UNIVERSIDADE ESTADUAL DO MARANHÃO
CENTRO DE CIÊNCIAS TECNOLÓGICAS
DEPARTAMENTO DE ENGENHARIA DA COMPUTAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

YAGO ALEXANDRE GOLTARA AFFONSO

AUTOMAÇÃO DE ENTREGA DE *SOFTWARE* COM DOCKER E CI/CD: UM ESTUDO DE CASO EXPERIMENTAL

SÃO LUÍS
2026

YAGO ALEXANDRE GOLTARA AFFONSO

AUTOMAÇÃO DE ENTREGA DE *SOFTWARE* COM DOCKER E CI/CD: UM ESTUDO DE CASO EXPERIMENTAL

Trabalho de Conclusão de Curso apresentado ao curso de Engenharia da Computação da Universidade Estadual do Maranhão, como parte dos requisitos necessários para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: Prof. Me. Pedro Brandão Neto

SÃO LUÍS

2026

YAGO ALEXANDRE GOLTARA AFFONSO

AUTOMAÇÃO DE ENTREGA DE *SOFTWARE* COM DOCKER E CI/CD: UM ESTUDO DE CASO EXPERIMENTAL

Trabalho de Conclusão de Curso apresentado ao curso de Engenharia da Computação da Universidade Estadual do Maranhão, como parte dos requisitos necessários para obtenção do grau de Bacharel em Engenharia da Computação.

Trabalho aprovado em 5 de fevereiro de 2026

Documento assinado digitalmente
gov.br
PEDRO BRANDAO NETO
Data: 20/03/2026 16:27:15-0300
Verifique em https://validar.iti.gov.br

Prof. Me. Pedro Brandão Neto
Orientador

Documento assinado digitalmente
gov.br
EDILSON CARLOS SILVA LIMA
Data: 23/03/2026 14:12:33-0300
Verifique em https://validar.iti.gov.br

Prof. Edilson Carlos Silva Lima
Examinador

Documento assinado digitalmente
gov.br
YONARA COSTA MAGALHAES
Data: 25/03/2026 15:30:08-0300
Verifique em https://validar.iti.gov.br

Profa. Me. Yonara Costa Magalhães
Examinador

Affonso, Yago Alexandre Goltara.
Automação de entrega de software com Docker e CI/CD: um estudo de caso experimental / Yago Alexandre Goltara Affonso. - São Luís - MA, 2026.
59 f.

Monografia (Graduação em Engenharia da Computação) - Universidade Estadual do Maranhão, São Luís, 2026.

Orientador: Prof. Me. Pedro Brandão Neto.

1. DevOps. 2. Docker. 3. Automação de Infraestrutura. 4. Computação em Nuvem. I. Título.

CDU: 004.41

Elaborado por Cassia Diniz - CRB 13/910

Agradecimentos

Primeiramente, agradeço a Deus, por ter me concedido a vida, a saúde e a sabedoria necessárias para percorrer esta jornada. Nos momentos de maior dificuldade e cansaço, foi a fé que renovou minhas forças e me permitiu persistir até a conclusão deste trabalho.

À minha família, minha base e meu refúgio seguro. Em especial aos meus pais, Marcos Affonso e Gérly Iêda, pelo amor incondicional, pelos sacrifícios realizados em prol da minha educação e por sempre acreditarem no meu potencial. Ao meu irmão, Luiz Fellipe, pelo incentivo e parceria constantes.

Ao meu orientador, Prof. Pedro Brandão Neto, pela condução precisa e correções fundamentais para a realização deste trabalho. Aos demais professores, obrigado pelo conhecimento compartilhado ao longo do curso.

Aos meus amigos e parceiros de trabalho, pela convivência diária e pela parceria leal. Agradeço pelas trocas de experiência, pelos debates técnicos e pelo suporte nos desafios profissionais. O ambiente de colaboração que construímos foi essencial para minha evolução prática na engenharia de *software* e infraestrutura. Aos amigos de longa data, obrigado pelos momentos de descontração que tornaram essa caminhada mais leve.

Por fim, a todos que, direta ou indiretamente, fizeram parte desta história, deixo meu muito obrigado.

Se você está sofrendo por coisas externas, não são elas que estão te perturbando, mas o seu próprio julgamento sobre elas. E está em seu poder anular este julgamento agora.

Marco Aurélio

Resumo

O desenvolvimento de *software* enfrenta o desafio crítico de garantir a confiabilidade das entregas em meio a processos manuais propensos a falhas. Este trabalho investiga a implementação técnica da automação de infraestrutura como mecanismo para mitigar inconsistências operacionais. Delimitado a um estudo de caso experimental de escopo acadêmico e controlado, foi desenvolvido um *pipeline* de Integração e Entrega Contínua (CI/CD) para uma aplicação *web* em *Python* (*FastAPI*). A arquitetura utilizou contêineres *Docker* para garantir a paridade de ambientes e o *GitLab CI* para orquestrar os estágios de teste, *build*, segurança e *deploy* em uma instância EC2 na nuvem AWS. A análise dos resultados priorizou a métrica de estabilidade em detrimento da velocidade pura. Os dados evidenciaram que, embora tenha havido uma otimização de 20,35% no tempo médio de entrega (*Lead Time*), o ganho mais expressivo foi a eliminação da variabilidade do processo (redução drástica do desvio padrão), transformando a entrega em uma rotina previsível. Adicionalmente, validou-se tecnicamente a abordagem *Shift-Left* com o uso do *Docker Scout* para varredura de vulnerabilidades. Conclui-se que a automação, mesmo em cenários de menor escala, é essencial para assegurar a reprodutibilidade técnica, a segurança e a estabilidade da engenharia de *software*.

Palavras-chave: DevOps. CI/CD. Docker. Automação de Infraestrutura. Computação em Nuvem.

Abstract

Software development faces the critical challenge of ensuring delivery reliability amidst manual processes prone to failure. This work investigates the technical implementation of infrastructure automation as a mechanism to mitigate operational inconsistencies. Limited to a controlled academic experimental case study, a Continuous Integration and Continuous Delivery (CI/CD) pipeline was developed for a Python web application (FastAPI). The architecture utilized Docker containers to ensure environment parity and GitLab CI to orchestrate testing, build, security, and deploy stages on an EC2 instance in the AWS cloud. The analysis of results prioritized the stability metric over pure speed. The data showed that, while there was a 20.35% optimization in average Lead Time, the most significant gain was the elimination of process variability (drastic reduction in standard deviation), transforming delivery into a predictable routine. Additionally, the Shift-Left approach was technically validated using Docker Scout for vulnerability scanning. It is concluded that automation, even in smaller-scale scenarios, is essential to ensure the technical reproducibility, security, and stability of software engineering.

Keywords: DevOps. CI/CD. Docker. Infrastructure Automation. Cloud Computing.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	Application Programming Interface (Interface de Programação de Aplicações)
AWS	Amazon Web Services
CD	Entrega Contínua (Continuous Delivery)
CI	Integração Contínua (Continuous Integration)
CLI	Command Line Interface (Interface de Linha de Comando)
CPU	Central Processing Unit (Unidade Central de Processamento)
CVE	Common Vulnerabilities and Exposures (Vulnerabilidades e Exposições Comuns)
Dev	Desenvolvimento (Development)
DevOps	Development and Operations (Desenvolvimento e Operações)
DevSecOps	Development, Security, and Operations (Desenvolvimento, Segurança e Operações)
DORA	DevOps Research and Assessment
EC2	Elastic Compute Cloud
FTP	File Transfer Protocol (Protocolo de Transferência de Arquivos)
HTTP	Hypertext Transfer Protocol (Protocolo de Transferência de Hipertexto)
IaC	Infrastructure as Code (Infraestrutura como Código)
IP	Internet Protocol (Protocolo de Internet)
JSON	JavaScript Object Notation
KPI	Key Performance Indicator (Indicador-Chave de Desempenho)
LTS	Long Term Support (Suporte de Longo Prazo)
MTTR	Mean Time to Restore (Tempo Médio para Restauração)
OCI	Open Container Initiative (Iniciativa de Contêineres Abertos)
Ops	Operações (Operations)

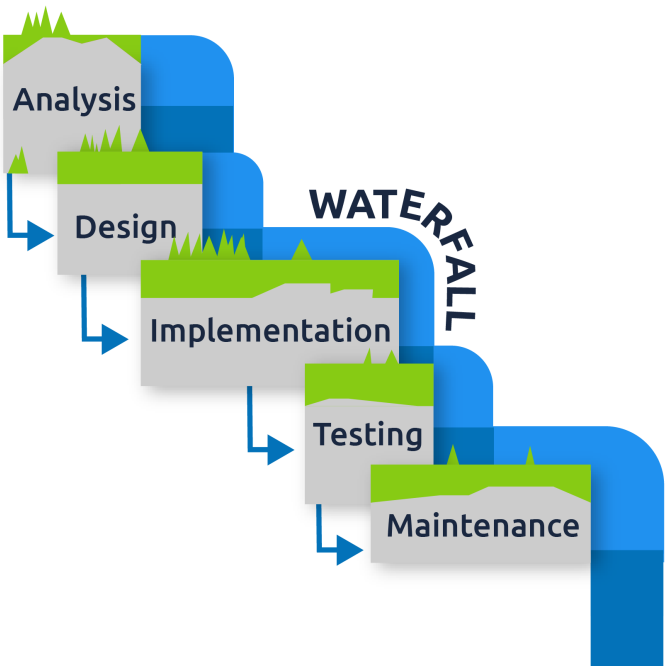
QA	Quality Assurance (Garantia de Qualidade)
REST	Representational State Transfer (Transferência de Estado Representacional)
SAST	Static Application Security Testing (Teste Estático de Segurança de Aplicação)
SO	Sistema Operacional
SSH	Secure Shell
TI	Tecnologia da Informação
VM	Virtual Machine (Máquina Virtual)
YAML	YAML Ain't Markup Language

Sumário

1 INTRODUÇÃO

No panorama contemporâneo da engenharia de *software*, a velocidade de entrega e a confiabilidade das aplicações tornaram-se fatores críticos de sucesso empresarial. Organizações de todos os setores enfrentam uma pressão crescente para inovar e responder às demandas do mercado em ciclos cada vez mais curtos. Contudo, os modelos tradicionais de desenvolvimento de *software*, frequentemente caracterizados por longas fases sequenciais, como o Modelo Cascata ilustrado na Figura 1, e uma separação estrita entre as equipes de desenvolvimento (*Dev*) e operações (*Ops*), criam gargalos que impedem a agilidade necessária. Esse desalinhamento estrutural historicamente resultou em processos de implantação lentos, manuais e propensos a falhas [19].

Figura 1 – Representação do modelo de desenvolvimento em cascata (*Waterfall*)



Fonte: Adaptado de TryHackMe (2024).

Para solucionar o conflito intrínseco entre a necessidade de mudança (desenvolvimento) e a necessidade de estabilidade (operações), emergiu a cultura *DevOps*. O *DevOps* transcende a ideia de ser apenas um conjunto de ferramentas; trata-se de uma transformação cultural que enfatiza a colaboração, a comunicação e a automação de processos para quebrar os silos organizacionais. Conforme descrito por Kim et al. [19], a implementação bem-sucedida do *DevOps* baseia-se em "Três Caminhos": otimização do fluxo de trabalho (*Flow*), amplificação dos *loops* de *feedback* e fomento de uma cultura de experimentação e aprendizado contínuos.

A definição de *DevOps* é corroborada pelos principais provedores de infraestrutura de nuvem, que são facilitadores essenciais da implementação dessas práticas em escala global. A Amazon Web Services (AWS), por exemplo, define *DevOps* como uma combinação de filosofias culturais, práticas e ferramentas destinadas a aumentar a capacidade de uma organização entregar aplicações e serviços em alta velocidade. Esta abordagem permite que as empresas evoluam e melhorem seus produtos em um ritmo mais rápido do que organizações que utilizam processos tradicionais de desenvolvimento de *software* e gerenciamento de infraestrutura [3].

A materialização dos princípios *DevOps* é alcançada através da adoção de práticas de engenharia de *software* disciplinadas, sendo a Integração Contínua (CI) e a Entrega Contínua (CD) seus pilares fundamentais. A Integração Contínua é a prática de automatizar a mesclagem de código de múltiplos contribuidores em um repositório central, onde compilações e testes são executados a cada alteração. Segundo Humble e Farley [15], isso previne o "inferno da integração"(*integration hell* ou *merge hell*) ao fornecer *feedback* rápido sobre a saúde do código. A Entrega Contínua estende a CI ao automatizar a preparação da *release/deploy* do *software*, garantindo que qualquer versão aprovada possa ser implantada em homologação e posteriormente em produção de forma segura e com baixa sobrecarga operacional [25].

A eficácia de um *pipeline* automatizado de CI/CD, no entanto, é frequentemente comprometida pela falta de paridade entre os ambientes de desenvolvimento, teste e produção. Diferenças sutis em versões de bibliotecas, configurações de rede ou dependências do sistema operacional podem invalidar os testes realizados e causar falhas inesperadas durante a implantação final. A tecnologia de contêineres, popularizada massivamente pela plataforma Docker, oferece uma solução robusta para este problema. O Docker permite encapsular a aplicação e todo o seu ecossistema de dependências em uma imagem imutável. Conforme detalhado por Matthias e Kane [22], os contêineres garantem que o artefato de *software* se comporte de maneira idêntica em qualquer infraestrutura *host*, desde a estação de trabalho do desenvolvedor até os servidores de produção em larga escala.

A sinergia entre os processos de CI/CD e a tecnologia de contêineres Docker representa uma evolução paradigmática na forma como o *software* é construído, testado e distribuído. A automação do *pipeline* não apenas reduz o *lead time* – tempo entre a concepção de uma funcionalidade e sua entrega ao usuário –, mas fundamentalmente elimina a variabilidade operacional inerente ao processo humano. Essa abordagem transforma a entrega de *software* de um evento arriscado e artesanal para uma rotina determinística, auditável e previsível, permitindo que as organizações recuperem-se de falhas com maior agilidade [15].

Em suma, a implementação de *pipelines* CI/CD somados à tecnologia de contêineres Docker não é apenas uma otimização de velocidade, mas estabelece a reprodutibilidade técnica necessária para mitigar falhas e garantir a estabilidade de sistemas. O domínio dessa arquitetura tornou-se uma competência essencial para a engenharia de *software* moderna. Este trabalho busca

aprofundar a compreensão técnica dessas tecnologias, detalhando a construção de um cenário experimental controlado para validar a eficiência e a previsibilidade do ciclo automatizado.

1.1 Justificativa

O cenário atual de desenvolvimento de *software* é impulsionado por uma demanda contínua por inovação. As empresas e organizações precisam entregar novas funcionalidades e atualizações em ciclos cada vez mais curtos para se manterem competitivas no mercado. No entanto, os processos de entrega manuais e a falta de padronização entre ambientes criam gargalos significativos, resultando em implantações lentas, arriscadas e propensas a falhas, o que compromete diretamente a qualidade e o valor percebido pelo usuário final.

Para endereçar o desafio da velocidade, a adoção de práticas de Integração Contínua (CI) e Entrega Contínua (CD) tornou-se um pilar da engenharia de *software* moderna. Conforme preconizado por Humble e Farley [15], essas práticas automatizam o ciclo de vida da construção, teste e *deploy* de *software*, permitindo entregas mais rápidas, frequentes e, fundamentalmente, mais seguras e confiáveis.

