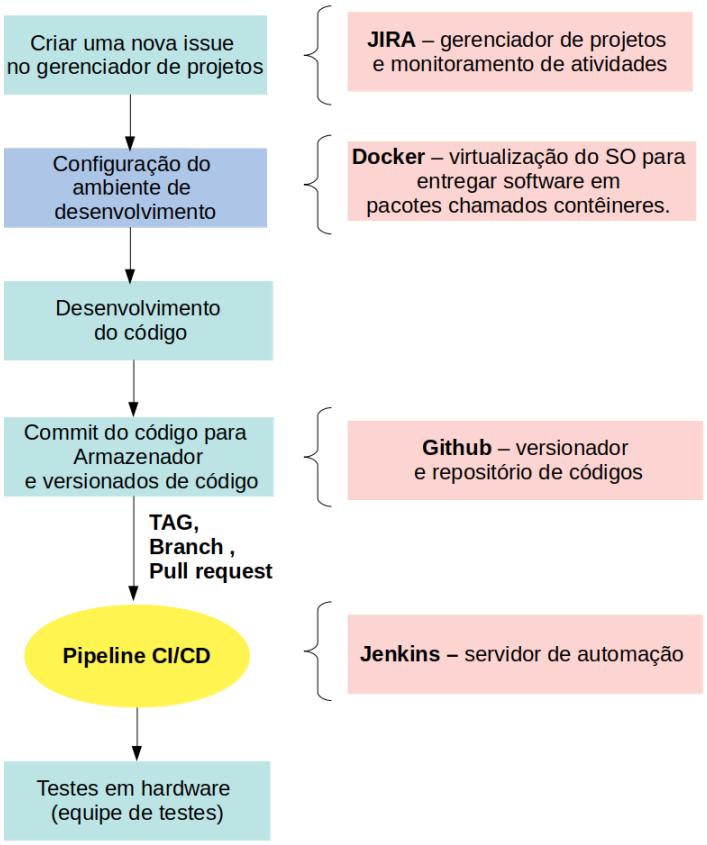
O primeiro passo visando melhorar este processo foi a utilização da ferramenta *Docker* na etapa de “Configuração do ambiente de desenvolvimento”, com o intuito de padronizar o ambiente de desenvolvimento e o de testes.

Para utilizar a ferramenta *Docker* foi necessário desenvolver um *Dockerfile* que descreveu tudo o que seria preciso para executar a aplicação, como as dependências, o sistema operacional que usaria como base e os comandos que serão executados.

Com o *Dockerfile* concluído, foi gerada uma imagem que continha o que estava descrito no *Dockerfile* e após isso, foi gerado um *container*. Esse *container* continha as dependências para compilar, desenvolver e testar o código. Dessa forma, quando um novo desenvolvedor iniciasse na equipe, não seria necessário desperdiçar tempo configurando o ambiente de desenvolvimento para compilar este código, bastaria executar o *Dockerfile* e gerar o *container*. Isso facilitaria, também, para a equipe de testes executar esta aplicação.

Com o ambiente de desenvolvimento configurado, iniciou-se o desenvolvimento de uma nova funcionalidade. Após isso, foi feito o *commit* desta funcionalidade para a plataforma *GitHub*, usada como repositório e versionador de código.

Figura 9 – Processo de desenvolvimento de *software* embarcado automatizado - GE



Fonte: Elaboração própria (2020).

Na etapa do *pipeline* CI/CD utilizou-se o *software* Jenkins, que possui alguns *plugins*, um deles é para integração com o *GitHub*. No Jenkins foi necessário criar um *Job*, que é um projeto, em cujas configurações foram cadastradas as credenciais para acessar o *GitHub* e a organização que seria utilizada neste *Job*, como mostra os campos da Figura 10.

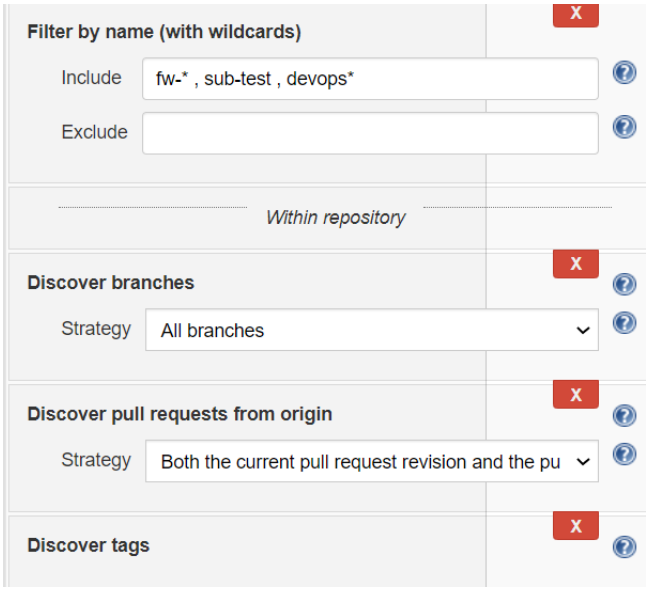
Figura 10 – Configurações do *Job* - credenciais do *GitHub*

Fonte: Elaboração própria (2020).

Na configuração do *Job* era possível também alterar os *triggers* para o

pipeline, como mostra a Figura 11. Em que é possível configurar o Jenkins para executar o *pipeline* quando uma determinada ação é realizada. Neste estudo de caso, optou-se por executar o *pipeline* sempre que uma nova *branch* e TAG fossem criadas, e quando um *Pull-Request* fosse aberto.

Figura 11 – Configurações do *Job -triggers* do pipeline



Fonte: Elaboração própria (2020).

Na Figura 11 no primeiro campo é possível filtrar os repositórios utilizando *wildcards*, que são funções coringas para filtrar arquivos; no segundo campo é possível configurar o jenkins para descobrir todas as branchs, ou apenas uma em específico; e no ultimo campo trata-se da descoberta de novos *pull-requests*.

