DESENVOLVIMENTO DE *PIPELINE PARA AUTOMATIZAR A* INTEGRAÇÃO E ENTREGA CONTÍNUAS PARA ENGENHARIA DE *SOFTWARE*

NETO, Paulo Renato[1]*; AZEVEDO, Tiago[1]; LOPES, Jeferson[1]; PINTANEL, Alice[1]; BORGES, Eduardo N.[1]; BERRI, Rafael A.[1]; ROSA, Vagner S.[1].

[1]Centro de Ciências Computacionais / Universidade Federal do Rio Grande - FURG

*seven.renato@furg.br

Resumo: A crescente automatização e integração de códigos-fonte de alta complexidade é de extrema importância para as aplicações modernas, uma vez que exigem abordagens eficientes para testar e implantar diversos sistemas em produção de forma rápida e com alta confiabilidade. Dessa forma, a Integração e Entrega Contínua (CI/CD) consolida-se como um conjunto de práticas e ferramentas essenciais a serem abordadas para automatizar o ciclo do Desenvolvimento de *Software*, aumentando a qualidade do produto final, tolerância a falhas e a produtividade das equipes envolvidas no processo. Muitas vezes a manutenção, sincronização e versionamento contínuo de repositórios GitHub para com códigos presente no servidor, tende a ser um processo extremamente manual com grande margem a existências de falhas humanas, o que pode colaborar com tempos demasiados de espera para inserção de novas atualizações, manutenções e demonstrações. Neste contexto, este artigo tem como objetivo principal apresentar uma forma de desenvolvimento da implementação de um *pipeline* CI/CD para a atualização automatizada de um sistema robusto para entrega em produção,

utilizando GitHub *Actions* para orquestrar o processo de geração de imagens *Docker* em GitHub *Container Registry*, usando *Jenkins* para um melhor monitoramento e automatização da implantação em ambiente de produção através da definição de sequências de passos a serem resolvidos até a atualização ser realizada de um arquivo *Docker Compose*. A solução definida demonstra como a integração entre ferramentas populares e a utilização de containers *Docker* podem simplificar o processo de entrega de *software*, garantindo maior agilidade e confiabilidade na implantação de novas versões do produto.

Palavras-chave: Engenharia de *Software*; Docker; CI/CD; DevOps; Jenkins.

1 INTRODUÇÃO

A Engenharia de *Software* tem desempenhado um papel fundamental na formação de aplicações de alta qualidade nas últimas décadas. Segundo Aggarwal (2005), Engenharia de *Software* consiste na capacidade de produzir um código de boa qualidade e de fácil manutenção dentro do custo e tempo estipulado. Desse modo, é possível perceber, que necessita-se de abordagens cada vez mais eficazes e precisas para aumentar a sua eficiência e produtividade de desenvolvimento de software, diminuindo a necessidade de processos manuais e aumentando a rapidez na entrega dos serviços.

Neste contexto, as ferramentas que facilitam esse processo, se tratam de ferramentas de DevOps, que cuidam da Integração Contínua e Entrega Contínua (CI/CD) das aplicações, facilitando e aumentando assim a produtividade do desenvolvimento e a definição de manutenções nos sistemas. Contudo, o processo de configuração dessas ferramentas tende a ser um desenvolvimento complexo e enigmático de aprender (Shahin, et al., 2017), onde comumente a equipe deve possuir alguém especializado na área. Diante dessa realidade, este trabalho visa contribuir teoricamente para a área, ajudando a desmistificar este processo, demonstrando os ganhos de produtividade proporcionados pela automação de *pipelines* de CI/CD e como essa otimização impacta positivamente o desempenho da equipe de desenvolvimento (Makani, 2022).

Logo como base fundamental para este trabalho, tem-se como objetivo apresentar a implementação de um *pipeline* de CI/CD automatizado para *deploy* de aplicações. O estudo foi realizado em um contexto convencional de aplicações, uma *Front-End* e outra *Back-End*, onde foram utilizadas ferramentas que buscam facilitar o desenvolvimento através do aumento da disponibilidade e automatização do processo de deploy automático. A solução definida utiliza GitHub Actions para orquestrar o processo de *build* e *push* de imagens *Docker*, e Jenkins junto a *Docker Compose* para automatizar o *deploy* em ambiente de produção, integrando ambas as ferramentas via *webhooks*.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta os principais conceitos que sustentam o entendimento necessário para implementação do *pipeline* de CI/CD para as aplicações, explorando as tecnologias e práticas que buscam guiar a automação do processo de desenvolvimento e entrega de *software*.