Contudo, a automação dos processos de CI/CD por si só não resolve o problema crônico da inconsistência entre os ambientes de desenvolvimento, teste e produção. A tecnologia de contêineres, popularizada pela plataforma Docker, é a peça fundamental para garantir essa paridade e sincronia. Ao empacotar a aplicação e todas as suas dependências no que é chamado imagens de contêineres isolados e portáteis, o Docker assegura que o *software* se comporte de maneira idêntica em qualquer infraestrutura, eliminando a principal fonte de erros durante a implantação [6].

A eficácia dessas ferramentas e processos é maximizada quando sustentada por uma cultura colaborativa. A abordagem *DevOps* promove a quebra de silos entre as equipes de desenvolvimento (*Dev*) e operações (*Ops*), fomentando a responsabilidade compartilhada e a comunicação contínua, bem o aprendizado compartido. Segundo Kim et al. [19], essa sinergia cultural é a chave para o sucesso da automação, permitindo que as organizações otimizem seus fluxos de trabalho de ponta a ponta. Afinal, não adianta ter domínio de ferramentas avançadas se a mentalidade das equipes está presa em um modelo retrógrado.

Portanto, este trabalho se justifica pela relevância de investigar a sinergia entre processos (CI/CD), tecnologia (Docker) e cultura (*DevOps*) como uma solução integrada para os desafios contemporâneos da entrega de *software*. A implementação de um *pipeline* automatizado, objeto deste estudo, não apenas representa o estado da arte em práticas de engenharia de *software*, mas também oferece um potencial significativo para a melhoria da eficiência, qualidade e confiabilidade dos sistemas, sendo um tema de grande interesse acadêmico e alta demanda no mercado de tecnologia.

1.2 Objetivos

Implementar um *pipeline* de Integração Contínua e Entrega Contínua (CI/CD) com a tecnologia Docker, a fim de automatizar o ciclo de vida de uma aplicação *web* em *Python* com

framework FastAPI e avaliar o impacto da cultura *DevOps* e das ferramentas empregadas na eficiência do processo de desenvolvimento.

1.2.1 Objetivos Específicos

Para a consecução do objetivo geral, destacam-se os seguintes objetivos específicos:

- Avaliar o impacto da utilização de contêineres Docker na padronização dos ambientes, investigando como a paridade entre desenvolvimento e produção contribui para a confiabilidade das entregas.
- Implementar e analisar um processo automatizado de Integração Contínua (CI) e Entrega Contínua (CD), detalhando as etapas de teste, construção, verificação de segurança e implantação da aplicação.
- Demonstrar tecnicamente como a automação do *pipeline* elimina a variabilidade operacional e o erro humano, assegurando a previsibilidade e a estabilidade do processo de entrega em comparação à execução manual.

1.3 Organização do Trabalho

Este trabalho está estruturado para fornecer uma compreensão progressiva e detalhada sobre a implementação de *pipelines* de CI/CD com Docker, partindo dos conceitos fundamentais até a sua aplicação prática.

O Capítulo 1, a presente Introdução, contextualiza o problema da entrega de *software* tradicional e apresenta a cultura *DevOps*, as práticas de CI/CD e a tecnologia Docker como soluções sinérgicas. Apresenta também a justificativa e os objetivos que norteiam a pesquisa.

O Capítulo 2, Fundamentação Teórica, aprofunda os pilares conceituais do trabalho. São detalhados os princípios da cultura *DevOps*, as mecânicas da Integração e Entrega Contínua, e a arquitetura e funcionamento da tecnologia de contêineres Docker.

O Capítulo 3, Metodologia, descreverá os procedimentos e as ferramentas utilizadas para a construção do *pipeline* de CI/CD. Serão detalhadas a aplicação de exemplo, a configuração do ambiente, as ferramentas de automação selecionadas e os passos para a implementação do fluxo de trabalho.

O Capítulo 4, Resultados e Discussões, apresentará os resultados obtidos com a implementação do *pipeline*. Serão analisados o funcionamento do fluxo automatizado, as métricas de eficiência coletadas e discutidas as implicações práticas da automação na qualidade e velocidade do processo de desenvolvimento.

Finalmente, o Capítulo 5, Considerações Finais, sintetiza os principais achados do trabalho, reforça a importância da integração entre *DevOps*, CI/CD e Docker, e aponta para possíveis trabalhos futuros e aprofundamentos na área.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 A Cultura DevOps

O *DevOps* representa, fundamentalmente, uma mudança cultural que emergiu ao final da década de 2000. Segundo Kim et al. [19], essa cultura visa solucionar os atritos gerados por objetivos inerentemente conflitantes entre as equipes de Desenvolvimento (*Dev*) e Operações (*Ops*). Historicamente, as equipes operavam sob incentivos distintos: enquanto os desenvolvedores eram impulsionados pela necessidade de introduzir novas funcionalidades rapidamente, os profissionais de operações tinham como principal meta garantir a estabilidade e a confiabilidade dos sistemas em produção.

2.1.1 O Cenário Pré-DevOps e a Influência Ágil

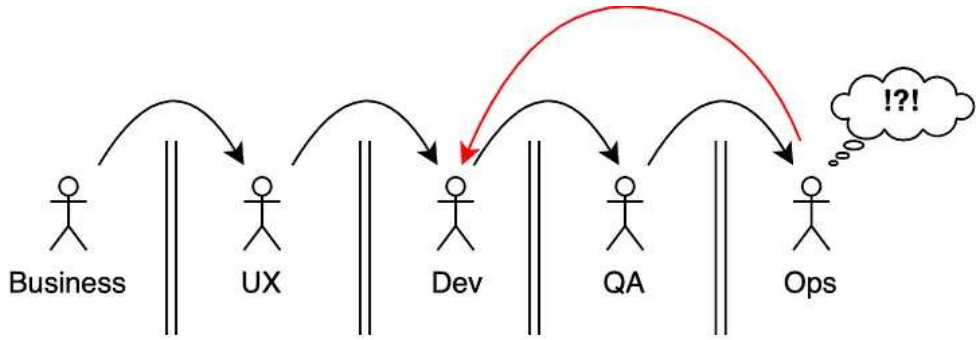
O desalinhamento histórico entre as equipes de Desenvolvimento (*Dev*) e Operações (*Ops*) não se limitava a objetivos conflitantes. Essa divergência era acentuada por uma estrutura organizacional que os segregava: os times eram avaliados por métricas de sucesso distintas (KPIs), respondiam a gestões separadas e, frequentemente, atuavam em lugares onde os membros de uma equipe sequer conheciam os da outra. Tal cenário fomentou uma cultura de responsabilidade limitada, na qual cada profissional possuía uma visão restrita apenas ao seu escopo de atuação.

Essa dinâmica resultava em um ciclo de consequências negativas para o negócio e também para os trabalhadores. Segundo Sato [25], os sintomas mais comuns incluíam a sobrecarga das equipes com trabalho manual e repetitivo, a alta frequência de instabilidades nos ambientes de produção após novas implantações e, como resultado final, a insatisfação dos clientes com o valor entregue pelo produto.

Essa barreira de comunicação e processo entre as equipes é formalmente conhecida na literatura como *Wall of Confusion*, ou "Muro da Confusão". O termo, popularizado por pioneiros do movimento *DevOps* como Andrew Clay Schafer, descreve a ruptura no fluxo de trabalho que ocorre em organizações de TI tradicionais. Um exemplo prático desse fenômeno é a transferência de responsabilidade abrupta, na qual a equipe de desenvolvimento conclui uma nova funcionalidade e entrega o artefato de *software* à equipe de operações sem o devido contexto ou documentação. Nesse cenário, a equipe subsequente é forçada a inferir como o código deve ser testado, implantado e mantido, o que gera ineficiência, atrasos e aumenta o risco de falhas em produção. A Figura 2 retrata de forma emblemática o *Wall of Confusion*.

Com a finalidade de reduzir o enrijecimento advindo da metodologia de desenvolvimento de *software Waterfall Model* (Modelo Cascata, comentado anteriormente na seção de Introdução) e com o surgimento do *Agile*, ilustrado na Figura 3, o foco do desenvolvimento de *software* foi redefinido. Essa nova abordagem, formalizada no Manifesto Ágil, promove um conjunto de

Figura 2 – Representação do Muro da Confusão entre times de TI

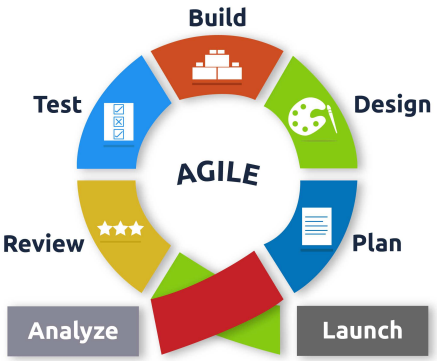


Fonte: Adaptado de Kawaguchi (2020)

valores que, segundo o material de referência do TryHackMe [26], podem ser interpretados da seguinte forma:

- Valorizar mais as pessoas e a forma como colaboram do que se apegar rigidamente a processos e ferramentas pré-definidas;
- Priorizar a entrega de um *software* funcional e que agregue valor real, em vez de produzir uma documentação exaustiva e detalhada;
- Manter uma colaboração contínua com o cliente ao longo do projeto, em detrimento de negociações contratuais inflexíveis;
- Ter a capacidade de se adaptar a mudanças de requisitos ou de cenário, em vez de seguir estritamente um plano inicial.

Figura 3 – O Ciclo de Desenvolvimento Iterativo do Modelo Ágil



Fonte: Adaptado de TryHackMe (2024)

2.1.2 A Gênese do Movimento DevOps

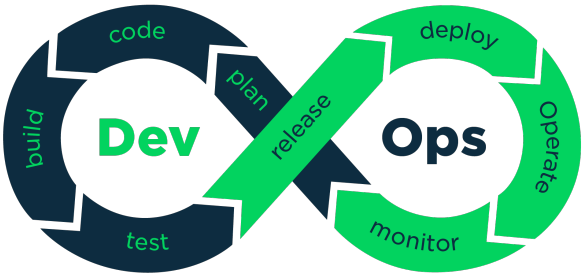
Ainda que a filosofia Ágil tivesse preparado o terreno para a mudança e realizado muitas melhorias, o *Wall of Confusion* entre desenvolvimento e operações persistia como um gargalo

crítico. A virada de chave para o surgimento do *DevOps* é detalhada com precisão no livro "Manual de DevOps"[19], escrito por Gene Kim, Jez Humble, Patrick Debois e John Willis.

A história do movimento, conforme narrada pelos autores, começa a tomar forma em 2008. Na conferência Ágil de Toronto, Patrick Debois e Andrew Schafer promoveram uma sessão informal para discutir a aplicação de princípios ágeis à infraestrutura. Embora o encontro tenha tido pouca adesão, ele serviu para conectar os primeiros entusiastas da ideia, incluindo John Willis, um dos coautores do manual.

O evento catalisador, no entanto, ocorreu no ano seguinte. Durante a conferência Velocity de 2009, John Allspaw e Paul Hammond, então engenheiros do Flickr, apresentaram a palestra seminal "10+ Deploys per Day: Dev and Ops Cooperation at Flickr". Nela, descreveram como a empresa alcançava uma alta frequência de implantações ao estabelecer objetivos compartilhados entre as equipes de Desenvolvimento e Operações, apoiados por práticas de integração contínua. O impacto da apresentação foi imediato e profundo na comunidade. Embora não estivesse presente, Patrick Debois ficou tão entusiasmado com o conceito que, no mesmo ano, criou e organizou o primeiro "DevOpsDays"em Ghent, na Bélgica. Foi nesse evento que o termo "*DevOps*"foi finalmente cunhado, dando nome ao movimento. Vale ressaltar que esse evento "DevOpsDays"é realizado até hoje no mundo todo, incluindo várias edições no Brasil.

Figura 4 – O Ciclo Infinito do DevOps



Fonte: Adaptado de Devopedia (2022)

2.1.3 Os Três Caminhos do DevOps

Para estruturar a implementação da cultura *DevOps*, Gene Kim introduz um modelo prescritivo conhecido como "Os Três Caminhos", descrito tanto no seu livro (Manual de DevOps)[19] quanto em seu *blog* (*IT Revolution*)[20]. Estes não são passos sequenciais, mas sim um conjunto de princípios interdependentes que, quando aplicados em conjunto, permitem a entrega de *software* de forma rápida e confiável, ao mesmo tempo em que promovem um ambiente de trabalho colaborativo e de alta confiança.

- **O Primeiro Caminho: Os Princípios do Fluxo.** Este caminho enfatiza a necessidade de otimizar o fluxo de trabalho da esquerda para a direita, ou seja, do Desenvolvimento até

as Operações e, finalmente, ao cliente. O objetivo é adotar uma visão sistêmica (*Systems Thinking*), focando na performance do sistema como um todo em vez de otimizações locais em silos (*Dev*, *Ops*, QA, etc.). A meta é aumentar a velocidade do fluxo de entrega, reduzindo o tamanho dos lotes de trabalho e os gargalos do processo ao passo que não sobrecarrega as equipes.

- **O Segundo Caminho: Os Princípios do Feedback.** Este princípio foca na criação de ciclos de *feedback* rápidos e constantes da direita para a esquerda. Ao amplificar o retorno de informações de cada etapa do processo, especialmente do ambiente de produção para o desenvolvimento, as equipes podem detectar e corrigir problemas mais cedo, garantindo a qualidade na origem. Isso previne que falhas avancem no fluxo de valor e sobrecarreguem as etapas finais.

- **O Terceiro Caminho: Os Princípios da Aprendizagem e Experimentação Contínua.** O último caminho visa fomentar uma cultura de alta confiança que incentive a experimentação e a aprendizagem contínua. Isso envolve a alocação de tempo para melhorias, a aceitação de riscos como parte do processo de inovação e o tratamento de falhas como oportunidades de aprendizado. A prática e a repetição são vistas como pré-requisitos fundamentais para a maestria e a resiliência organizacional.

2.1.4 Métricas Fundamentais em DevOps

A mensuração de desempenho é um pilar da cultura *DevOps*, permitindo que as equipes avaliem a eficácia de seus processos e identifiquem pontos de melhoria de forma objetiva. Nesse contexto, emergiu um conjunto de quatro métricas-chave, popularizadas pela pesquisa do *DevOps Research and Assessment* (DORA). Estas métricas, detalhadas na obra "Accelerate"de Forsgren et al. [11], são consideradas o padrão da indústria e são divididas em duas categorias principais: métricas de velocidade (*throughput*) e métricas de estabilidade.

1. *Lead Time for Changes* (Tempo de Ciclo para Mudanças): Mede o tempo total decorrido desde o *commit* de um código no controle de versão até a sua efetiva implantação em produção. Esta métrica avalia a eficiência de todo o *pipeline* de entrega, sendo um indicador direto da agilidade do processo.

2. *Deployment Frequency* (Frequência de Implantação): Indica a frequência com que o *software* é implantado com sucesso no ambiente de produção. Equipes de alto desempenho buscam uma alta frequência de implantações de baixo risco, o que geralmente implica em entregar mudanças menores e mais gerenciáveis.