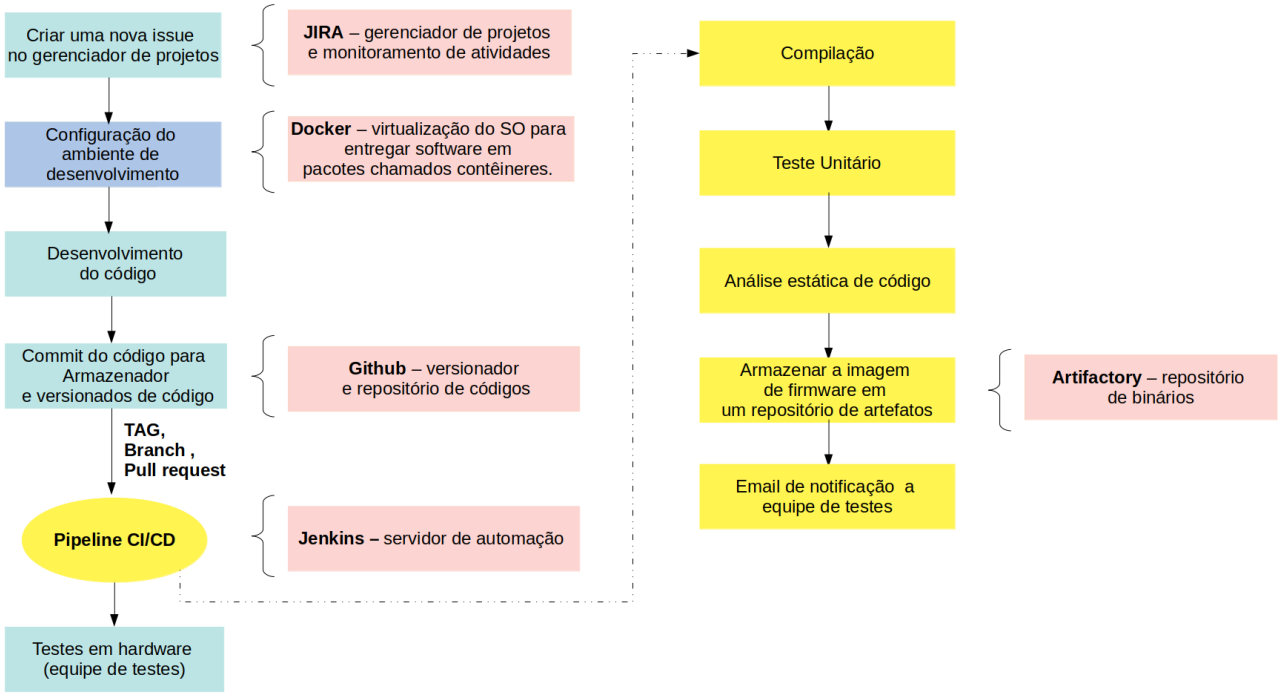
Uma organização do *GitHub* é um conjunto de repositórios, sendo assim, quando algum dos repositórios dentro da organização possuir um arquivo com nome de "Jenkinsfile" na raiz do projeto, este repositório aparecerá na aba "*Repositories*", como mostra a Figura 12. Os nomes dos repositórios e suas descrições estão propositalmente desfocadas por motivos de sigilo industrial.

Figura 12 – Interface do *software Jenkins*

Fonte: Elaboração própria (2020).

Com isso, foi necessário desenvolver o código do *Jenkinsfile* para que ele realizasse todo o processo da Figura 13 mostrado em amarelo.

Figura 13 – Processo com *pipeline* de automatização de processos

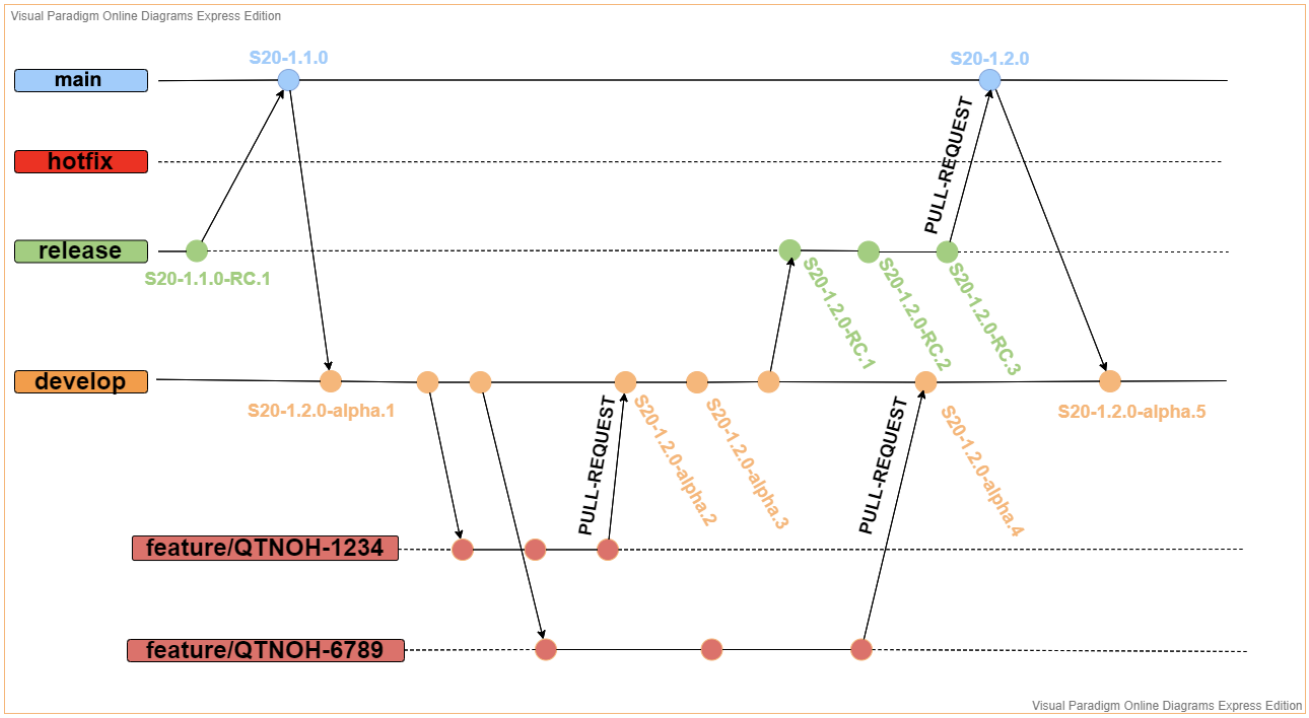


Fonte: Elaboração própria (2020).