2.1 DevOps

O presente estudo aprofundou-se na área de DevOps como alicerce para a construção da solução implementada do *pipeline* CI/CD. Segundo a literatura (Zhu et al., 2016), DevOps é definido como um conjunto de práticas ágeis que buscam implementar automatização ponta a ponta na entrega e no desenvolvimento de *software,* onde existe um intuito principal de garantir qualidade na entrega do produto. Desse modo, visa-se integrar equipes de desenvolvimento (Dev) e operações (Ops) para otimizar o ciclo de vida do *software*, promovendo a colaboração através da automação e a comunicação eficiente entre todas as áreas (Kim et al., 2013).

2.2 Integração Contínua e Entrega Contínua (CI/CD)

A Integração Contínua (CI) e a Entrega Contínua (CD) são práticas essenciais em DevOps, as quais visam automatizar e otimizar o processo de entrega da aplicação. Como definido por Pinheiro (2022), "CI/CD se trata de um método de produção de *software* onde em uma organização os desenvolvedores entregam frequentemente alterações de aplicações aos clientes com a inclusão de ferramentas de automação no processo de desenvolvimento". Desse modo, percebe-se então que o objetivo da CI/CD é fornecer automação e monitoramento contínuos em todas as fases do ciclo de vida de uma aplicação, desde a fase de integração e testes até a entrega e implantação.

Dessa forma, pode-se conceituar a Integração Contínua (CI) como mudanças para aumento da produtividade da equipe de *software* que são feitas a uma aplicação em um curto período de tempo, automaticamente confirmadas, testadas antes de serem implementadas no ambiente de produção. Elas definem agilidade ao não exigir dos desenvolvedores a necessidade da execução manual de testes, ou criação de imagens Docker (Shahin, et al., 2017).

De outro modo, Entrega Contínua (CD) é a capacidade de implementar de forma contínua todos os tipos de alterações em *software*, incluindo novos recursos, modificações de configuração e correções de *bugs*, em produção de forma segura, rápida e duradoura (Humble et. al, 2010).

Já a combinação desses dois conceitos permite a criação de um *pipeline* de CI/CD que conecta as equipes de operações e desenvolvimento, fomentando a adoção de uma cultura DevOps de produtividade sendo assim um dos alicerces principais deste presente estudo de implementação.

2.3 *Containers* de *software*

Em síntese, *containers* são unidades de *software* que empacotam o código completo da aplicação e todas suas dependências dentro de uma camada diferente do sistema operacional. Eles utilizam imagens, garantindo portabilidade e consistência entre diferentes ambientes, além da diminuição de erros de compatibilidade ou versionamento em um contexto de diversas aplicações no mesmo ambiente de sistema operacional. As imagens, por sua vez, se tratam de conjuntos de arquivos executáveis usados para criar um *container*, dentro dela existem todas as dependências e arquivos necessários para a execução correta do *container*.

Docker se trata de uma plataforma de código aberto que busca ajudar a simplificar o processo de implantação de *software*, de forma menos propensa a erros (Merkel, 2014), através do desenvolvimento, entrega e execução de aplicações dentro de containers. De outro modo, *Docker Compose* se trata de uma acréscimo ao *Docker*, sendo uma ferramenta que permite a definição e o gerenciamento de aplicações multi-*container* por meio de um arquivo YAML. Dessa forma, seu principal benefício é a capacidade de simplificar a entrega (*deploy*), a qual se trata do processo de disponibilizar uma aplicação em um ambiente específico, seja ele de desenvolvimento, teste, homologação ou produção. Neste contexto, através do uso de *containers*, no *Docker Compose o deploy* pode ser sintetizado a executar a imagem da aplicação em um *container* no ambiente desejado, garantindo que todas as dependências e configurações estejam presentes e funcionando corretamente (Gkatziouras, E., 2022).