3. *Mean Time to Restore* - MTTR (Tempo Médio para Restauração): Mede o tempo médio necessário para restaurar o serviço após a ocorrência de uma falha em produção, desde a

detecção do incidente até a sua completa resolução. É uma métrica crucial para avaliar a resiliência e a capacidade de resposta da equipe de operações.

4. *Change Failure Rate* (Taxa de Falha em Mudanças): Calcula a porcentagem de implantações em produção que resultam em uma falha, exigindo uma intervenção corretiva como um *hotfix* ou um *rollback*. Esta métrica é um indicador direto da qualidade e da estabilidade do processo de entrega.

Em conjunto, estas quatro métricas oferecem uma visão holística e balanceada, permitindo que as organizações melhorem tanto a velocidade quanto a confiabilidade de sua entrega de *software*.

2.1.5 DevSecOps e a Estratégia Shift-Left

A evolução do *DevOps* trouxe a necessidade de integrar a segurança diretamente no fluxo de entrega, originando o conceito de *DevSecOps*. Central a essa abordagem estáa o princípio de *Shift-Left*, que consiste em antecipar as verificações de segurança para as fases iniciais do desenvolvimento, em vez de realizá-las apenas no final. Segundo Kohgadai, *Principal Product Marketing Manager* da RedHat [21], essa estratégia mitiga gargalos e reduz custos operacionais, permitindo que vulnerabilidades sejam detectadas e corrigidas automaticamente dentro do *pipeline* de CI/CD antes mesmo da implantação em produção.

2.2 Integração e Entrega Contínua (CI/CD)

As práticas de Integração Contínua (CI) e Entrega Contínua (CD) são a materialização técnica dos princípios *DevOps*, utilizando a automação para aumentar a velocidade e a confiabilidade da entrega de *software*. A ausência de um fluxo de trabalho automatizado e visível é uma das fontes primárias de caos em departamentos de TI, um cenário vividamente retratado na obra "O Projeto Fênix"[18], onde o trabalho invisível e as dependências não gerenciadas levam a falhas sistêmicas.

Para solucionar esse problema, a Entrega Contínua propõe a criação de um *pipeline* de implantação (*deployment pipeline*). Conforme definido por Humble e Farley em sua obra seminal "Entrega Contínua: Como Entregar *Software* de Forma Rápida e Confiável"[15], o *pipeline* é um processo automatizado que modela o caminho que o *software* percorre desde o controle de versão até a entrega de valor ao usuário. Cada etapa do *pipeline* aumenta a confiança na qualidade da versão candidata, garantindo que o *software* esteja sempre em um estado implantável.

2.2.1 Integração Contínua (*Continuous Integration* - CI)

A Integração Contínua é uma prática de desenvolvimento na qual os desenvolvedores integram seu código a um repositório central várias vezes ao dia. O principal objetivo é detectar

problemas de integração o mais cedo possível, evitando o que é comumente chamado de "inferno da integração"(*integration hell*), que ocorre quando se tenta mesclar grandes volumes de código conflitante após longos períodos de desenvolvimento isolado [15].

O processo de CI é disparado a cada *commit* no sistema de controle de versão (como o Git) e automatiza as seguintes etapas do *pipeline*:

- **Compilação e *Build*:** O código-fonte é compilado e empacotado, gerando um artefato binário (um executável, um pacote '.jar', uma imagem Docker, etc.). Se esta etapa falhar, o *feedback* é imediato para a equipe.
- **Testes Unitários e de Integração:** Uma suíte de testes automatizados é executada para validar a lógica interna dos componentes (testes unitários) e verificar se eles colaboram entre si corretamente (testes de integração).
- **Análise de Código:** Ferramentas de análise estática são executadas para identificar automaticamente possíveis *bugs*, vulnerabilidades de segurança e violações de padrões de codificação, garantindo a qualidade intrínseca do código.

Ao final de um processo de CI bem-sucedido, a equipe tem um artefato testado e verificado, com um alto grau de confiança de que ele está pronto para avançar para as próximas etapas.

2.2.2 Entrega Contínua (*Continuous Delivery* - CD)

A Entrega Contínua é a extensão lógica da Integração Contínua. Seu objetivo é garantir que todo artefato que passa com sucesso pela fase de CI seja automaticamente preparado e implantado em ambientes que simulam a produção, garantindo que o *software* esteja sempre em um estado implantável [15].

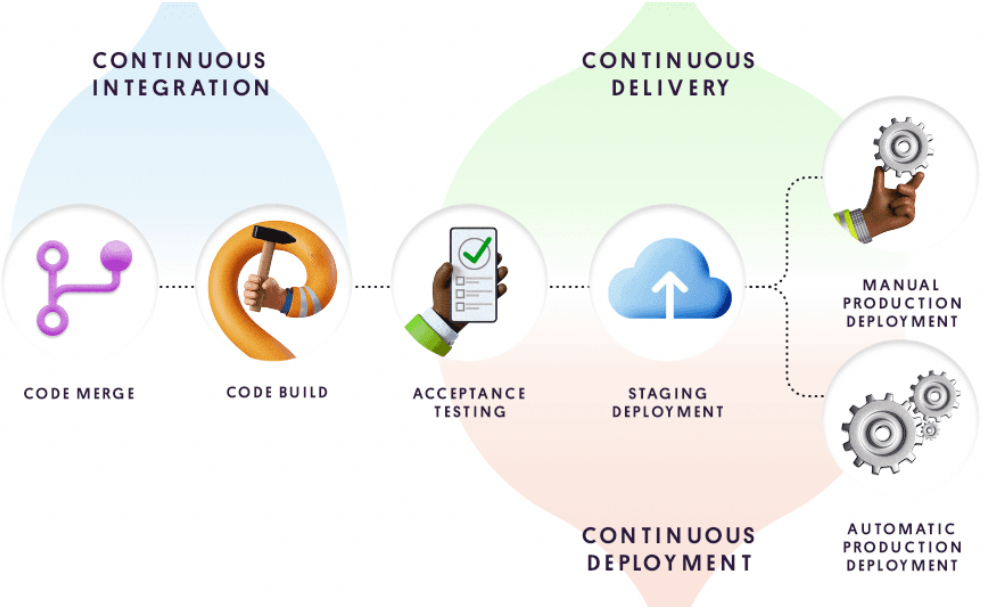
A prática de CD automatiza as etapas finais do *pipeline*:

- **Testes de Aceitação:** O artefato é implantado em um ambiente de testes dedicado, onde uma suíte de testes de aceitação automatizados é executada para verificar se as funcionalidades atendem aos critérios de negócio e aos requisitos do cliente.
- **Implantação em *Staging*:** A versão aprovada é então implantada em um ambiente de *staging* (ou pré-produção), que deve ser uma réplica exata do ambiente de produção. Isso permite a realização de testes manuais exploratórios finais e validações de *performance* ou segurança.
- **Implantação em Produção:** A grande diferença da Entrega Contínua é que o passo final de implantação em produção é um evento de negócio, não técnico. A implantação é um "apertar de botão", que pode ser realizado a qualquer momento, de forma rápida e segura, basta que tenha a ciência e aprovação das equipes.

É importante diferenciar a Entrega Contínua da Implantação Contínua (*Continuous Deployment*). Enquanto na Entrega Contínua o *deploy* final para produção é um passo manual e controlado, na Implantação Contínua, toda alteração que passa por todas as etapas do *pipeline* é implantada em produção *automaticamente*, sem intervenção humana.

Em suma, as práticas de CI e CD se unem para formar um *pipeline* coeso que automatiza o caminho do código desde o *commit* até o *deploy* em produção. A Figura 5 ilustra de forma clara essa sinergia, demonstrando como a fase de Integração Contínua (CI), focada na construção e teste, alimenta a fase de Entrega Contínua (CD), que gerencia a liberação e implantação, completando o ciclo de entrega de valor.

Figura 5 – O Fluxo de um *Pipeline* de CI/CD



Fonte: Adaptado de Unity ([s.d.])

2.3 A Tecnologia de Contêineres

Um dos desafios perenes na engenharia de *software* é garantir a paridade entre os ambientes de desenvolvimento, teste e produção. Historicamente, diferenças sutis nas versões do sistema operacional, do *kernel*, de bibliotecas ou de *frameworks* eram fontes comuns de falhas inesperadas no momento da implantação de um sistema. Com esse problema em mãos surgiu o que conhecemos hoje como "conteinerização".

A conteinerização é uma forma de virtualização no nível do sistema operacional que resolve o problema crônico da incompatibilidade entre ambientes. A técnica consiste em encapsular uma aplicação, juntamente com todas as suas dependências — como bibliotecas, *frameworks* e arquivos de configuração — em uma unidade isolada e autossuficiente chamada contêiner. Isso contrasta diretamente com o método tradicional, no qual o *software* era desenvolvido para um

sistema operacional específico, tornando sua migração para novos ambientes um processo frágil, propenso a *bugs* e que consumia tempo e recursos valiosos.

O principal benefício dessa abordagem é a portabilidade. Uma vez que um *software* é empacotado em um contêiner, ele pode ser executado de forma consistente em qualquer infraestrutura, independentemente do sistema operacional do hospedeiro, pois todas as suas necessidades já estão contidas nele. O contêiner funciona como um invólucro computacional padronizado e portátil, garantindo que a aplicação se comporte da mesma maneira na máquina do desenvolvedor, nos servidores de teste e no ambiente de produção [24].

Embora a tecnologia de isolamento de processos já existisse há décadas, foi a empresa Docker, com o lançamento do Docker Engine em 2013, que democratizou e padronizou o uso de contêineres. Ao fornecer uma interface de linha de comando intuitiva e uma abordagem universal para o empacotamento, o Docker acelerou massivamente a adoção da tecnologia. Esse sucesso culminou na criação da *Open Container Initiative* (OCI), que hoje garante a interoperabilidade entre diversas ferramentas de conteinerização, como Docker, Podman e Buildah, consolidando os contêineres como um pilar da engenharia de *software* moderna [19].

2.3.1 O Problema da Paridade de Ambientes

A inconsistência entre ambientes é a raiz do notório problema resumido pela frase "na minha máquina funciona", que por décadas gerou atritos entre as equipes de desenvolvimento e operações. Conforme aponta Romero [24], essa falta de portabilidade e reprodutibilidade dos artefatos de *software* representava um obstáculo significativo para a automação.

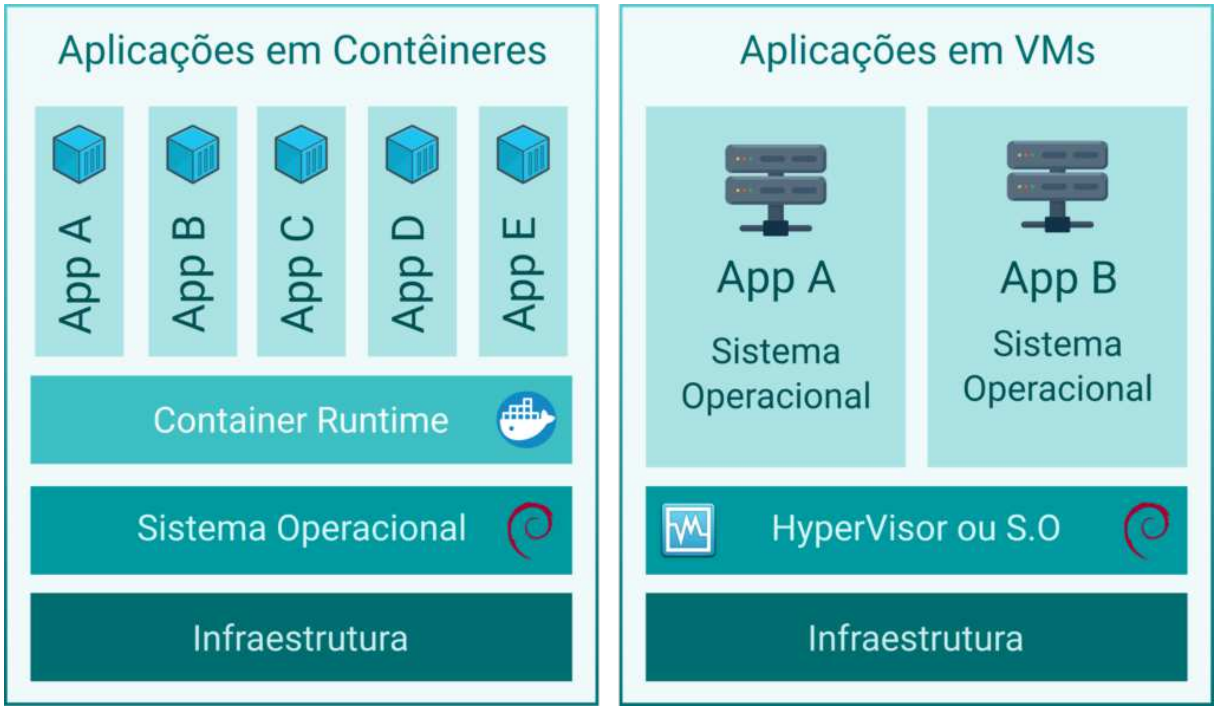
Na prática, esse cenário impunha uma sobrecarga de trabalho manual às equipes: além de lidar com os *bugs* inerentes ao código, os desenvolvedores precisavam verificar e sincronizar constantemente as versões de pacotes e bibliotecas entre os diferentes estágios, adicionando complexidade e uma fonte constante de erros ao processo de desenvolvimento.

2.3.2 Virtualização Tradicional vs. Contêineres

A virtualização tradicional, implementada através de Máquinas Virtuais (VMs), foi por muito tempo a solução padrão para criar ambientes de desenvolvimento e teste isolados. Conforme explica Romero [24], uma VM emula um sistema de *hardware* completo — com sua própria CPU, memória e armazenamento virtuais — sobre o qual um sistema operacional "convidado"inteiro é executado. Essa camada de abstração é gerenciada por um *software* chamado *hypervisor*, que fica entre o *hardware* físico e a VM. Embora eficaz, essa abordagem se tornou um gargalo para as práticas de desenvolvimento ágil, pois o processo de inicialização de uma VM é demorado e seu consumo de recursos é alto, com cada instância ocupando gigabytes de espaço e alocando porções significativas de memória e CPU da máquina hospedeira.

Os contêineres surgiram como uma evolução dessa tecnologia, oferecendo uma abordagem de virtualização mais leve e eficiente. A diferença fundamental, destacada por Romero [24], reside na arquitetura: em vez de virtualizar o *hardware*, os contêineres virtualizam o sistema operacional. Eles compartilham o mesmo *kernel* do sistema hospedeiro e utilizam recursos de isolamento como *Namespaces* (para isolar processos, redes, etc.) e *Cgroups* (para limitar o uso de recursos). Isso significa que um contêiner inclui apenas a aplicação e suas dependências diretas, resultando em um tamanho drasticamente menor, geralmente medido em megabytes. A Figura 6 ilustra essa diferença arquitetônica de forma visual.

Figura 6 – Comparativo arquitetônico entre Máquinas Virtuais e Contêineres



Fonte: Adaptado de 4Linux ([s.d.])

Como consequência direta, os contêineres oferecem vantagens significativas em velocidade e portabilidade: sua inicialização é quase instantânea e o artefato gerado é muito mais leve para ser distribuído. Essa característica os torna a tecnologia ideal para habilitar arquiteturas de baixa acoplagem, como os microsserviços. Conforme argumentado por Kim et al. [19], são essas arquiteturas que permitem que as equipes trabalhem de forma autônoma e implantem com alta

frequência, tornando os contêineres um pilar fundamental para o sucesso dos *pipelines* de CI/CD e da cultura *DevOps*.