O *Jenkinsfile* é desenvolvido usando a linguagem *Groovy*; no *pipeline*, o código é validado por estágios (*stages*) e, cada estágio possui uma função. E neste estudo de caso estão presentes os estágios de compilação, testes unitários, análise estática de código e o estágio para publicar a imagem de *firmware* com notificação por e-mail.

Como repositório e versionador de código, utilizou-se o *GitHub* com a estratégia de *branch GitFlow* (ATLASSIAN, 2020b), que pode ser observado na Figura 14.

Figura 14 – Estratégia de branch *Gitflow*



Fonte: Elaboração própria (2020).

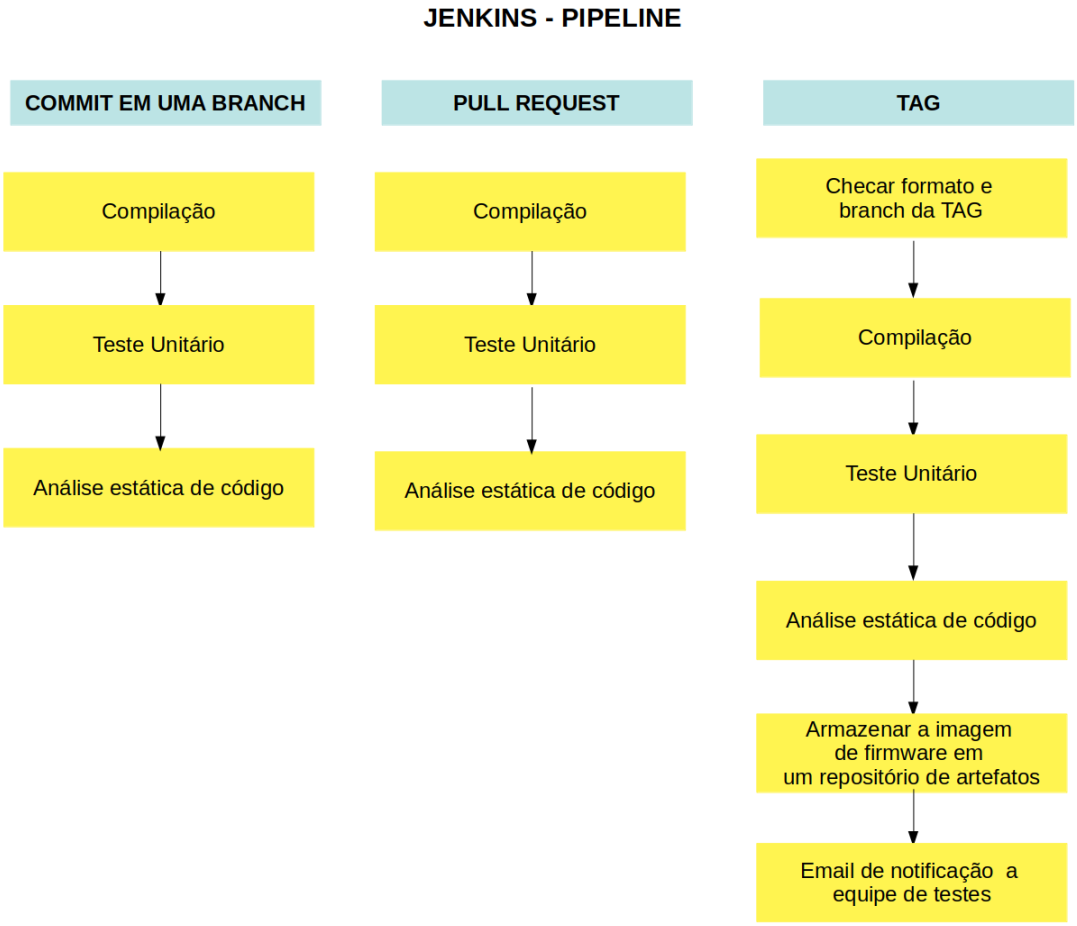
Nessa estratégia, sempre que um novo trecho de código era desenvolvido o desenvolvedor criava uma *branch* chamada *"feature/nome-da-feature"*, indicada em vermelho, a partir da branch *develop*. Após isso, um *pull request* era aberto. Caso o *pipeline* do Jenkins executesse sem nenhum erro, o desenvolvedor possui a permissão para dar *merge* para a *branch develop*.

Ao final do desenvolvimento de uma nova versão alfa na *branch develop*, uma TAG era criada; se fosse uma RC (*release candidate*) a TAG é criada na *branch* de *release*; caso fosse uma versão oficial, era realizado o *merge* para a *branch main* e criada a TAG nesta *branch main*.

Sempre que uma TAG era criada, publicava-se a imagem de *firmware* no *JFrog Artifactory*, pois a TAG no *GitHub* era utilizada para marcar um código; neste estudo de caso, utilizou-se a TAG como uma sinalização de uma nova versão, ou seja, sempre que ocorria a criação de uma TAG, publicava-se no *Jfrog Artifactory* essa nova versão.

A Figura 15 ilustra os *triggers* do *pipeline* do Jenkins, os quais eram executados sempre que os eventos "*commit* em uma *branch*", "*pull-request*" e TAG aconteciam.