2.4 GitHub *Actions*

O GitHub *Actions* (GA) se trata de uma ferramenta que implementa os conceitos fundamentais para a criação de um *pipeline* automatizado de forma moderna e simplificada Ele é uma plataforma de CI/CD nativa do GitHub que permite automatizar fluxos de trabalho (*workflows*) dentro do próprio repositório de código, adicionando grandes possibilidades de monitoramento e disparada de ações, o que permite sua utilização em áreas diversas para o versionamento no contexto atual. (Wessel et al., 2023)

O GitHub *Actions* se baseia em eventos que disparam fluxos de trabalho pré-definidos. Esses fluxos de trabalho são descritos em arquivos do tipo YAML, e podem executar uma série de ações, como compilar código, executar testes, construir imagens *Docker* e realizar *deploy* de aplicações, demonstrando assim a grande contribuição para a implementação de uma solução de automatização do processo de Integração e Entrega Contínua deste trabalho.

Por fim, vale ressaltar que este trabalho, utilizou-se de um ambiente para registrar as imagens Docker e realizar o *deploy* das aplicações, o GitHub Container Registry (GHCR), o qual se trata de um serviço oferecido pelo GitHub para armazenar e gerenciar imagens Docker de forma privada ou pública. A escolha pelo GHCR se

justifica pela sua integração nativa com o GitHub Actions, o que acaba por facilitar a automação do processo e abordagem do *build* e *push* das imagens.

2.5 Jenkins

Jenkins é um *framework* de código aberto, assim como *Docker*, amplamente usado na área de DevOps para a implementação de *pipeline*s de CI/CD. Devido ao seu contexto de ser livre e produzido em Java, é altamente extensível por meio de *plugins*, oferecendo dessa forma grande flexibilidade e integração com diversas ferramentas (Pinheiro, 2022).

No Jenkins, diversos conceitos para automatização são abordados. Entre eles, destacam-se os *jobs*, que representam tarefas específicas dentro de um *pipeline* de CI/CD. Cada *job* pode executar uma série de ações, como compilar código, executar testes, construir imagens *Docker* ou realizar o *deploy* da aplicação.

3 METODOLOGIA

Esta seção detalha a metodologia empregada para a automação do *pipeline* de CI/CD, descrevendo as abordagens iniciais e a inspiração na literatura.

3.1 Abordagens Iniciais

Durante a busca por uma solução para a implementação da ideia substancial deste trabalho, existiu principalmente duas abordagens iniciais as quais fundamentaram o caminho de evolução dos estudos e pesquisas na literatura, sendo essas abordagens manuais, nomeadas como: “Baseada em Arquivos” e “Utilizando GitHub Actions & Docker Compose”, evidenciadas nas subseções a seguir.

3.1.1 Abordagem 1 - Baseada em Arquivos

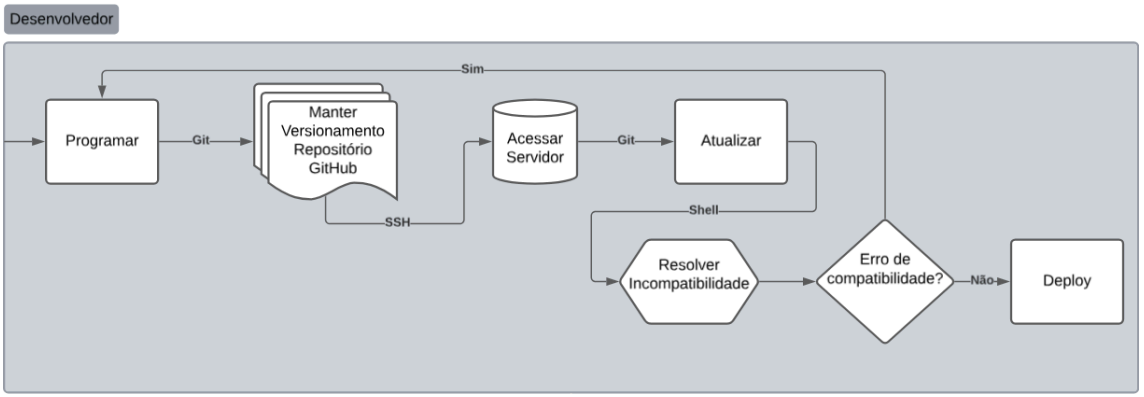

Figura 1: Diagrama Abordagem 1 - Baseada em Arquivos.