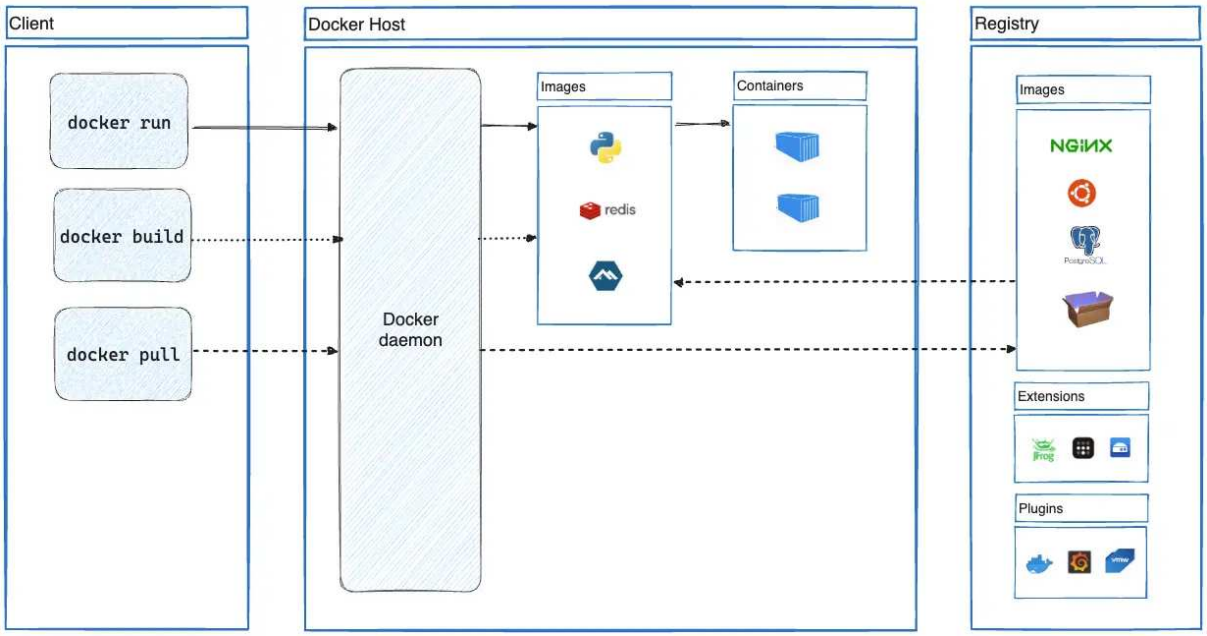
2.3.3 Arquitetura e Componentes do Docker

A plataforma Docker é construída sobre uma arquitetura cliente-servidor. A interação do usuário se dá principalmente através de uma interface de linha de comando (CLI), que se comunica com um serviço de longa execução chamado *Docker Daemon*. Este *Daemon* é o coração do Docker, responsável por construir, executar e gerenciar os contêineres. Conforme detalhado por Romero [24], o ecossistema Docker é composto por vários componentes fundamentais que trabalham em conjunto.

- **Dockerfile:** O ponto de partida para a criação de qualquer contêiner é o *Dockerfile*. Trata-se de um arquivo de texto que funciona como uma "receita", contendo um conjunto de instruções sequenciais (como `FROM`, `RUN`, `COPY`, `CMD`, etc.) que o *Docker Daemon* segue para construir uma imagem de forma automatizada. Além dos comandos nativos, sua flexibilidade é ampliada, pois, conforme Romero (2015, p. 19), "O *Dockerfile* é um arquivo que aceita rotinas em *shell script* para serem executadas". Ele define, portanto, o sistema operacional base, as dependências, os arquivos da aplicação e como ela deve ser executada.
- **Imagem Docker:** Uma imagem é um *template* imutável e somente de leitura, construído a partir das instruções de um *Dockerfile*. Cada instrução no *Dockerfile* cria uma nova "camada"na imagem. Usando a analogia de Romero [24], se a imagem é a "planta de uma casa", ela contém tudo o que é necessário para criar um ambiente executável e padronizado.
- **Contêiner Docker:** Um contêiner é a instância em execução de uma imagem. É a "casa"construída a partir da planta. Múltiplos contêineres podem ser executados a partir da mesma imagem, e cada um opera em um ambiente isolado, com seu próprio sistema de arquivos, rede e processos, sem interferir nos outros ou no sistema hospedeiro.
- **Docker Registry:** Uma vez que uma imagem é construída, ela precisa ser armazenada e distribuída. Um *Registry* é um serviço de armazenamento e distribuição de imagens Docker. O mais conhecido é o **Docker Hub**, um registro público mantido pela Docker Inc., mas as organizações também podem manter seus próprios registros privados para armazenar imagens proprietárias.

Para consolidar a interação entre esses componentes, a Figura 7 ilustra a arquitetura cliente-servidor do Docker. Nela, observa-se como o cliente (*Docker CLI*) envia comandos (como *build*, *pull* e *run*) para o *Docker Daemon*, que por sua vez gerencia as imagens e os contêineres, interagindo com um *Registry* remoto quando necessário para baixar ou enviar imagens.

Figura 7 – A arquitetura cliente-servidor do Docker



Fonte: Adaptado de Docker ([s.d.])

2.4 Sinergia entre DevOps, CI/CD e Docker

A sinergia entre a filosofia colaborativa proposta pelo *DevOps*, a automação dos *pipelines* de CI/CD e a portabilidade tecnológica dos contêineres Docker constituem uma aliança fundamental na engenharia de *software* moderna. Cada componente desempenha um papel crucial para viabilizar a entrega de *software* rápida e confiável.

Nesta tríade, o Docker atua como a fundação tecnológica, endereçando diretamente o problema crônico da paridade de ambientes. Conforme discutido anteriormente, ele permite que uma aplicação seja encapsulada com todas as suas dependências — bibliotecas, pacotes e artefatos — em um único contêiner. Isso garante um comportamento consistente e reprodutível em qualquer infraestrutura, independentemente do sistema operacional hospedeiro [24].

Dessa forma, o notório obstáculo resumido pela frase "na minha máquina funciona"é efetivamente superado. O contêiner Docker torna-se a unidade padrão de implantação que flui através do *pipeline*, resgatando e ampliando o conceito de *Write Once, Run Anywhere* (Escreva uma vez, Rode em qualquer lugar), um princípio anteriormente popularizado pela plataforma Java.

Os *pipelines* de CI/CD desempenham um papel central na materialização dos princípios do *DevOps*, atuando como o mecanismo prático que implementa "Os Três Caminhos"[19]. Primeiramente, ao automatizar e orquestrar o fluxo de trabalho completo — desde o planejamento, passando pelo desenvolvimento e testes, até a implantação em produção — o *pipeline* materializa

o Primeiro Caminho (Princípios do Fluxo), garantindo uma entrega de valor rápida, visível e previsível.

Adicionalmente, o *pipeline* é o principal habilitador do Segundo Caminho (Princípios do Feedback). Ao executar testes automatizados e validações em cada estágio, ele fornece um retorno contínuo e imediato às equipes de desenvolvimento. Conforme Humble e Farley [15] defendem, esse *feedback* rápido é o que permite a detecção e correção de erros no início do ciclo, promovendo a qualidade na origem e viabilizando a melhoria contínua.

Finalmente, o contêiner Docker é o elemento tecnológico que unifica a cultura e o processo. Ele se torna o artefato padrão e imutável que flui através de todo o *pipeline* de implantação. A fase de Integração Contínua (CI) passa a ter como principal objetivo a construção e validação automatizada desse artefato (a imagem Docker). Por sua vez, a fase de Entrega Contínua (CD) se encarrega de promover esse mesmo artefato com segurança através dos ambientes de teste, homologação e, por fim, produção.

Essa abordagem resolve o conflito central do *DevOps*: o que a equipe de Desenvolvimento constrói é exatamente o mesmo que a equipe de Operações executa, garantindo consistência total e eliminando a principal fonte de atrito. Com os pilares conceituais de cultura (*DevOps*), processo (CI/CD) e tecnologia (Docker) devidamente estabelecidos, este trabalho avança agora para a Metodologia, onde será detalhada a implementação prática desta sinergia na construção de um *pipeline* automatizado.

3 METODOLOGIA E PROJETO DA SOLUÇÃO

Para alcançar os objetivos propostos de validar a eficiência da automação de infraestrutura, este capítulo detalha os procedimentos metodológicos e técnicos adotados. A estrutura do trabalho fundamenta-se na construção de um cenário experimental controlado, composto por uma aplicação-alvo e um *pipeline* de CI/CD automatizado, visando demonstrar a resolução dos problemas de paridade de ambiente e agilidade de entrega.

3.1 Metodologia de Pesquisa

Este trabalho caracteriza-se como uma Pesquisa Aplicada, de natureza Experimental. Segundo Gil (2019) [13], a pesquisa aplicada objetiva gerar conhecimentos para aplicação prática, dirigidos à solução de problemas específicos. Para operacionalizar a pesquisa, adotou-se o método de Estudo de Caso Experimental, onde um cenário controlado foi criado para reproduzir os desafios de paridade de ambiente e validar a eficácia da automação proposta

A metodologia foi estruturada em um ciclo de cinco etapas principais, adaptadas do planejamento original do projeto:

1. **Definição do Problema e Cenário:** Nesta etapa inicial, caracterizou-se o cenário de estudo simulado, definindo as variáveis problemáticas inerentes ao desenvolvimento de *software* tradicional. O foco foi delimitar gargalos específicos, como a intervenção manual excessiva e a falta de paridade entre ambientes, justificando a automação como meio de reduzir a variância entre execuções e garantir a previsibilidade técnica.

2. **Projeto da Solução (*Design*):** A partir da delimitação do problema, projetou-se a arquitetura técnica da solução. Nesta fase, foram selecionadas as ferramentas (Docker, GitLab CI/CD) e desenhado o fluxo de automação, fundamentando-se nos princípios de engenharia de Entrega Contínua [15] e nas práticas do "Manual de DevOps"[19] para garantir a robustez teórica do artefato proposto.

3. **Implementação do Protótipo:** Consistiu na construção técnica dos artefatos propostos. Nesta fase, foram codificados a aplicação-alvo, o *script* de configuração de contêiner (*Dockerfile*) e as rotinas (*stages*) de orquestração do *pipeline*, materializando o projeto em um ambiente de laboratório controlado.

4. **Testes e Validação:** O protótipo implementado foi submetido a execuções controladas para validar sua funcionalidade técnica. A validação focou em verificar se o *pipeline* automatizado cumpria os requisitos de integração, segurança e entrega, utilizando as métricas de estabilidade e velocidade (baseadas no DORA) como critérios de sucesso.

5. **Análise dos Resultados:** Por fim, os dados obtidos nas validações foram compilados e analisados. Realizou-se um comparativo técnico entre o cenário tradicional (manual) e a solução automatizada, discutindo a eficácia do *pipeline* em resolver os problemas de inconsistência e lentidão identificados na primeira etapa.

3.2 Descrição da Aplicação Alvo

Para validar a implementação do *pipeline* de CI/CD, foi desenvolvida uma aplicação-alvo que simula um cenário de desenvolvimento controlado, porém realista. A aplicação desenvolvida foi uma API *RESTful* em *Python*, utilizando o *framework* FastAPI, devido à sua alta performance, leveza e facilidade de conteinerização, características que agilizam os ciclos de *build* e teste.

O escopo da aplicação foi intencionalmente simplificado para focar na estrutura do *pipeline*, servindo como um artefato padronizado para validação. A API consiste em um *endpoint* raiz para verificação e em um *endpoint* de saúde (`/health`), que retorna o *payload* JSON `{"status": "UP"}`.

O código-fonte completo da aplicação (`main.py`) é apresentado na Figura 8, seguido pelo seu respectivo teste de integração (`test_main.py`) na Figura 9.

Figura 8 – Código-fonte da aplicação-alvo em FastAPI (main.py).

```python
from fastapi import FastAPI
from typing import Dict

# Inicializa a aplicação FastAPI
app = FastAPI()

# Endpoint para fins de testes de integração
@app.get("/health")
def health_check() -> Dict[str, str]:
    """
    Endpoint simples para verificar se a aplicação está no ar.
    Retorna um JSON: {"status": "UP"}
    """
    return {"status": "UP"}

# Endpoint raiz/base
@app.get("/")
def read_root():
    return {"message": "API da Aplicação-Alvo do TCC está no ar!"}
```

Fonte: O autor (2025)

Figura 9 – Código do teste de integração da aplicação (test_main.py).

```python
from fastapi.testclient import TestClient
from main import app

# Cria um cliente de teste
client = TestClient(app)

def test_health_check():
    """
    Testa se o endpoint /health está funcionando e
    retornando o JSON esperado.
    """
    response = client.get("/health")

    # Verifica se o status HTTP é 200 (OK)
    assert response.status_code == 200

    # Verifica se o JSON retornado é exatamente {"status": "UP"}
    assert response.json() == {"status": "UP"}
```

Fonte: O autor (2025)

3.3 Arquitetura do Pipeline de CI/CD Proposto

Esta seção detalha a arquitetura técnica da solução implementada, justificando as ferramentas selecionadas e descrevendo o fluxo de trabalho automatizado, desde o *commit* do código até a implantação em produção, com base no fluxo definido para este estudo de caso.

3.3.1 Ferramentas Selecionadas

A seleção das ferramentas foi um passo crucial para a construção de um *pipeline* eficiente. Para este trabalho, os seguintes componentes foram escolhidos, com base no fluxo proposto:

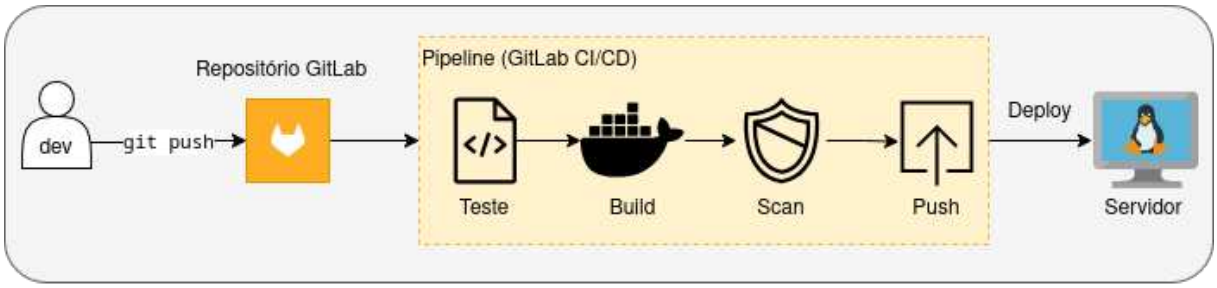
- **Controle de Versão e Repositório: Git e GitLab.** O GitLab foi utilizado como a plataforma central, hospedando o repositório de código-fonte utilizando a ferramenta de controle de versionamento Git.
- **Ferramenta de CI/CD: GitLab CI/CD.** Foi utilizada a solução de integração contínua nativa do GitLab. A escolha se justifica pela sua integração completa com o repositório, permitindo que a definição do *pipeline* (através do arquivo '.gitlab-ci.yml') e o controle de versão de código sejam gerenciadas na mesma interface.