Figura 15 – *Trigger pipeline* Jenkins

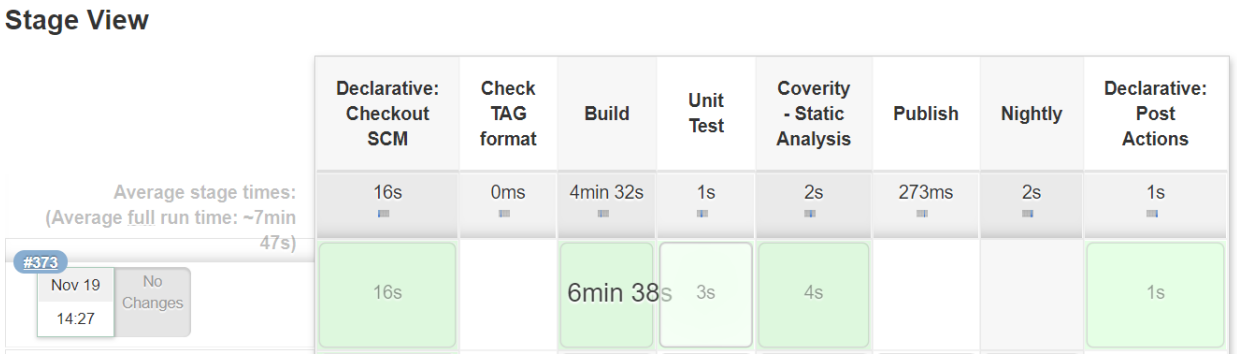


Fonte: Elaboração própria (2020).

Observando a Figura 15 nota-se que quando ocorria um novo *commit* ou um *pull-request* na plataforma do *GitHub*, ocorria somente a execução dos seguintes *stages*: compilação do código, testes e a análise estática de código - diferente de quando possuía uma *TAG* que era executados um pipelie completo com mais *stages* como: checar TAG, compilação, testes unitários, análise estática de código, publicação da imagem de *firmware*. A Figura 16 ilustra o *pipeline* do Jenkins quando o *trigger* "novo *commit* em uma *branch*" ou *pull-request* era acionado, observa-se que somente é executado os estágios de *build* (compilação), *unit test* (testes unitátios) e *static analysis* (analise estática).

Caso algum desses estágios possuísse erros, o *pipeline* era parado imediatamente. No *GitHub* era exibido uma notificação de que havia sido encontrado um erro no pipeline e que este código não pode ser acrescentado a *branch develop* ou *master*.

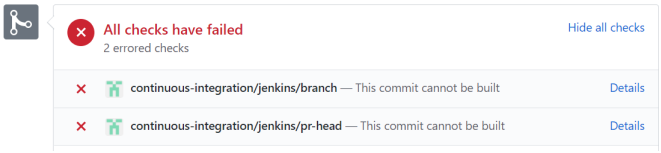
Figura 16 – Somente os stages de compilação e testes sendo executados



Fonte: Elaboração própria (2020).

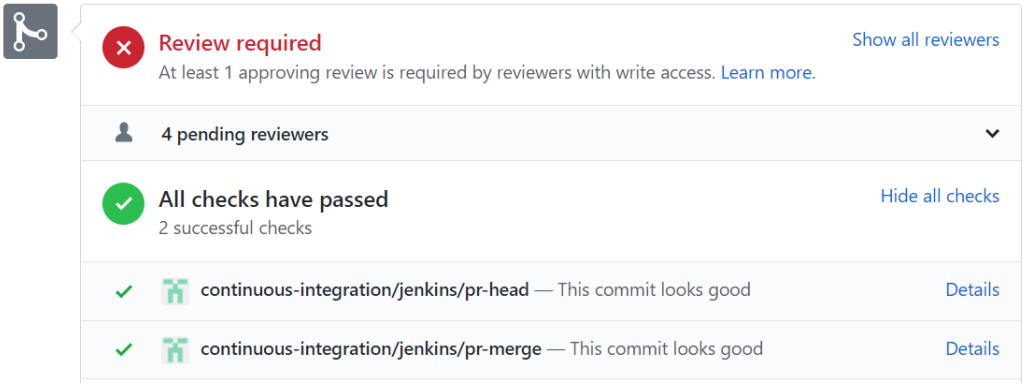
Na Figura 17 é possível observar a aba de *Pull-request* do *GitHub* quando o *pipeline* do Jenkins possui erros de compilação. Já a Figura 18, ilustra quando o *pipeline* foi executado sem erros permitindo um eventual *merge*. Nota-se que a mensagem apresentada diz que todos os estágios passaram com sucesso.

Figura 17 – *GitHub* Pull request quando possui erros no *pipeline* do jenkins



Fonte: Elaboração própria (2020).

Figura 18 – *GitHub* Pull request quando não possui erros no *pipeline* do jenkins



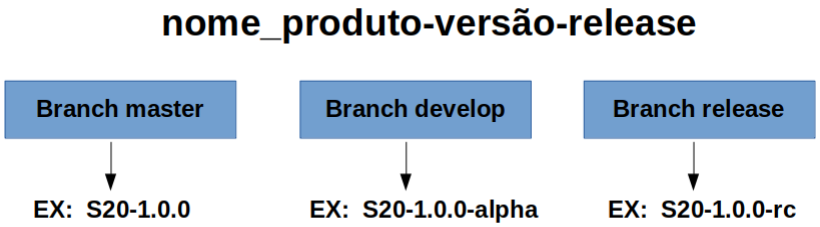
Fonte: Elaboração própria (2020).

Para suprir todas as necessidades deste projeto, algumas funções genéricas foram desenvolvidas em uma *shared library* em *groovy* e foram adicionadas ao Jenkins.

Utilizando a abordagem de *shared library* foi possível reutilizar essas funções em outros repositórios e *pipelines*. Como o objetivo era que todos os produtos desenvolvidos tivessem um *pipeline* de integração contínua, optou-se por fazer esta biblioteca genérica para ser reaproveitada em outros projetos.