Ao início do projeto de implementação da solução, sem as definições gerais das aplicações, a forma pela qual as manutenções estavam estabelecidas era primordialmente pelo código-fonte. A Figura 1 ilustra a abordagem desse processo. O

desenvolvedor, ao gerar uma nova atualização no repositório do GitHub, acessava o servidor via SSH, baixava as novas atualizações dos códigos versionados pelo GitHub via linha de comando e trabalhava sobre o código-fonte da aplicação utilizando *scripts* sem utilização de *Docker*. Não utilizar a ferramenta de orquestração de *containers* contribui com versões diferentes durante toda a aplicação, além de problemas com versão de bibliotecas, dependências e incompatibilidade devido conflitos entre as aplicações que estavam a ser executadas no servidor (Merkel, 2014).

Como pode-se perceber, o método em questão, acabava por envolver o Desenvolvedor em todas as etapas de configuração, manutenção e *deploy*. Ademais, possuía muitos intervalos aos quais poderiam gerar problemas ou lentidão no momento da entrega dos resultados ou novas alterações. É interessante ressaltar que nesse período era muito comum os erros serem resolvidos dentro do próprio servidor, devido à comodidade e ao uso de arquivos, desse modo em muitos casos o versionamento fica incongruente em relação ao GitHub, gerando diversos problemas de versão e atualização.

3.1.2 Abordagem 2 - Utilizando GitHub Actions & Docker Compose

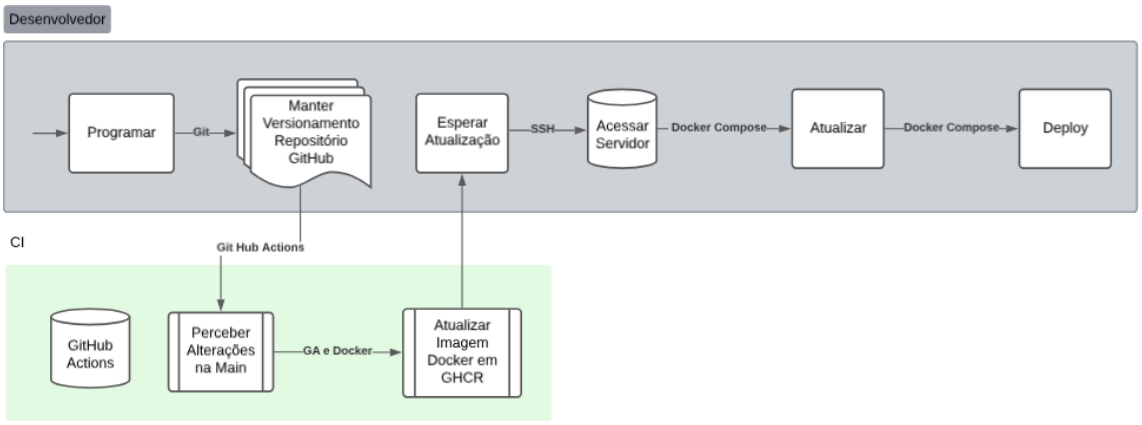

Figura 2: Diagrama Abordagem 2 - Utilizando GitHub Actions & Docker Compose.

Uma vez que anteriormente o método utilizado para manter atualizado o servidor tratava-se de uma abordagem extremamente manual e continha intrinsecamente problemas com compatibilidade, a solução mais viável surgiu com a utilização de *Docker* junto ao GitHub *Actions*. A Figura 2 demonstra que agora, ao adicionar novos dados na ramificação principal do repositório do GitHub, Cargas de Trabalho definidas no GitHub *Actions* inferem que automaticamente sejam criadas imagens *Docker* a partir do repositório em questão, atualizando o GitHub *Registry Repository* (GHCR).

Quando comparada ao processo da Figura 1, pode-se perceber que esta abordagem demonstra novas funcionalidades ao garantir mais eficiência por diminuir a necessidade de validações de incompatibilidade entre as aplicações. Entretanto, ainda existem processos manuais e demorados que envolvem constantemente o

desenvolvedor, os quais seriam: frequentes acessos SSH ao servidor e a necessidade da execução de comandos de *Docker* para baixar novas atualizações, terminar a instância atual dos *containers* e iniciar novamente. Todas essas situações ocupam a equipe e o desenvolvedor em resolver problemas desnecessários, além de exigir a necessidade do conhecimento de ferramentas como *Docker*, aumentando o tempo de treinamento de novos integrantes da equipe.