- **Tecnologia de Contêiner: Docker.** Conforme fundamentado no Capítulo 2, o Docker foi a escolha para encapsular a aplicação, suas bibliotecas e dependências, garantindo a paridade de ambientes.
- **Análise de Segurança: Docker Scout.** Para a etapa de verificação de segurança da imagem Docker, foi selecionado o *Docker Scout*. Esta ferramenta é especializada em analisar as camadas da imagem em busca de vulnerabilidades de segurança conhecidas (CVEs), alinhando o *pipeline* com as práticas de *DevSecOps*.
- **Registro de Contêineres: Docker Hub.** O Docker Hub foi o registro de imagens escolhido para armazenar e distribuir publicamente as imagens Docker construídas e verificadas pelo *pipeline*.
- **Ambiente de Implantação: AWS EC2.** Para o ambiente de produção, foi provisionada uma instância EC2 na AWS, executando uma distribuição Linux (Ubuntu Server) com o *Docker Engine* instalado e configurado para receber a aplicação.

3.3.2 Diagrama da Arquitetura

O fluxo de trabalho automatizado, desde o *commit* do desenvolvedor até a implantação final, pode ser visualizado no diagrama da arquitetura do *pipeline* proposto, conforme ilustrado na Figura 10. O *pipeline* é acionado quando o desenvolvedor envia seu código ao repositório GitLab, que por sua vez orquestra todas as etapas de validação e entrega.

Figura 10 – Diagrama da Arquitetura do Pipeline de CI/CD Implementado



Fonte: O autor (2025)

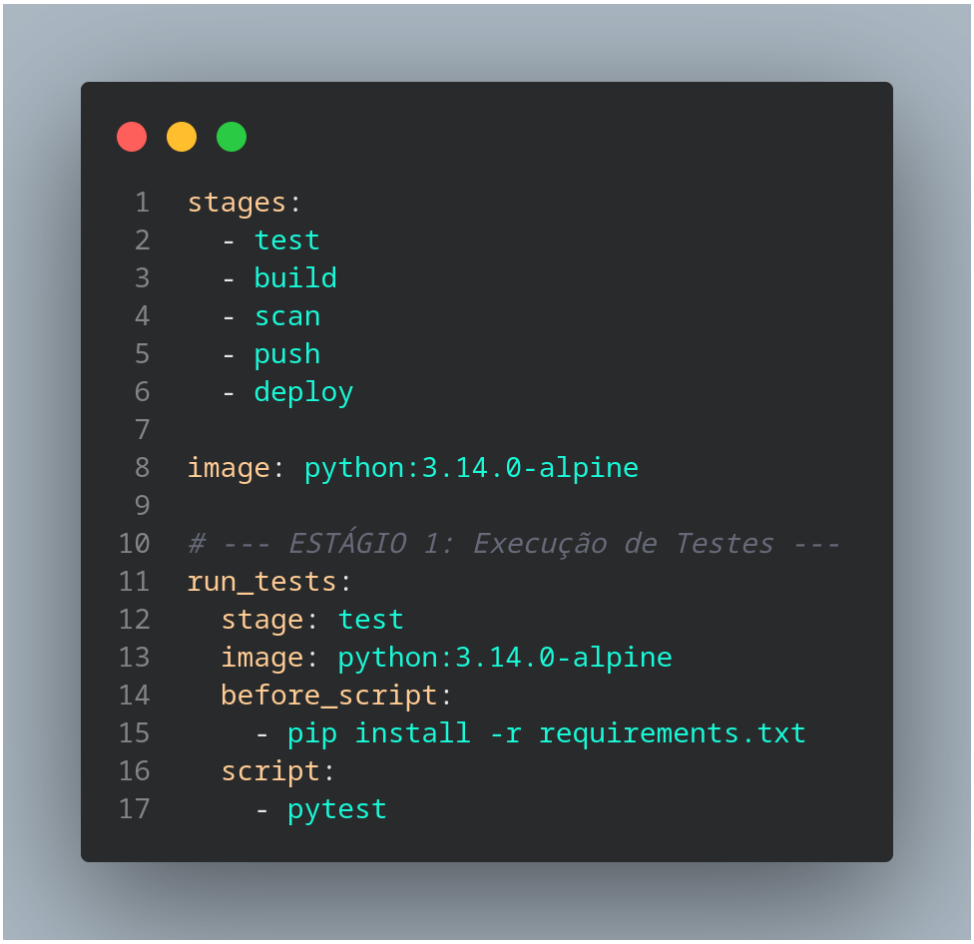
3.3.3 Detalhamento das Etapas do Pipeline

O *pipeline* é configurado através de um arquivo de definição denominado `.gitlab-ci.yml`, posicionado na raiz do repositório. É neste arquivo que todos os estágios (*stages*) do fluxo de trabalho são declarados. A lógica operacional do *pipeline* é acionada por um gatilho (*trigger*) que, para este projeto, foi configurado para disparar a cada evento de *push* no repositório. Uma vez acionado, o GitLab CI executa a sequência de estágios pré-definidos com o objetivo de validar e entregar a aplicação.

Para maior clareza, em vez de apresentar o arquivo completo, os trechos de código YAML de cada estágio serão apresentados individualmente ao passo que forem sendo apresentados os *stages*. O fluxo de trabalho implementado neste projeto segue as seguintes etapas:

- **Estágio 1 - Teste de Código:** Após o *commit* e *push* do desenvolvedor, o *GitLab Runner* inicia o primeiro estágio. Ele é responsável por executar a suíte de testes automatizados. O principal **teste de integração** deste estágio verifica se o *endpoint* `/health` retorna um *status code* 200 (OK) e o *payload* JSON esperado: `{"status": "UP"}`. Esta é a primeira barreira de qualidade; se qualquer teste falhar, o *pipeline* é interrompido e o desenvolvedor é notificado.

Figura 11 – Definição do estágio `test` no `.gitlab-ci.yml`



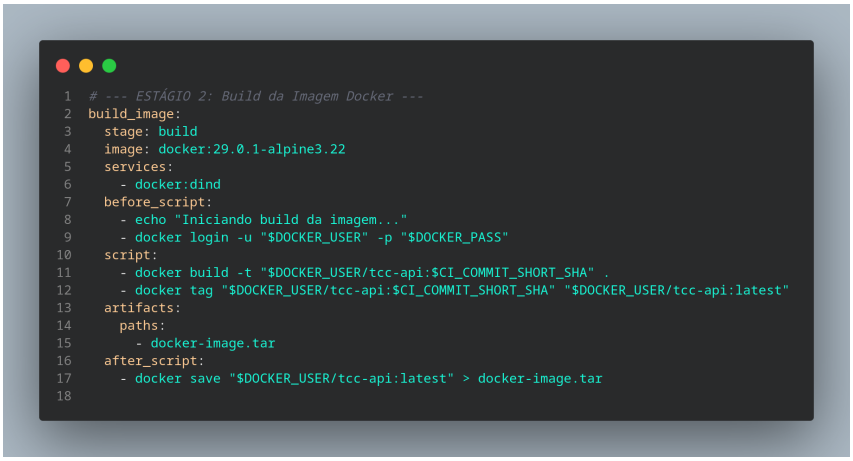
```yaml
stages:
  - test
  - build
  - scan
  - push
  - deploy

image: python:3.14.0-alpine

# --- ESTÁGIO 1: Execução de Testes ---
run_tests:
  stage: test
  image: python:3.14.0-alpine
  before_script:
    - pip install -r requirements.txt
  script:
    - pytest
```

Fonte: O autor (2025)

- **Estágio 2 - Construção da Imagem:** Com a aprovação do código-fonte na etapa anterior, o *pipeline* prossegue para a construção do **artefato imutável** da aplicação. Esta etapa executa o comando *docker build*, que utiliza a "receita"/"blueprint"de construção definida no *Dockerfile* — apresentada na Figura 13 — para encapsular o código e todas as suas dependências. Ao final deste estágio, o *Docker Engine* gera uma imagem Docker

padronizada, garantindo a portabilidade e a consistência que serão validadas nas etapas seguintes.

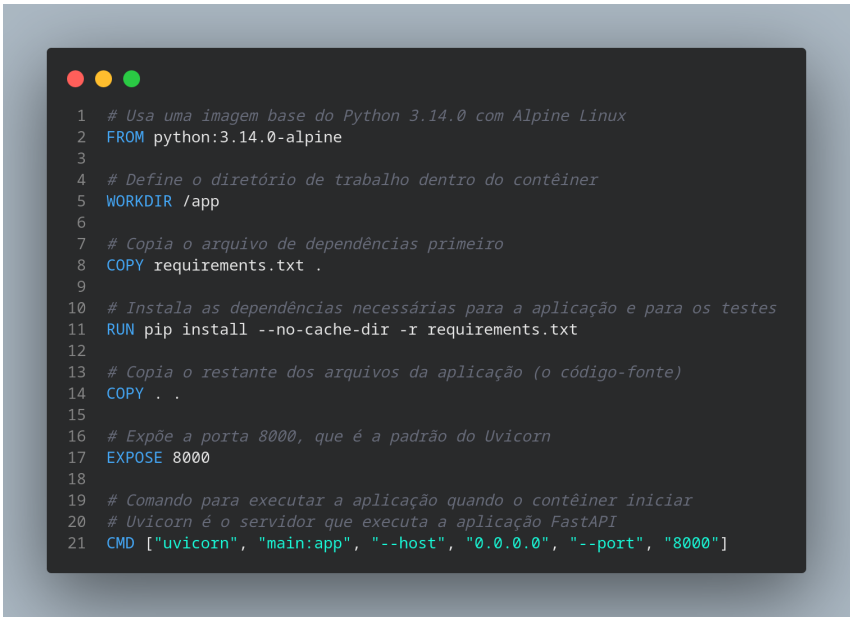
Figura 12 – Definição do estágio `build` no `.gitlab-ci.yml`



```yaml
# --- ESTÁGIO 2: Build da Imagem Docker ---
build_image:
  stage: build
  image: docker:29.0.1-alpine3.22
  services:
    - docker:dind
  before_script:
    - echo "Iniciando build da imagem..."
    - docker login -u "$DOCKER_USER" -p "$DOCKER_PASS"
  script:
    - docker build -t "$DOCKER_USER/tcc-api:$CI_COMMIT_SHORT_SHA" .
    - docker tag "$DOCKER_USER/tcc-api:$CI_COMMIT_SHORT_SHA" "$DOCKER_USER/tcc-api:latest"
  artifacts:
    paths:
      - docker-image.tar
  after_script:
    - docker save "$DOCKER_USER/tcc-api:latest" > docker-image.tar

```

Fonte: O autor (2025)

Figura 13 – Código-fonte do Dockerfile da aplicação.

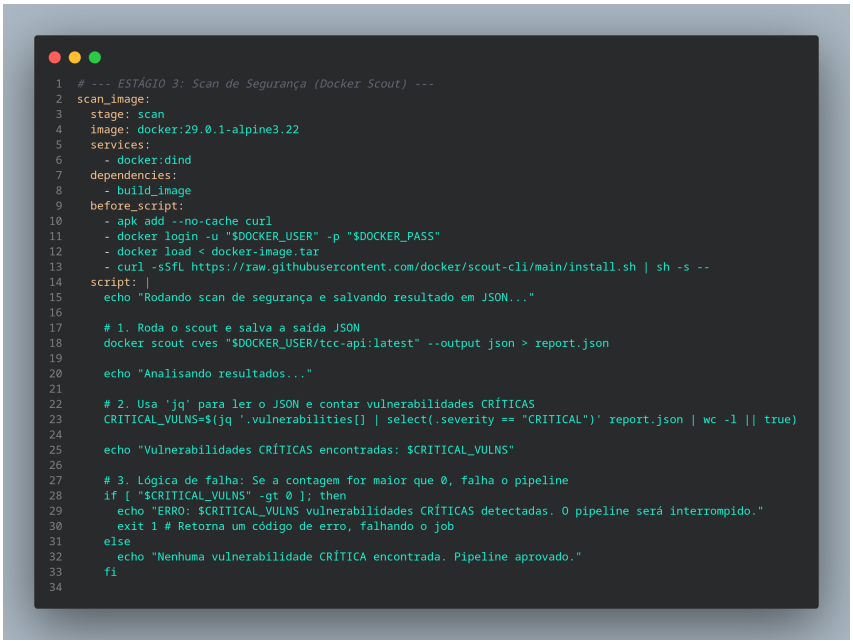


```dockerfile
# Usa uma imagem base do Python 3.14.0 com Alpine Linux
FROM python:3.14.0-alpine

# Define o diretório de trabalho dentro do contêiner
WORKDIR /app

# Copia o arquivo de dependências primeiro
COPY requirements.txt .

# Instala as dependências necessárias para a aplicação e para os testes
RUN pip install --no-cache-dir -r requirements.txt

# Copia o restante dos arquivos da aplicação (o código-fonte)
COPY . .

# Expõe a porta 8000, que é a padrão do Uvicorn
EXPOSE 8000

# Comando para executar a aplicação quando o contêiner iniciar
# Uvicorn é o servidor que executa a aplicação FastAPI
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Fonte: O autor (2025)

- **Estágio 3 - Análise de Segurança (*security_scan*):** Este é um passo crítico de *DevSecOps*. Antes de a imagem ser compartilhada, ela é analisada pela ferramenta *Docker Scout*. O *scanner* inspeciona as camadas da imagem em busca de vulnerabilidades de segurança conhecidas. O *pipeline* é configurado para falhar caso sejam encontradas vulnerabilidades de criticidade crítica, impedindo que *software* inseguro seja promovido.

Figura 14 – Definição do estágio `scan` no `.gitlab-ci.yml`

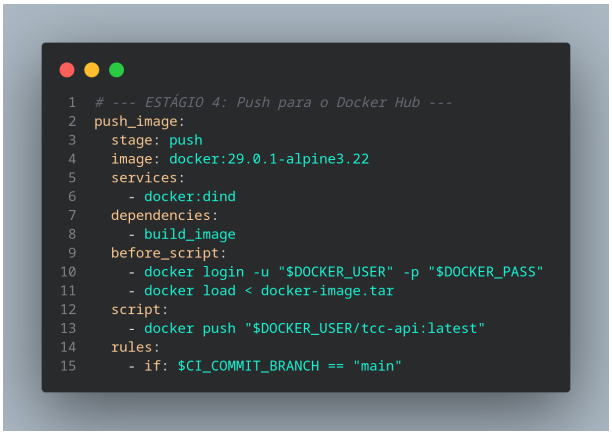


```yaml
# --- ESTÁGIO 3: Scan de Segurança (Docker Scout) ---
scan_image:
  stage: scan
  image: docker:29.0.1-alpine3.22
  services:
    - docker:dind
  dependencies:
    - build_image
  before_script:
    - apk add --no-cache curl
    - docker login -u "$DOCKER_USER" -p "$DOCKER_PASS"
    - docker load < docker-image.tar
    - curl -sSfL https://raw.githubusercontent.com/docker/scout-cli/main/install.sh | sh -s --
  script: |
    echo "Rodando scan de segurança e salvando resultado em JSON..."

    # 1. Roda o scout e salva a saída JSON
    docker scout cves "$DOCKER_USER/tcc-api:latest" --output json > report.json

    echo "Analisando resultados..."