Na *shared library* foi desenvolvida uma função para analisar se a TAG estava no formato certo, pois quando uma TAG era criada, ela precisava cumprir o formato mostrado na Figura 19.

Figura 19 – Formato das TAGS

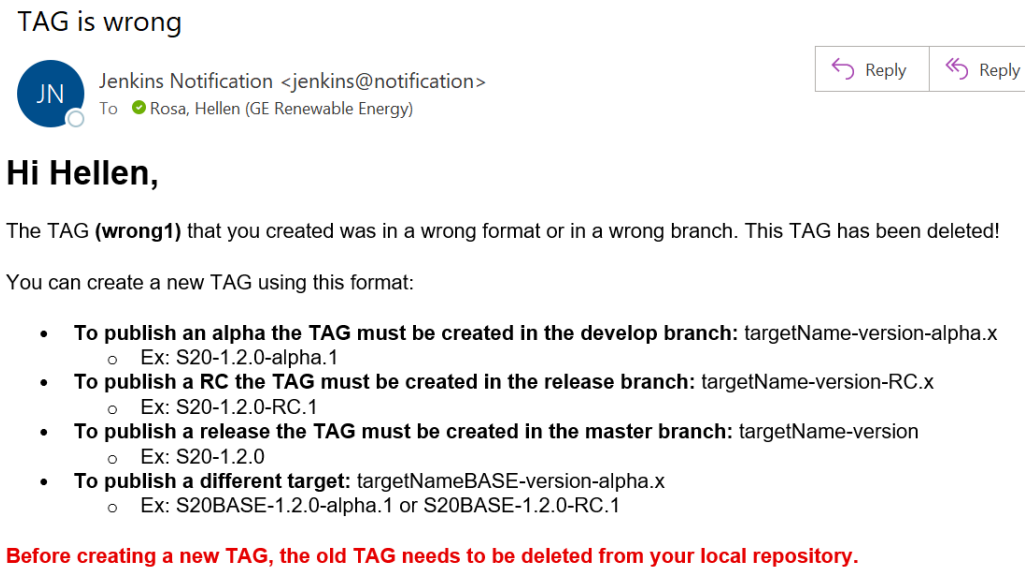


Fonte: Elaboração própria (2020).

Como a criação das TAGs no *GitHub* foi feita de forma manual pelo desenvolvedor, criou-se esta função para evitar erros humanos.

Sempre que uma TAG fosse criada no formato errado ou em uma *branch* errada, a *shared library* possuía uma função que deletava esta TAG e enviava um *email* diretamente para o desenvolvedor que criou a TAG errada, como mostra a Figura 20.

Figura 20 – Email enviado quando a TAG está errada



Fonte: Elaboração própria (2020).

Após analisar se a TAG estava no formato correto e na *branch* correta, era feita a compilação do código. Não contendo nenhum erro, seguia para a etapa de testes unitários que eram desenvolvidos utilizando o *Google Tests*; após isso, eram feita a análise estática de código utilizando o *software Coverity*.

Caso todas as etapas anteriores tivessem sido executadas sem erros, era feita a publicação da imagem de *firmware* no *JFrog Artifactory*. Com as informações obtidas a partir da TAG, o *jenkins* infere em qual pasta deveria ser feita a publicação; se for uma nova *release*, seria publicado na pasta *release* daquele produto, garantindo a organização e padronização no repositório de binários.

Após a publicação, era automaticamente enviado um *e-mail* para a equipe de testes. Este *email* continha as seguintes informações sobre a nova imagem de *firmware*:

a) *Link* para baixar a imagem;

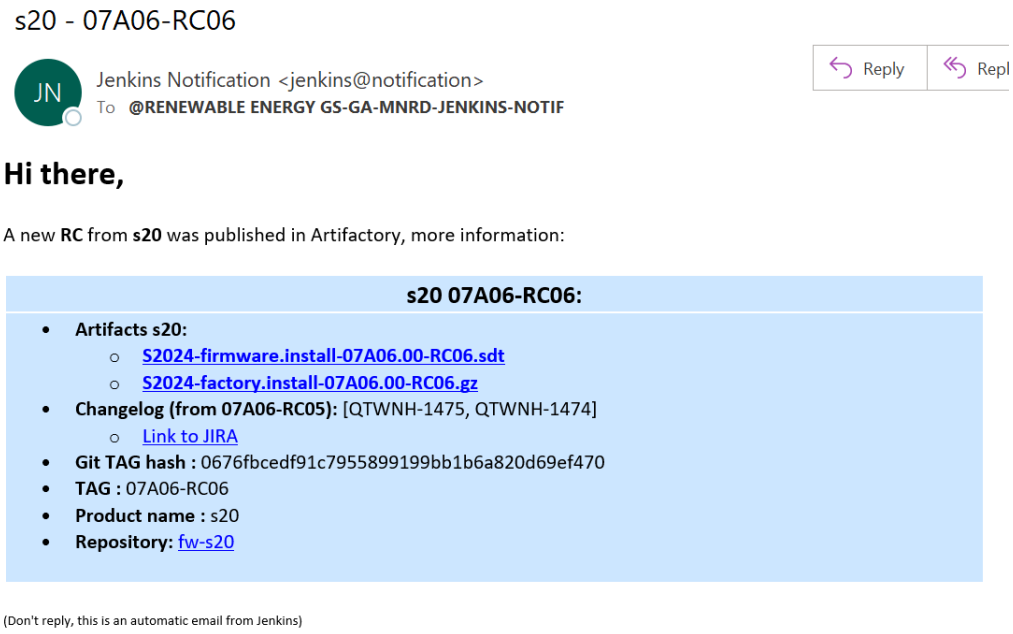
b) *Link* para o JIRA com todas as *issues e tasks* implementadas nessa nova versão, sendo possível saber todas as funções desenvolvidas;

c) *Link* para o repositório do projeto no Github;

d) *Commit hash* do *GitHub* e nome da TAG que foi gerada essa nova versão.