3.2 Inspiração na Literatura

Diante das limitações das abordagens descritas anteriormente, buscou-se encontrar uma solução que automatizasse completamente o *pipeline* de CI/CD para o contexto do presente trabalho, eliminando a necessidade de intervenção humana nas áreas onde não se existia necessidade e garantindo maior agilidade e confiabilidade na entrega de novas versões da aplicação.

Desse modo, o presente trabalho direcionou a atenção para a contribuição teórica de Sampedro, et al. (2018), que demonstra a implementação de CI/CD em ambientes de HPC utilizando *Singularity* e Jenkins. A partir de inspirações dessa abordagem e a percepção da necessidade de soluções mais genéricas e simplificadas para todos os contextos, desenvolveu-se uma solução que integra GitHub *Actions*, Jenkins e *Docker* para automatizar o *pipeline* de CI/CD das aplicações.

4 ARQUITETURA DA SOLUÇÃO

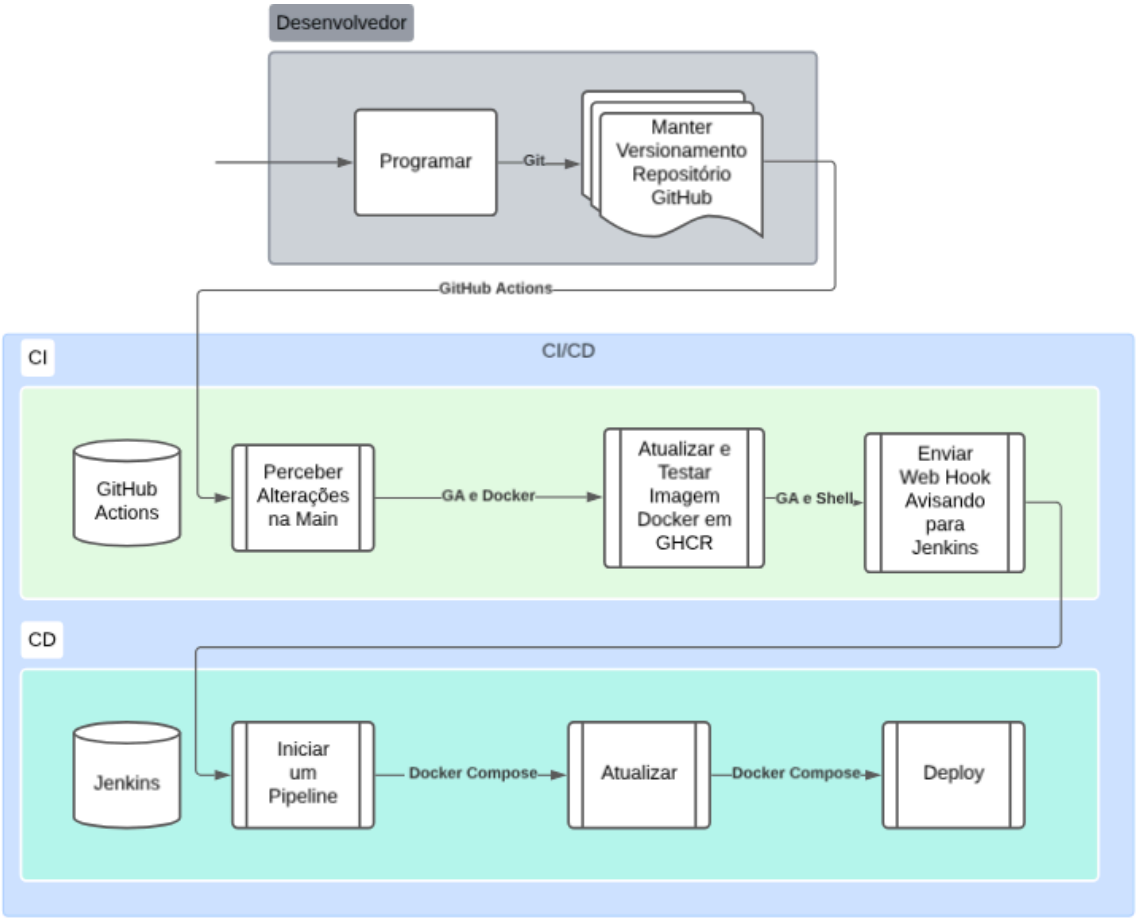

Figura 3: Diagrama do *Pipeline* Automatizado.