    # 2. Usa 'jq' para ler o JSON e contar vulnerabilidades CRÍTICAS
    CRITICAL_VULNS=$(jq '.vulnerabilities[] | select(.severity == "CRITICAL")' report.json | wc -l || true)

    echo "Vulnerabilidades CRÍTICAS encontradas: $CRITICAL_VULNS"

    # 3. Lógica de falha: Se a contagem for maior que 0, falha o pipeline
    if [ "$CRITICAL_VULNS" -gt 0 ]; then
      echo "ERRO: $CRITICAL_VULNS vulnerabilidades CRÍTICAS detectadas. O pipeline será interrompido."
      exit 1 # Retorna um código de erro, falhando o job
    else
      echo "Nenhuma vulnerabilidade CRÍTICA encontrada. Pipeline aprovado."
    fi
```

Fonte: O autor (2025)

- **Estágio 4 - *Push* para o *Registry* (*push*):** Uma vez que a imagem foi construída e aprovada na varredura de segurança, ela é "etiquetada"(*taggeada*) com uma versão e enviada, através do comando *docker push*, para o Docker Hub. Neste ponto, a imagem se torna um artefato imutável e validado, pronto para a implantação em qualquer ambiente, bastando apenas que possua o *Docker Engine* para executar os contêineres.

Figura 15 – Definição do estágio `push` no `.gitlab-ci.yml`



```yaml
# --- ESTÁGIO 4: Push para o Docker Hub ---
push_image:
  stage: push
  image: docker:29.0.1-alpine3.22
  services:
    - docker:dind
  dependencies:
    - build_image
  before_script:
    - docker login -u "$DOCKER_USER" -p "$DOCKER_PASS"
    - docker load < docker-image.tar
  script:
    - docker push "$DOCKER_USER/tcc-api:latest"
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
```

Fonte: O autor (2025)

- **Estágio 5 - Implantação (*deploy*):** A etapa final do *pipeline* concretiza a implantação contínua da aplicação. Neste estudo de caso, o ambiente que simula a produção é uma instância AWS EC2 (*Elastic Compute Cloud*) com Linux Ubuntu 22.04, previamente configurada com o *Docker Engine*. O *GitLab Runner* estabelece uma conexão segura via

SSH e executa remotamente os comandos para atualizar o ambiente. O processo consiste em baixar a nova imagem do Docker Hub, interromper o contêiner antigo e iniciar a nova versão, mapeando a aplicação para a porta **80 (HTTP)**, garantindo que o serviço esteja acessível publicamente através da *internet*. A definição deste estágio no arquivo de configuração é apresentada na Figura 16.

Figura 16 – Definição do estágio `deploy` no `.gitlab-ci.yml`

```yaml
# --- ESTÁGIO 5: Implantação (Deploy) ---
deploy_prod:
  stage: deploy
  image: alpine:latest
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
  before_script:
    - apk add --no-cache openssh-client
    - eval $(ssh-agent -s)
    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - ssh-keyscan -H "$SERVER_IP" >> ~/.ssh/known_hosts
  script:
    - echo "Conectando ao servidor AWS EC2 ($SERVER_IP)..."
    - ssh -o StrictHostKeyChecking=no $SERVER_USER@$SERVER_IP "
        echo '1. Autenticando no Docker Hub...' &&
        docker login -u $DOCKER_USER -p $DOCKER_PASS &&

        echo '2. Baixando a imagem mais recente...' &&
        docker pull $DOCKER_USER/tcc-api:latest &&

        echo '3. Removendo container antigo (se existir)...' &&
        docker stop tcc-api || true &&
        docker rm tcc-api || true &&

        echo '4. Subindo o novo container na porta 80...' &&
        docker run -d --name tcc-api -p 80:8000 --restart always $DOCKER_USER/tcc-api:latest"

    - echo "Deploy realizado com sucesso!"
```

Fonte: O autor (2025)

3.4 Procedimento de Coleta e Análise de Dados

Para validar a hipótese de melhoria na eficiência e estabilidade, foi realizado um experimento comparativo controlado entre o processo manual e o automatizado.

O experimento consistiu na execução de três ciclos completos de entrega (*end-to-end*) para cada cenário. Para garantir a paridade técnica e a validade da comparação, o cenário manual foi executado seguindo rigorosamente as mesmas cinco etapas definidas no pipeline automatizado (Testes, Build, Scan de Segurança, Push e Deploy), sendo cronometrado o tempo de cada etapa individualmente.

As métricas coletadas focaram em dois indicadores principais do DORA (*DevOps Research and Assessment*):

- ***Lead Time for Changes:*** O tempo total decorrido desde o início do processo até a aplicação estar rodando atualizada no servidor.
- ***Estabilidade e Variabilidade:*** A análise do desvio padrão entre as execuções para identificar a consistência e a previsibilidade do processo.

Os dados temporais foram registrados a partir dos *logs* de data/hora do GitLab CI (para o cenário automatizado) e através de cronometragem direta (para o cenário manual), sendo posteriormente tabulados para análise comparativa no Capítulo 4.

4 IMPLEMENTAÇÃO E RESULTADOS

Este capítulo apresenta a execução prática do projeto definido na metodologia. Serão detalhados a configuração do ambiente experimental na nuvem AWS, a execução passo a passo do *pipeline* arquitetado e a validação final da aplicação no ambiente de produção.

4.1 Configuração do Ambiente Experimental

4.1.1 Provisionamento da Infraestrutura na Nuvem

O ambiente de produção foi provisionado na nuvem da Amazon Web Services (AWS), utilizando uma instância EC2 do tipo *t2.micro* com sistema operacional Ubuntu Server 22.04 LTS. Para garantir a acessibilidade e o gerenciamento remoto, o *Security Group* foi configurado para permitir o tráfego de entrada nas portas 22 (SSH) e 80 (HTTP) a partir de qualquer origem (0.0.0.0/0).

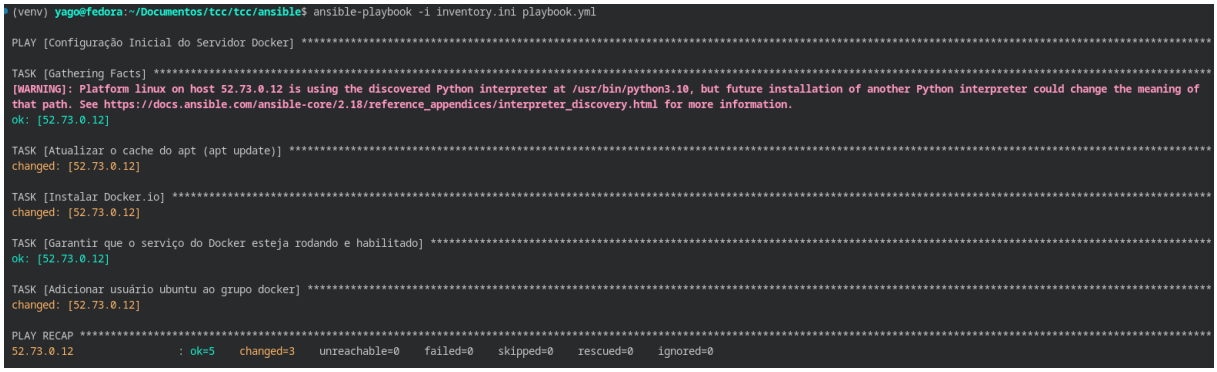
Adotando práticas de Infraestrutura como Código (IaC), o processo de criação e configuração foi totalmente automatizado. Utilizou-se o *Terraform* para a orquestração dos recursos de infraestrutura (instância, grupo de segurança e par de chaves SSH), cuja execução bem-sucedida é apresentada na Figura 17. Subsequentemente, a configuração do sistema operacional foi realizada via *Ansible*, através de um *playbook* responsável por atualizar os pacotes do sistema, instalar o *Docker Engine*, habilitar o serviço e configurar as permissões do usuário padrão (ubuntu) para execução de contêineres, conforme demonstrado no *log* de execução da Figura 18. A Figura 19 demonstra a instância ativa no painel da AWS após a execução dos *scripts* de automação.

Figura 17 – Log de execução do Terraform: Provisionamento da infraestrutura.

```
Plan: 5 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + comando_ssh        = (known after apply)
  + instance_public_ip = (known after apply)
tls_private_key.pk: Creating...
aws_security_group.allow_ssh_http: Creating...
tls_private_key.pk: Creation complete after 1s [id=99a28729b03e2c2b0a0edfbfe85e1174911eb86a]
aws_key_pair.kp: Creating...
local_file.pem_file: Creating...
local_file.pem_file: Creation complete after 0s [id=b2c705f9fbc962f7e941d6fe54df3db38a371f44]
aws_key_pair.kp: Creation complete after 0s [id=acesso-terraform]
aws_security_group.allow_ssh_http: Creation complete after 5s [id=sg-011e968f165f0f203]
aws_instance.server_free_tier: Creating...
aws_instance.server_free_tier: Still creating... [00m10s elapsed]
aws_instance.server_free_tier: Creation complete after 15s [id=i-074b7e490010d020a]

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Outputs:

comando_ssh = "ssh -i acesso.pem ubuntu@52.73.0.12"
instance_public_ip = "52.73.0.12"
```

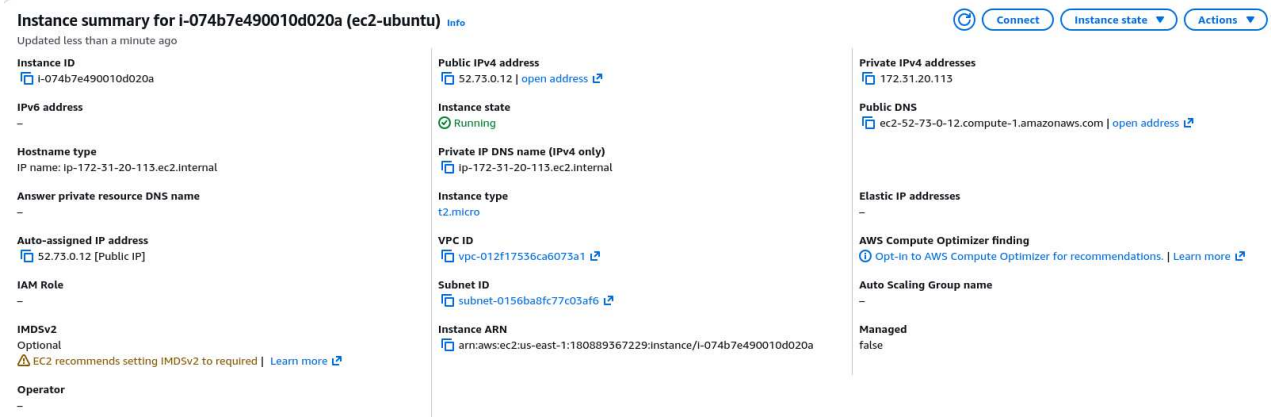
Fonte: O autor (2025)

Figura 18 – Log de execução do Ansible: Configuração do Docker na instância.



Fonte: O autor (2025)

Figura 19 – Instância EC2 provisionada e em execução no painel da AWS.

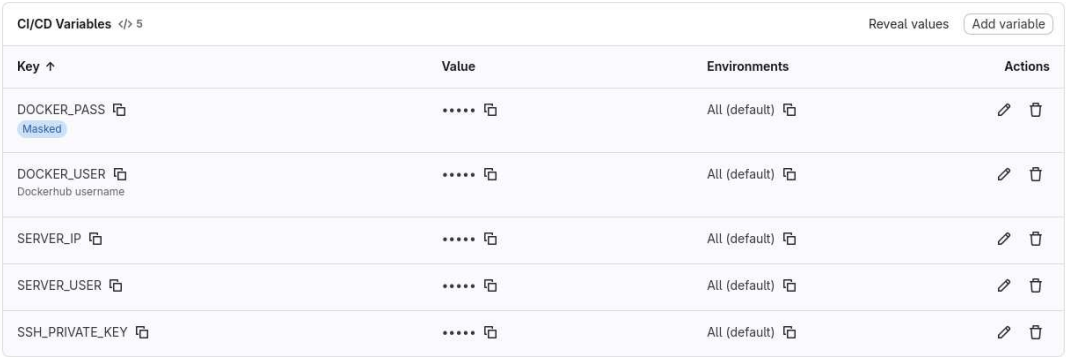


Fonte: O autor (2025)

4.1.2 Configuração de Segredos e Variáveis

Em conformidade com as boas práticas de *DevSecOps*, evitou-se a exposição de dados sensíveis diretamente no código-fonte (*hardcoding*). Para isso, utilizou-se o recurso de variáveis de CI/CD do GitLab, que permite armazenar segredos de forma segura e mascarada nos *logs* de execução. Foram configuradas as credenciais de autenticação do Docker Hub, bem como os dados de conexão SSH para o servidor de produção (chave privada, endereço IP e usuário), conforme detalhado na Figura 20.

Figura 20 – Painel de configuração de variáveis de ambiente no GitLab CI/CD.

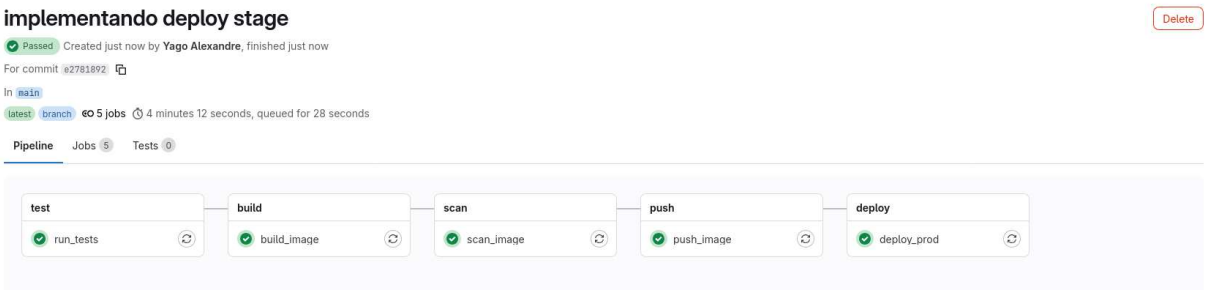


Fonte: O autor (2025)

4.2 Execução e Validação do Pipeline

O fluxo de CI/CD foi acionado automaticamente após um *commit* realizado na *branch* principal (main). A visão geral da execução, com todos os estágios concluídos com sucesso (*passed*), é apresentada na Figura 21. As seções a seguir detalham o resultado individual de cada etapa.

Figura 21 – Visão geral do pipeline executado com sucesso no GitLab CI.

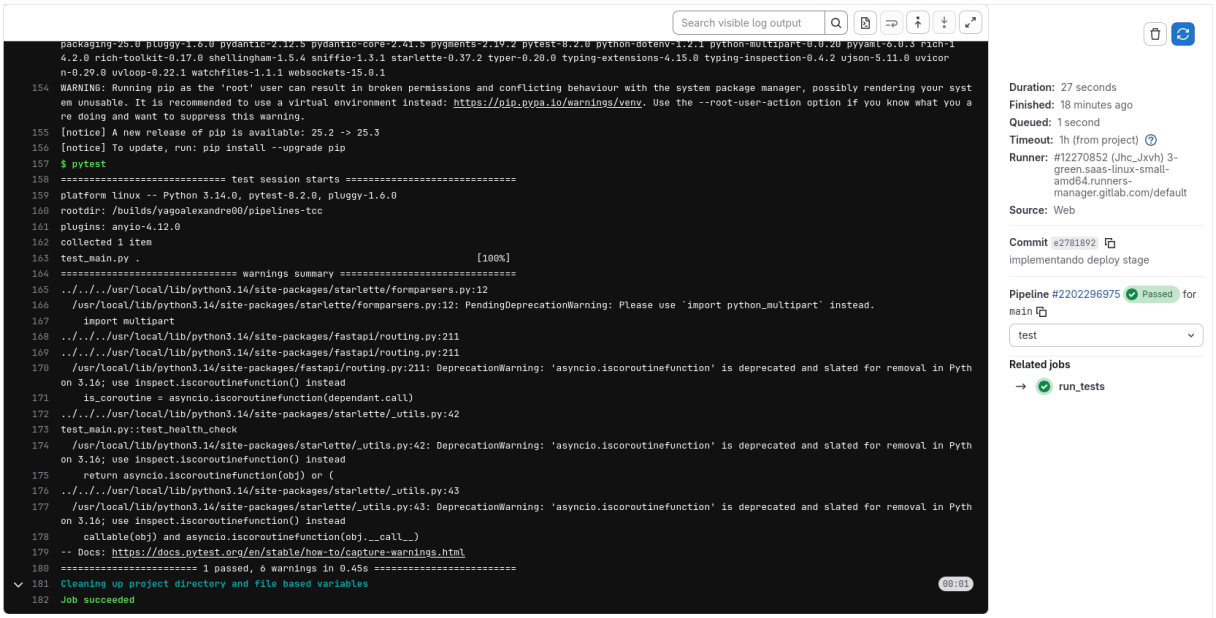


Fonte: O autor (2025)

4.2.1 Estágio 1: Integração e Testes Automatizados

A execução do *pipeline* iniciou-se com o estágio de verificação de código. O *GitLab Runner* realizou o *checkout* do repositório e executou a suíte de testes de integração utilizando o *framework* pytest. Esta etapa validou a resposta do *endpoint* da API, garantindo a integridade funcional da aplicação antes do processo de empacotamento. A Figura 22 apresenta o *log* de execução, evidenciando a aprovação do teste e permitindo a continuidade do fluxo.

Figura 22 – Log de execução do estágio de testes (*test*) no GitLab CI.

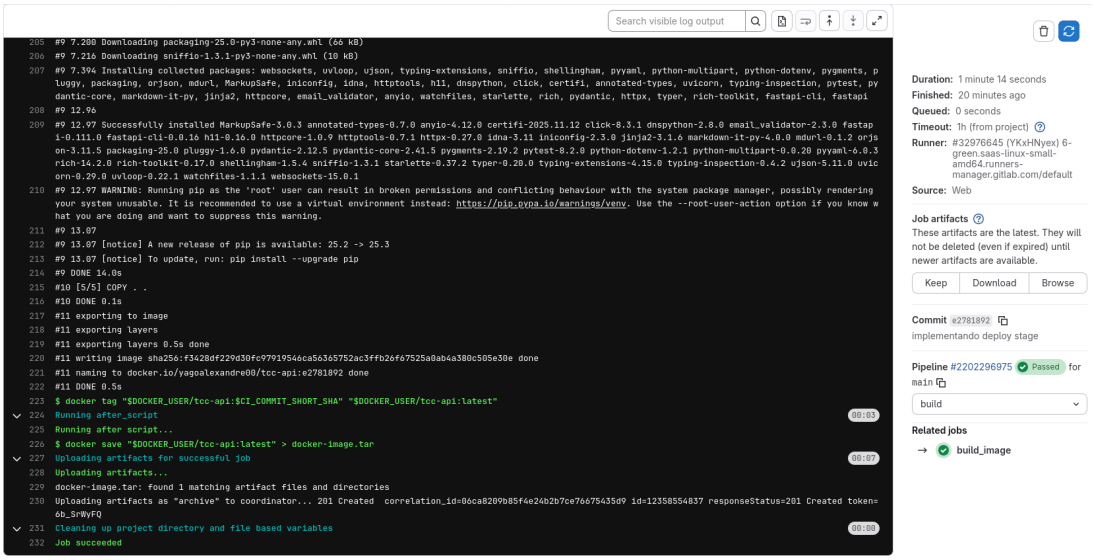


Fonte: O autor (2025)

4.2.2 Estágio 2: Construção da Imagem Docker (*Build*)

Com a aprovação na etapa de testes, o fluxo prosseguiu para a construção do artefato de *software*. Nesta fase, o *GitLab Runner* executou as instruções definidas no *Dockerfile* para gerar a imagem Docker da aplicação. Este processo encapsulou o código-fonte, as bibliotecas e todas as dependências necessárias em uma unidade padronizada, pronta para ser implantada no ambiente de produção. A Figura 23 apresenta o *log* de saída desta etapa, evidenciando a construção bem-sucedida das camadas e o tagueamento final da imagem.

Figura 23 – Log de execução do estágio de construção (*build*) no GitLab CI.



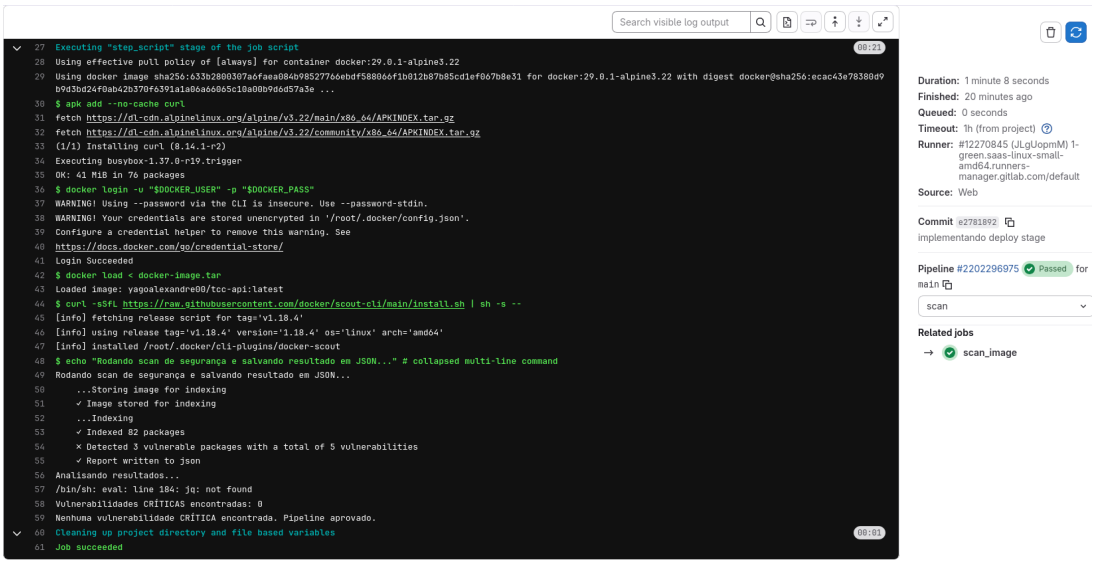
Fonte: O autor (2025)

4.2.3 Estágio 3: Análise de Segurança (*DevSecOps*)

Sequencialmente à construção do artefato (imagem Docker), o *pipeline* executou a etapa de análise de segurança, alinhada às práticas de *DevSecOps*. Utilizando a ferramenta *Docker Scout*, foi realizada uma varredura na imagem recém-criada em busca de vulnerabilidades conhecidas (CVEs).

Conforme observado na Figura 24, o relatório identificou vulnerabilidades de severidade baixa e média. No entanto, como o critério de aceitação (*Quality Gate*) definido no *script* do *pipeline* estabelecia o bloqueio apenas para vulnerabilidades de nível CRÍTICO (*Critical*), o estágio foi considerado bem-sucedido, permitindo o avanço do fluxo automatizado.

Figura 24 – Relatório de análise de vulnerabilidades gerado pelo Docker Scout.



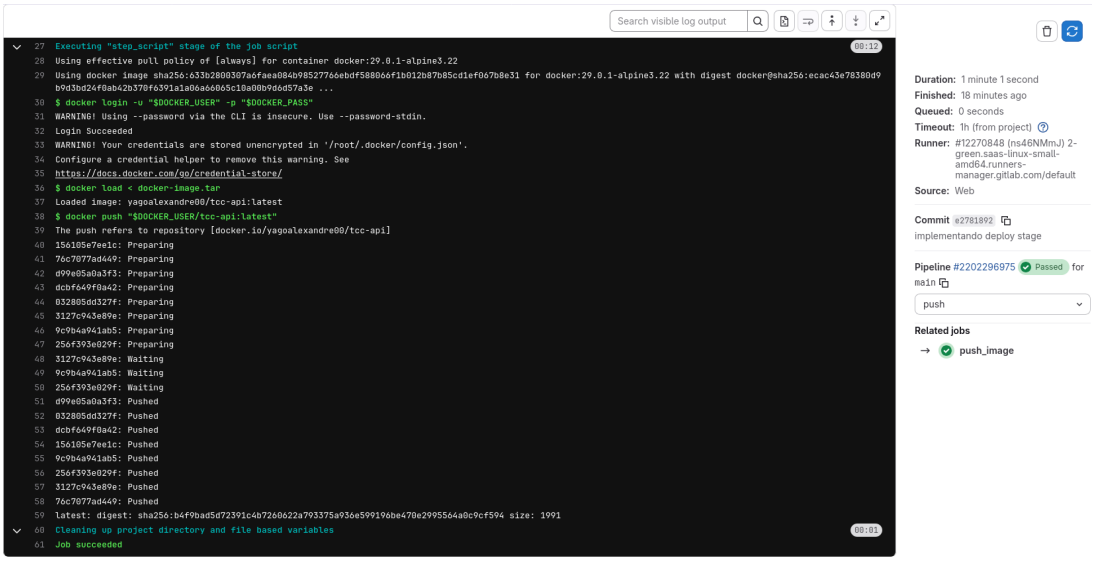
Fonte: O autor (2025)

4.2.4 Estágio 4: Publicação da Imagem (*Push*)

Com a validação do artefato nos quesitos de funcionalidade e segurança, o *pipeline* procedeu com o envio (*push*) da imagem para o registro remoto no Docker Hub. Esta etapa é crucial para garantir que a versão exata do *software* aprovada no fluxo de CI esteja disponível e centralizada para ser baixada pelo servidor de produção.

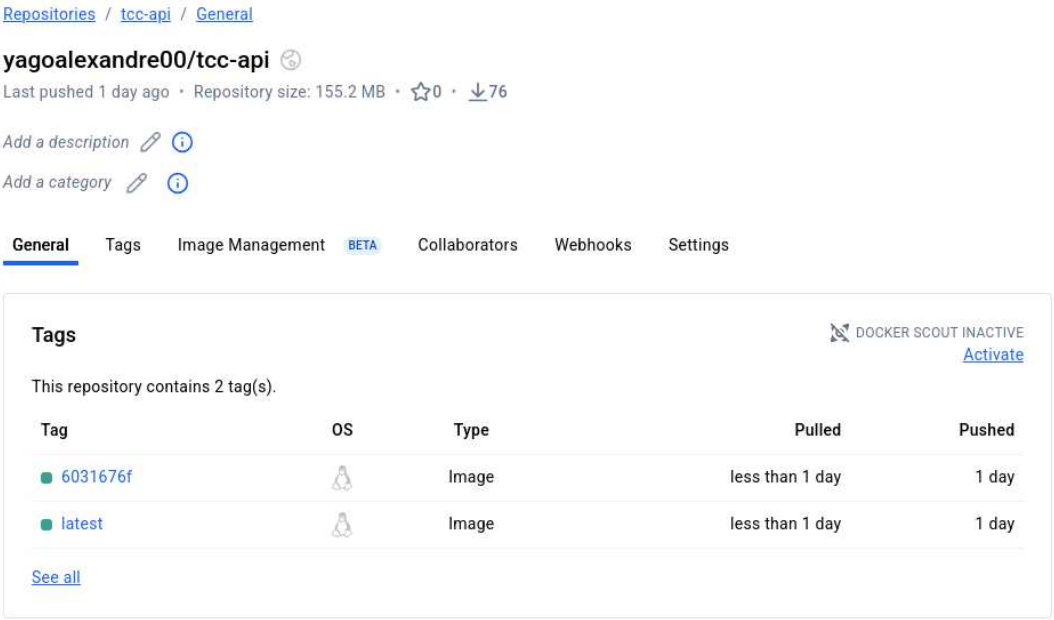
A Figura 25 demonstra o *log* de execução do envio das camadas da imagem para o repositório. Somado a isso, a Figura 26 comprova a efetivação da publicação, exibindo a imagem com a *tag latest* atualizada no painel do Docker Hub.

Figura 25 – Log de execução do envio da imagem (*docker push*) no GitLab CI.



Fonte: O autor (2025)

Figura 26 – Repositório no Docker Hub exibindo a imagem publicada recentemente.



Fonte: O autor (2025)

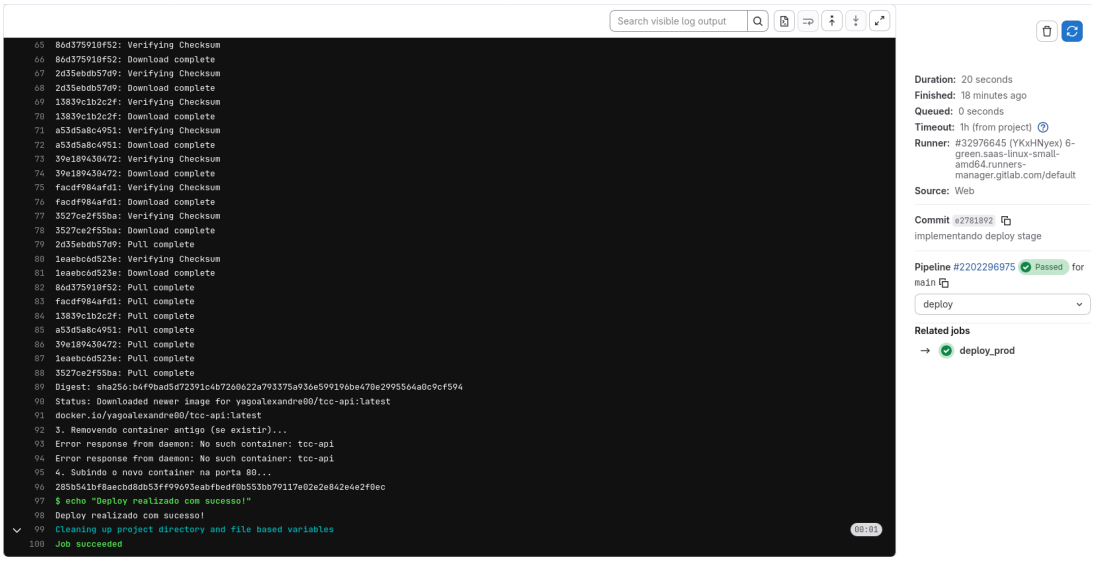
4.2.5 Estágio 5: Implantação Contínua (*Deploy*)

A última etapa do *pipeline* concretizou a Implantação Contínua (*Continuous Deployment*) da aplicação. Mediante uma conexão segura via protocolo SSH, o *GitLab Runner* acessou remotamente a instância EC2 na AWS e orquestrou a atualização do serviço.

O *script* de implantação executou sequencialmente: a autenticação no registro, o *pull* da imagem atualizada, a interrupção do contêiner anterior e, finalmente, a execução da nova versão

mapeada para a porta 80. A Figura 27 evidencia o *log* de saída deste processo, confirmando a substituição bem-sucedida do contêiner em produção sem intervenção humana direta.

Figura 27 – Log de execução do estágio de implantação (*deploy*) via SSH na AWS.



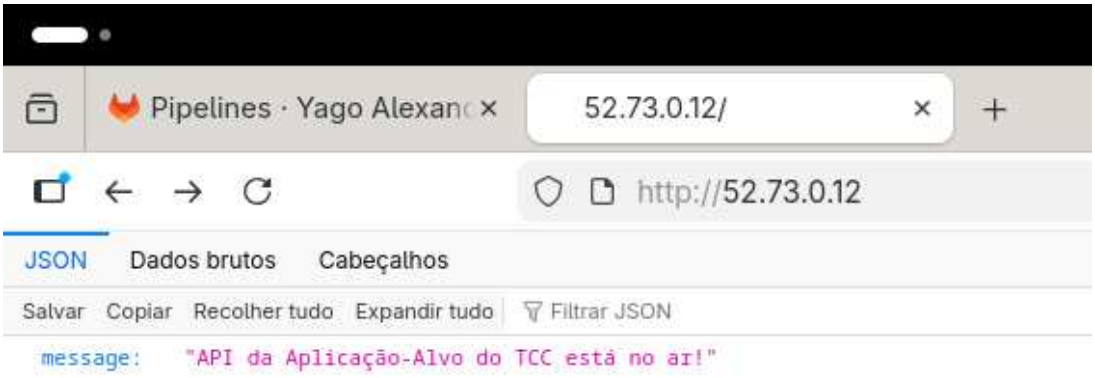
Fonte: O autor (2025)

4.3 Verificação Funcional em Produção

A validação funcional definitiva consistiu no acesso externo à aplicação, comprovando sua disponibilidade pública na *internet*. Visto que o *Security Group* foi configurado para permitir tráfego HTTP (porta 80) de qualquer origem global (0.0.0.0/0), foi possível acessar o serviço diretamente através do endereço IP público da instância AWS.

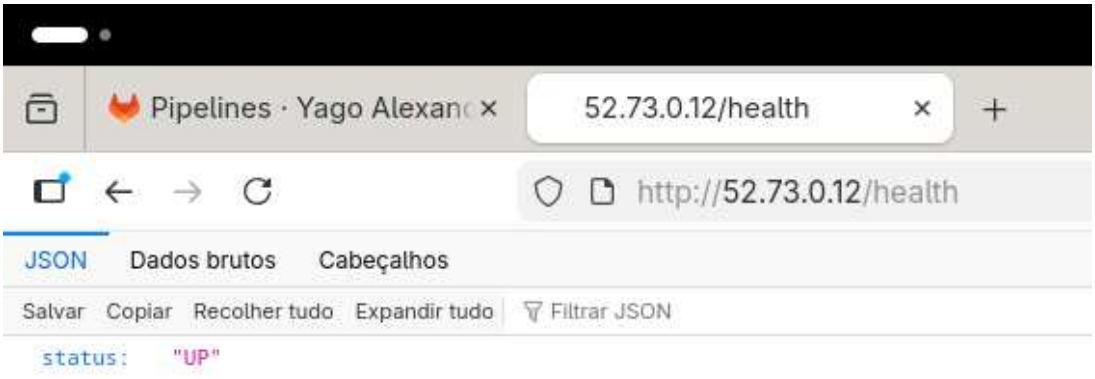
O teste foi realizado em duas etapas. Inicialmente, acessou-se a rota raiz ("/"), que retornou a mensagem de boas-vindas da API, confirmando que o servidor *web* está ativo e recebendo requisições (Figura 28). Em seguida, requisitou-se o *endpoint* de verificação de saúde ("/health"), obtendo o retorno do *payload* JSON `{"status": "UP"}` (Figura 29). Estes resultados confirmam que o contêiner foi implantado corretamente e que a lógica da aplicação está operante, validando o sucesso integral do *pipeline* de CI/CD.

Figura 28 – Validação funcional: Acesso à rota raiz ("/") via navegador.



Fonte: O autor (2025)

Figura 29 – Validação funcional: Acesso ao *endpoint* "/health"via navegador.



Fonte: O autor (2025)

4.4 Análise e Discussão dos Resultados

A Tabela 1 sintetiza os ganhos obtidos com a implementação do *pipeline*, comparando as características de um fluxo de *deploy* manual tradicional com a solução automatizada desenvolvida neste estudo de caso.

Tabela 1 – Comparativo: Cenário Tradicional vs. Cenário Automatizado

Critério de Análise	**Cenário Tradicional (Manual)**	**Cenário Automatizado (Proposto)**
Tempo de Entrega (*Lead Time*)	**Alto**. O processo depende de disponibilidade humana, cópia manual de arquivos e configuração de servidor, podendo levar vários minutos ou até horas dependendo do cenário.	**Reduzido.** O tempo entre o *commit* e o *deploy* foi de apenas alguns minutos (tempo de execução do *pipeline*), sem grandes esperas ou mudanças no padrão.
Paridade de Ambientes	**Baixa**. Risco de inconsistência ambiental. Divergência de bibliotecas entre *dev* e *prod* ("funciona na minha máquina").	**Garantida.** O uso de contêineres Docker assegura que a imagem testada é binariamente idêntica à implantada.
Segurança da Aplicação	**Reativa**. Vulnerabilidades geralmente são descobertas apenas em produção ou em auditorias tardias.	**Proativa (*Shift-Left*).** O estágio de *scan* bloqueia vulnerabilidades críticas antes mesmo da imagem ser gerada.
Confiabilidade do *Deploy*	**Baixa**. Processos manuais (FTP, comandos no terminal) são propensos a erro humano e esquecimento de etapas.	**Alta.** O processo é determinístico e repetível, eliminando a variabilidade da intervenção humana.
Rastreabilidade e Auditoria	**Nula ou Difícil**. Alterações via SSH/FTP não geram histórico centralizado. Difícil identificar quem alterou o quê e quando.	**Total.** Todo *deploy* possui registro de autoria, data e está vinculado a um *commit* específico e ao *log* do *job* no GitLab.

Fonte: O autor (2025)

Com base nos dados apresentados na Tabela 1 e nas evidências coletadas durante a execução experimental, é possível realizar uma análise crítica sobre quatro pilares fundamentais: velocidade, paridade de ambientes, segurança e rastreabilidade.