Esta funcionalidade de envio de *email* foi desenvolvida em *html* na *shared library*, para ser facilmente reaproveitada em outros projetos. Para obter as informações contidas neste *email* foram desenvolvidas funções em *groovy* também na *shared library*. A Figura 21 ilustra o modelo de *email* desenvolvido, que era enviado para o time quando ocorria uma nova publicação da imagem de *firmware*.

Figura 21 – Email enviado quando um arquivo é publicado



Fonte: Elaboração própria (2020).

As informações que contidas no *email* eram extremamente necessárias para a equipe de testes saber o que deveria ser testado; por esse motivo, foi automatizado o processo de obter as informações e fazer o envio do *email*, evitando falhas humanas e desperdício de tempo do desenvolvedor.

Após implementar e colocar em prática o *Jenkinsfile*, a *shared library* com as funções e a utilização do *Docker*, tornou-se possível analisar os resultados obtidos e os benefícios da automatização de processos para este estudo de caso. Na Seção 4.2 serão detalhado todos os resultados obtidos com esse processo.

4.2 Resultados obtidos com a automatização de processos

Nesta seção serão abordados os resultados obtidos com a utilização do *Docker* e da automatização de processos com o *Software Jenkins*.

4.2.1 Resultados obtidos com a utilização do *Docker*

Com a utilização do *Docker Container* observaram-se os seguintes benefícios para este estudo de caso:

a) **Mais leve do que uma máquina virtual** - ele compartilha o sistema operacional, com isso o *docker* consegue ler as bibliotecas e binários do sistema operacional hospedeiro, tornando menor do que uma maquina virtual.

b) **Ambientes semelhantes** - um ambiente que pode ser utilizado no computador dos desenvolvedores e na equipe de testes, sem faltar pacotes ou dependências.

c) **Facilmente portável** - geralmente a imagem do *Container* é armazenado em um repositório de imagens *Docker*, permitindo que seja baixado e utilizado em qualquer computador.

d) **Empacotamento de aplicações e dependências** - por utilizar a estrutura de camadas, pode ser facilmente alterado caso novos pacotes precisem ser instalados.

A utilização do *Docker* trouxe a padronização do ambiente da equipe de desenvolvimento e de testes e validações, tornando possível também a utilização da abordagem de micro-serviços, que é quando a aplicação é desmembrada em pequenos serviços.

Anteriormente sem a utilização do *Docker*, ocorriam diversos problemas na hora de compilar o código de um produto, principalmente quando era um produto desenvolvido muito tempo atrás. Como cada desenvolvedor tinha suas preferências de ferramentas, editores de código e sistema operacional, acabava ocorrendo divergências entre as versões de bibliotecas nativas dos sistema operacional, ou até mesmo a falta delas. Por isso, era trabalhoso até descobrir as dependências daquele determinado produto e conseguir compilá-lo.

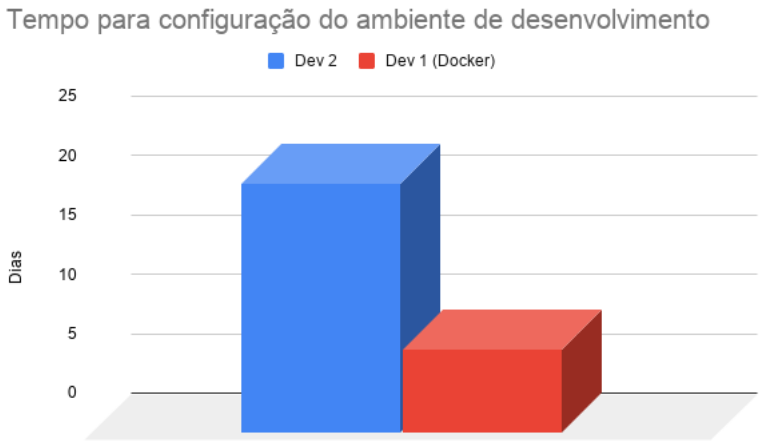
Por isso, um impacto significativo da utilização do *Docker* foi a facilidade de receber novos integrantes na equipe de desenvolvimento. Com o *Docker* tornou-se mais rápida a configuração do ambiente de desenvolvimento, pois o desenvolvedor possuía a aplicação empacotada com suas dependências.

A Figura 22 ilustra dois casos de novos desenvolvedores, o Dev 2 iniciou na GE antes da utilização do *Docker container* e levou em média 21 dias para configurar todo ambiente de desenvolvimento, instalar as ferramentas necessárias de trabalho, descobrir e instalar os pacotes e dependências para compilar o código do produto, para então iniciar o desenvolvimento de uma nova funcionalidade. Um problema muito recorrente neste processo era a versão das dependências e pacotes, que se diferentes, apresentavam incompatibilidades para compilar o código.

O Dev 1, iniciou na GE quando o *Docker container* já estava sendo utilizado, levou em média 7 dias para configurar o computador, instalar as ferramentas necessárias, configurar o ambiente de desenvolvimento, compilar o produto e começar a desenvolver uma nova aplicação.

Esta comparação de tempo ocorreu desde o momento em que o desenvolvedor recebeu o computador da empresa até o momento em que ele começou a de fato desenvolver uma nova funcionalidade ou aplicação.

Figura 22 – Tempo para configuração do ambiente de desenvolvimento



Fonte: Elaboração própria (2020).

O resultado apresentado por meio da Figura 22 é derivado do tempo de compreensão do encadeamento de bibliotecas do *firmware* uma vez que cada *firmware* possuía um elevado número de bibliotecas internas e externas. Obter essas bibliotecas internas não era uma tarefa trivial, além de que muitas bibliotecas eram dependentes do sistema operacional utilizado.

Por fim, observa-se um ganho significativo de aproximadamente 14 dias para cada desenvolvedor que inicia na equipe, aumentando a produtividade e motivação dos mesmos.