A Figura 3 evidencia a solução implementada, a partir do momento em que o código é enviado ao repositório, o processo de *deploy* é totalmente automatizado, sem exigir qualquer intervenção manual. Nessa conjuntura, ressalta-se que diferentemente da solução apresentada por Sampedro et al. (2018), que utiliza *Singularity* para a criação de containers e foca em ambientes de HPC, a presente solução utiliza *Docker* para a conteinerização e se aplica a um contexto mais amplo de desenvolvimento de *software*.

Com o intuito de escrever a abordagem, como pode ser observado na Figura 3, a primeira parte do *pipeline* é gerenciada pelo GitHub *Actions* (GA), algo que se diferencia em relação à inspiração na literatura e foi adaptado ao contexto da aplicação. O GA atua como o gatilho para o processo de CI assim como na Figura 2. Ao detectar alterações no *branch* principal (*main*), o GA inicia automaticamente a execução do *pipeline*, realizando o *build* da aplicação, executando os testes automatizados para validar imagem e, em seguida, criando e enviando a imagem *Docker* atualizada para o GitHub *Container Registry* (GHCR).

Uma vez que a imagem *Docker* está disponível no GHCR, o GitHub *Actions* envia um *webhook,* ao qual se tratam de funções específicas que atuam como gatilhos

de forma remota para a execução de funcionalidades da aplicação, para o Jenkins, acionando a segunda parte do *pipeline* CI. O Jenkins, por sua vez, inicia um novo *job* que realiza o *pull* da imagem *Docker* do GHCR e a implanta no ambiente de produção utilizando *Docker Compose*.

A implementação demonstra tornar o processo bem mais simplificado ao desenvolvedor das aplicações. A parte de configuração para *deploy* fica em uma camada da aplicação mais distante, o que direciona um maior foco da equipe em desempenhar a suas funções, sem a necessidade de se ocupar com outras áreas que exigem um estudo prévio técnico para a configuração.

4 Resultados e Discussão

Os resultados obtidos mostram uma inovação em relação à literatura uma vez que solucionam de forma diferente dos padrões apresentados, dificultando uma comparação direta à contribuição teórica. Dessa forma, essas soluções demonstram que são capazes de colaborar com a diminuição de processos manuais do projeto e colaboram com o aumento significativo da produtividade da equipe. O estudo de ferramentas como Jenkins integrado ao GitHub *Actions* demonstrou tornar o processo extremamente ágil e rápido, com grande eficiência e mais confiabilidade em relação aos métodos descritos nas abordagens 1 e 2.

Tabela 1: Comparação dos diferentes tipos de processos e métodos utilizados no trabalho.

	Erros Manuais	Erros de Compatibilidade	Intervenção Humana	Tempo Médio para Deploy
Abordagem 1	Alto	Alto	Alta	20 Minutos
Abordagem 2	Médio	Baixo	Média	10 Minutos
Pipeline Automatizado	Nenhum	Baixo	Mínima	2 Minutos

A Tabela 1 demonstra uma comparação entre os diferentes tipos de abordagens que foram testados durante o estudo para a realização do projeto de implementação, cabe-se também ressaltar que a coluna “Tempo Médio para *Deploy*” se trata do tempo total ao qual um desenvolvedor fica envolvido na tarefa, de enviar uma atualização ao servidor e ela ser processada e atualizar.

Analisando os resultados da Tabela, pode-se inferir uma evolução do processo de *deploy* ao longo do desenvolvimento da solução. A Abordagem 1, se tratava de algo

totalmente manual, apresentava um alto índice de erros manuais e de compatibilidade além de demandar grande intervenção humana, resultando em grandes tempos de atraso nas entregas de manutenção ou de novas atualizações.

Em segundo momento, analisando a Abordagem 2, com a introdução do *Docker* e GitHub *Actions,* teve-se uma redução significativa em relação aos erros de compatibilidade. Todavia, a intervenção humana ainda era necessária, ou seja, aos desenvolvedores ainda eram requisitados conhecimentos de outras áreas, além da parte de *Front-End* e *Back-End*, precisando também de frequentes acessos ao servidor via SSH, e iniciar os container *Docker* com comandos, o que gerava atrasos de treinamento, produtividade e entrega na equipe.

Por fim, a implementação do *pipeline* automatizado (Figura 3), como pode ser percebido na terceira linha da Tabela 1, resultou na eliminação completa dos erros manuais, além da minimização da intervenção humana no processo, resultando em uma solução extremamente viável, onde os desenvolvedor possuem um tempo médio de envolvimento no *Deploy* das aplicações *Front-End* e *Back-End*, muito baixo em comparação às abordagens 1 e 2, apenas sendo necessário se preocupar com os cuidados de versionamento GitHub e códigos que irão para a *Branch* principal do repositório.

5 CONCLUSÃO

Neste estudo percebe-se que, por meio da demonstração da implementação de um *pipeline* de CI/CD, é possível perceber a eficiência que as áreas de Engenharia de *Software* e DevOps desempenham dentro de uma equipe e que o processo de evolução é inerente à área da tecnologia. Diferentes abordagens e buscas por contribuições teóricas foram utilizadas, o que resultou em uma melhora de produtividade geral do contexto das aplicações, definindo desse modo um ótimo meio de se produzir uma solução adequada para automatização de processos.

Conclui-se que os resultados encontrados exibem um avanço ao implementar a solução, a qual demonstra fornecer grande ajuda ao Desenvolvimento de *Software* devido a abordagem de automatizar todo o processo de um sistema, eliminando erros manuais e diminuindo intensamente a necessidade da intervenção humana em várias partes, definindo assim um tempo para *deploy* menor, e mais eficiente. Concomitantemente, ressalta-se a importância dessas implementações e a necessidade de um profissional da equipe para a configuração desses padrões de automatização, uma vez que elas também colaboram com o desacoplamento do sistema e dos desenvolvedores, de forma a diminuir a necessidade de treinamento em relação às ferramentas de *deploy* em áreas que não são próximas a isso, aumentando ainda mais a produtividade e o tempo útil trabalhado da equipe.

REFERÊNCIAS

Aggarwal, K. K. *Software* Engineering. New Age International, 2005.

Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). DevOps. IEEE *software*, 33(3), 94-100.

Gkatziouras, E. (2022). A Developer's Essential Guide to *Docker* Compose: Simplify the development and orchestration of multi-container applications. Packt Publishing Ltd.

Humble, J., & Farley, D. (2010). Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education.

Kim, G., Behr, K., & Spafford, G. (2013). The Phoenix project: A novel about IT, DevOps, and helping your business win. IT Revolution Press.

Loukides, M. (2012). What is DevOps?. " O'Reilly Media, Inc.".

Makani, Sai Teja, and ShivaDutt Jangampeta. (2022) "THE EVOLUTION OF CICD TOOLS IN DEVOPS FROM JENKINS TO GITHUB ACTIONS."

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. Linux j, 239(2), 2.

Pinheiro, G. M. F. (2022). CI/CD *Pipelines* for Microservice-Based Architectures (Master's thesis).

Rosa, C. S., Vieira, L. M., & Ponciano, L. Simplificação do Processo de Configuração de uma *Pipeline* de Integração e Entrega Contínua no Jenkins.

Shahin, M., Muhammad A. B., and Liming Z.. (2017). "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices.

Sampedro, Z., Holt, A., & Hauser, T. (2018). Continuous integration and delivery for HPC: Using Singularity and Jenkins. In Proceedings of the Practice and Experience on Advanced Research Computing (pp. 1-6).

Wessel, M., Vargovich, J., Gerosa, M. A., & Treude, C. (2023). Github actions: the impact on the pull request process. Empirical *software* Engineering, 28(6), 131.

Zhu, L, Bass, L., & Champlin-Scharff, G. (2016). DevOps and its practices. IEEE software, 33(3), 32-34.