4.2.2 Automatização dos processos utilizando o *Software Jenkins*

Utilizando o *Software Jenkins* e a *shared library* desenvolvida, obtiveram-se muitos benefícios: economia de tempo, qualidade de código, confiabilidade dos dados e melhora na comunicação entre as equipes.

Na Subseção 4.2.2.1 será abordado sobre os testes automatizados e seus benefícios e na Subseção 4.2.2.2 os benefícios da publicação automática.

4.2.2.1 Testes automatizados

Aplicando a automatização dos processos, obteve-se uma melhora na qualidade do código, visto que o *pipeline* possuía os seguintes estágio, compilação, testes unitários, testes de análise estática de código, publicação e envio de *email*. Usando o *pipeline* sempre que era introduzido um novo código na *branch master*, esse código passou por uma compilação em um ambiente limpo, por testes unitários e análise estática de código.

O teste de análise estática de código é longo, por esse motivo anteriormente era feito somente ao fim de uma nova versão, levando em média 1 mês somente para correções dos erros provenientes deste teste. Assim, optou-se por incluir o teste de análise estática ao *pipeline* do *Jenkins*; dessa forma, todo código adicionado ao *Github* era devidamente testado e só poderia ir para a *branch master* caso não tivesse erros para ser corrigido. Caso o teste de análise estática resultasse em algum erro, o mesmo deveria ser corrigido, pois o *Pull-request* era bloqueado se existisse algum erro no *pipeline* do *Jenkins*. Desta forma, a análise estática era executada em partes pequenas de código, resultando em um tempo menor de execução deste teste.

Observou-se que era mais produtivo executar os testes de análise estática em pequenos códigos incluídos no *Github* ao invés de executá-los somente ao final de uma nova versão. Para o desenvolvedor, era mais simples corrigir um erro de um código que já estava sendo trabalhado, ao invés de precisar rever o código ao final da versão para corrigir os erros. Observou-se uma melhora na qualidade do código e a diminuição de erros nas etapas finais do projeto.

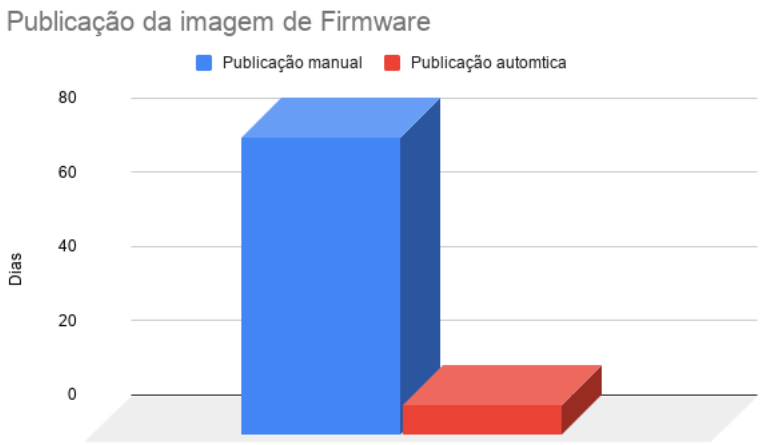
4.2.2.2 Publicação automática

Utilizando a abordagem da publicação automática a partir de uma TAG no *GitHub* obteve-se um ganho significativo de tempo.

Na GE em um ano, trabalha-se com uma média de 8 projetos, cada projeto possui uma média de 10 publicações de novas versões, cada publicação leva 1 dia de trabalho do desenvolvedor, pois é necessário limpar o ambiente, configurar o ambiente, compilar o código, montar um *changelog* que especifica o que foi modificado e implementado nessa nova versão, fazer a publicação no *Artifactory* e mandar o *email* para o time com essas informações; tudo de forma manual.

Observa-se, na Figura 23, o tempo utilizado para publicações em um ano de trabalho, comparando a publicação manual, que consome em média 80 dias no ano, com a publicação automática, que investe uma média de 8 dias no ano. Ambas as médias estão contabilizando imprevistos que podem ocorrer ao longo do processo.

Figura 23 – Tempo para publicação da imagem de *Firmware* (um ano)



Fonte: Elaboração própria (2020).

Utilizando o *pipeline* do *Jenkins* para publicação automática, obteve-se uma economia do tempo do desenvolvedor de 72 dias no ano.

Além da economia de tempo, deve-se levar em conta a confiabilidade da publicação, a consistência dos dados do *changelog* que por ser automatizado, evita falhas humanas e a comunicação assertiva entre as equipes de desenvolvimento e testes. Do ponto de vista da equipe de *firmware*, este foi um grande avanço.

5 CONSIDERAÇÕES FINAIS

O presente trabalho abordou um estudo de caso dos processos de desenvolvimento de um *software* embarcado desenvolvido pela GE (*General Electric*), com isso analisou-se o processo tradicional de desenvolvimento, realizou-se a automatização deste processo e posteriormente foi feita a análise dos resultados obtidos.

Com o objetivo de acelerar o *time-to-market*, melhorar qualidade, evitar trabalhos manuais e melhorar a comunicação entre equipes aplicou-se a automatização de processos que utilizou a ferramenta *Jenkins*, onde foi desenvolvido um *pipeline* de automatização. Com essa ferramenta foi possível automatizar a compilação, testes e publicação de uma nova versão. Como resultado da publicação automática obteve-se uma economia de 72 dias de trabalho de um desenvolvedor por ano e a diminuição dos erros de código nas etapas finais do projeto, pois cada nova parte de código inserida, percorria uma série de testes resultando em uma melhora na qualidade de código.

Em conjunto, utilizou-se o *Docker* para empacotar a aplicação e padronizar os ambientes de desenvolvimento e de testes e validações, contribuindo para a chegada de novos integrantes na equipe e facilitando a compilação do produto no time de testes e validações. Utilizando esta ferramenta foi possível obter um ganho de 14 dias para cada novo desenvolvedor que inicia na equipe.

Este estudo de caso alcançou seus objetivos de melhorar a fluidez do processo de desenvolvimento deste *software* embarcado. Ofereceu-se aos desenvolvedores a oportunidade de focar nas aplicações e não em processos manuais e também facilitou a comunicação entre times, visto que foi desenvolvido um envio de *e-mail* automático.

Pelos motivos citados acima, com este trabalho obteve-se um resultado satisfatório. Foi desenvolvido com sucesso apesar das dificuldades encontradas por consequência da pandemia (COVID19). Neste período, a empresa estava fechada e não foi possível fazer os testes em *hardware*, que também estavam no escopo inicial deste trabalho.

Para futuros trabalhos, sugere-se as seguintes oportunidades de melhoria:

a) instalar novas versões e atualizações no *hardware* de forma automática;
b) automatizar o monitoramento de consumo de CPU e memória;
c) implantar a automatização de processos em todos os demais produtos desenvolvidos pela empresa.

REFERÊNCIAS

AGILE MANIFEST. *Agile Manifest - principles*. 2021. Disponível em: <https://agilemanifesto.org/iso/ptbr/principles.html>. Acesso em: 07 mar 2021. Citado na página 19.

ARBULU, R.; BALLARD, G.; HARPER, N. Kanban in construction. *Proceedings of IGLC-11, Virginia Tech, Blacksburgh, Virginia, USA*, p. 16–17, 2003. Citado na página 20.

ATLASSIAN. *Ferramentas ágeis para equipes de software*. 2020. Disponível em: <https://www.atlassian.com/br/software/jira/agile#scrum>. Acesso em: 29 dez 2020. Citado 4 vezes nas páginas 12, 21, 22 e 23.

ATLASSIAN. *Fluxo de trabalho de Gitflow*. 2020. Disponível em: <https://www.atlassian.com/br/git/tutorials/comparing-workflows/gitflow-workflow>. Acesso em: 23 abr 2021. Citado na página 38.

COHN, M. *Desenvolvimento de software com Scrum: aplicando métodos ágeis com sucesso*. [S.l.]: Bookman, 2000. Citado na página 12.

DOCKER. *Why Docker*. 2021. Disponível em: <https://www.docker.com/why-docker>. Acesso em: 28 fev 2021. Citado 2 vezes nas páginas 12 e 25.

FUGGETTA, A. Software process: A roadmap. 22nd int. In: *Conf. on Software Engineering (ICSE'2000), Future of Software Engineering Track, June*. [S.l.: s.n.], 2000. p. 4–11. Citado na página 12.

GE GRID SOLUTIONS. *Reason S20*. 2020. Disponível em: <https://www.gegridsolutions.com/communications/catalog/s20.htm>. Acesso em: 23 abr 2021. Citado na página 31.

GIL, A. C. *Como elaborar projetos de pesquisa*. 4. ed. [S.l.]: Editora Atlas S.A., 2002. ISBN 9788522478408,8522478406. Citado na página 29.

GITHUB. *GitHub Guides*. 2021. Disponível em: <https://guides.github.com/activities/hello-world/#what>. Acesso em: 24 fev 2021. Citado 2 vezes nas páginas 17 e 18.

GOMES, A. F. *Agile: Desenvolvimento de software com entregas frequentes e foco no valor de negócio*. [S.l.]: Editora Casa do Código, 2014. Citado 3 vezes nas páginas 19, 20 e 21.

HUMBLE, J.; MOLESKY, J. Why enterprises must adopt devops to enable continuous delivery. *Cutter IT Journal*, v. 24, n. 8, p. 6, 2011. Citado na página 14.

JENKINS. *Jenkins User Documentation*. 2021. Disponível em: <https://www.jenkins.io/doc/>. Acesso em: 31 jan 2021. Citado 2 vezes nas páginas 12 e 25.

KIM, G. et al. *Manual de Devops*. [S.l.]: Como obter agilidade, confiabilidade e segurança em organizações . . . , 2018. Citado 2 vezes nas páginas 12 e 13.

LWAKATARE, L. E. et al. Towards devops in the embedded systems domain: Why is it so hard? In: *2016 49th Hawaii International Conference on System Sciences (HICSS)*. [S.l.: s.n.], 2016. p. 5437–5446. Citado 2 vezes nas páginas 12 e 15.

MALHOTRA, N. *Pesquisa de marketing: uma orientação aplicada*. 6. ed. [S.l.]: Bookman, 2010. ISBN 978-85-407-0062-8. Citado na página 29.

MORO, R. D. *Avaliação e Melhoria de Processos de Software: Conceituação e Definição de um Processo para Apoiar a sua Automatização*. Tese (Doutorado) — Dissertação de Mestrado, Departamento de Informática, Centro Tecnológico . . . , 2008. Citado na página 12.

NETO, A.; DIAS, C. Introdução a teste de software. *Engenharia de Software Magazine*, v. 1, p. 22, 2007. Citado na página 27.

Seth, N.; Khare, R. Aci (automated continuous integration) using jenkins: Key for successful embedded software development. In: *2015 2nd International Conference on Recent Advances in Engineering Computational Sciences (RAECS)*. [S.l.: s.n.], 2015. p. 1–6. Citado 2 vezes nas páginas 23 e 24.

SHAH, J. A.; DUBARIA, D. Netdevops: A new era towards networking devops. In: *2019 IEEE 10th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*. [S.l.: s.n.], 2019. p. 0775–0779. Citado na página 16.

TERRA, R.; BIGONHA, R. S. Ferramentas para análise estática de códigos java. *Monografia, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte*, 2008. Citado na página 28.

WIJAYA, P. E.; ROSYADI, I.; TARYANA, A. An attempt to adopt devops on embedded system development: empirical evidence. In: IOP PUBLISHING. *Journal of Physics: Conference Series*. [S.l.], 2019. v. 1367, n. 1, p. 012078. Citado na página 16.