4.4.1 Melhoria na Velocidade e Eficiência (*Lead Time*)

O primeiro impacto observável foi a otimização do tempo de entrega (*Lead Time for Changes*). Segundo Forsgren, Humble e Kim [11], esta métrica do DORA (*DevOps Research and Assessment*) é fundamental para correlacionar a frequência de *deploy* com a estabilidade organizacional.

Para quantificar este ganho, foi realizada uma simulação controlada do processo manual em três execuções distintas. Para garantir a paridade exata com o pipeline automatizado, o fluxo manual incluiu obrigatoriamente as cinco etapas críticas do ciclo de vida, incluindo a verificação de segurança (*Docker Scout*), frequentemente negligenciada em processos manuais devido à pressão por prazos.

Os tempos aferidos estão detalhados na Tabela 2.

Tabela 2 – Cronometragem do Processo Manual (Ciclo Completo)

Etapas do Processo	**Teste #1**	**Teste #2**	**Teste #3**	**Média**
1. Execução de Testes *(pytest local)*	00m 08s	00m 14s	00m 12s	**00m 11s**
2. Build da Imagem *(docker build)*	00m 40s	00m 37s	00m 38s	**00m 38s**
3. Scan de Segurança *(docker scout)*	02m 23s	00m 55s	01m 03s	**01m 27s**
4. Push para Registry *(docker push)*	01m 54s	01m 42s	01m 57s	**01m 51s**
5. Deploy na Nuvem *(SSH + Pull + Run)*	02m 22s	00m 55s	01m 15s	**01m 31s**
TEMPO TOTAL	**07m 28s**	**04m 24s**	**05m 05s**	**05m 39s**

Fonte: O autor (2025)

4.4.1.1 Análise de Desempenho e Variabilidade:

A análise comparativa entre as médias das execuções manuais e automatizadas revela ganhos tanto na eficiência temporal quanto na previsibilidade do processo.

Em termos de velocidade média, o processo manual exigiu 05 minutos e 39 segundos para completar o ciclo. Em contrapartida, a média de três execuções do pipeline automatizado — cujos registros de tempo podem ser visualizados na Figura 30 — resultou em 04 minutos e 30 segundos. Isso representa uma redução direta de aproximadamente 20,35% no tempo total de entrega.

Contudo, o indicador mais crítico observado não foi a velocidade, mas a estabilidade. Enquanto o processo manual apresentou uma alta variabilidade, com uma amplitude de mais de 3 minutos entre o teste mais rápido e o mais lento (devido a fatores humanos como digitação e leitura de logs), o processo automatizado manteve-se consistente, com uma variação máxima de apenas 32 segundos entre as execuções, atribuída a oscilações normais de latência de rede na nuvem AWS.

Essa consistência valida a hipótese de que a automação transforma o *deploy* em um processo determinístico e auditável, eliminando a incerteza operacional inerente à intervenção humana.

Figura 30 – Evidências de Execução do Pipeline (Tempos Registrados)

Fonte: O autor (2025)

4.4.2 Garantia de Paridade e Imutabilidade

A utilização do Docker eliminou o problema clássico de inconsistência ambiental, alinhando o projeto ao princípio de "Paridade entre Desenvolvimento e Produção"defendido por Kim et al. [19]. Ao encapsular a aplicação e suas dependências em uma imagem Docker única no Estágio 2, garantiu-se que o artefato validado no Estágio 1 (Testes) fosse binariamente idêntico ao implantado no Estágio 5 (Produção).

Essa abordagem implementa na prática o conceito de *Immutable Infrastructure* (Infraestrutura Imutável). Segundo Morris [23], essa prática consiste em substituir componentes inteiros em vez de alterá-los, mitigando riscos de "configuração à deriva"(*configuration drift*). Adicionalmente, Matthias e Kane [22] reforçam que o isolamento provido pelos contêineres é superior ao de máquinas virtuais para empacotamento de aplicações, validando a hipótese de que a conteinerização é a solução definitiva para a estabilidade de *releases*.

4.4.3 Segurança Proativa (Abordagem *Shift-Left*)

A inclusão do Estágio 3 (Análise de Segurança) materializou o conceito de *Shift-Left Security*. Segundo a Fortinet [12], essa abordagem propõe mover a segurança do final para o início do ciclo de desenvolvimento, permitindo a detecção de falhas antes que elas se tornem onerosas.

Essa prática alinha-se aos princípios de *DevSecOps* descritos por Bass, Weber e Zhu [6], que tratam a segurança como um atributo de arquitetura e não apenas uma verificação final.

Diferente do cenário tradicional, onde a segurança atua como um gargalo reativo, o *pipeline* implementado funcionou como um *Quality Gate* automatizado. A capacidade de identificar vulnerabilidades CVE (*Common Vulnerabilities and Exposures*) durante o *build* impede a propagação de código inseguro, alinhando o projeto às trilhas modernas de aprendizado em *DevSecOps* [26].

4.4.4 Rastreabilidade e Auditoria de Mudanças

Um benefício adicional, frequentemente subestimado em *deploys* manuais, é a capacidade de rastreabilidade (*traceability*) proporcionada pelo *pipeline*. Em um cenário tradicional, alterações realizadas diretamente no servidor via terminal raramente deixam registros claros, criando uma "caixa preta"operacional.

Com a implementação do CI/CD, cada implantação tornou-se um evento auditável. O GitLab CI/CD vincula inequivocamente a versão em produção ao *hash* do *commit* no Git. Segundo Sato [25], a capacidade de saber exatamente o que está rodando em produção é essencial para a entrega confiável de *software*. Além disso, Kim et al. [19] argumentam que essa prática facilita a resolução de incidentes (*troubleshooting*) e atende a requisitos rigorosos de governança e conformidade, transformando o processo de *release* em um fluxo transparente.

5 CONSIDERAÇÕES FINAIS

O presente trabalho abordou os desafios inerentes ao ciclo de vida de desenvolvimento de *software*, com ênfase na transição crítica entre as equipes de desenvolvimento e operação. A problemática central, identificada por Kawaguchi [17] como o "Muro da Confusão", demonstrou como processos manuais e a falta de padronização geram ineficiências operacionais e riscos elevados de falha. Através de uma pesquisa aplicada de natureza experimental, buscou-se demonstrar que a integração de contêineres Docker e *pipelines* de CI/CD não constitui apenas uma atualização tecnológica, mas uma mudança fundamental na arquitetura de entrega de *software*, alinhada aos princípios de *DevOps* descritos por Bass, Weber e Zhu [6].

A execução do estudo de caso, materializada na implantação automatizada de uma API *Python* (*FastAPI*) em ambiente de nuvem (AWS), confirmou a hipótese de que a automação é o vetor principal para garantir agilidade, segurança e confiabilidade, respondendo às demandas de negócios modernos conforme preconizado por Sato [25].

5.1 Síntese dos Resultados e Objetivos Alcançados

Retomando o objetivo geral deste trabalho — implementar um *pipeline* de CI/CD com Docker para automatizar o ciclo de uma aplicação *web* e avaliar o impacto na eficiência —, conclui-se que o mesmo foi plenamente atingido. A validação funcional e a análise dos tempos de execução comprovaram a eficácia da solução proposta frente ao cenário manual tradicional, corroborando a visão de Humble e Farley [15] sobre a necessidade de entregas de *software* rápidas e confiáveis.

No que tange aos objetivos específicos definidos na Seção 1.2.1, as seguintes conclusões foram obtidas:

- **Padronização e Paridade de Ambientes (Objetivo 1):** A utilização do Docker mostrou-se eficaz na eliminação de inconsistências ambientais. Ao empacotar a aplicação e suas dependências em uma imagem imutável, garantiu-se que o comportamento observado nos testes locais fosse replicado fielmente no servidor de produção. Isso valida a vantagem dos contêineres sobre máquinas virtuais tradicionais citada pela 4Linux [1] e por Matthias e Kane [22], assegurando a portabilidade e eliminando o problema do "funciona na minha máquina".

- **Implementação e Análise do Pipeline (Objetivo 2):** O processo automatizado foi implementado com êxito em cinco estágios distintos. A análise dos resultados comprovou uma otimização de **20,35%** no tempo de entrega (*Lead Time*), validando a eficiência apontada por Forsgren, Humble e Kim [11]. Mais relevante que o ganho de velocidade,

o *pipeline* transformou um processo manual instável e sujeito a variações humanas em um fluxo determinístico e previsível, instituindo a rastreabilidade total das mudanças e a garantia de segurança (*Shift-Left*), em conformidade com os conceitos de Entrega Contínua definidos pela Atlassian [2] e AWS [5].

- **Eliminação da Variabilidade e Erro Humano (Objetivo 3):** A investigação técnica demonstrou que a substituição de intervenções manuais por *scripts* declarativos eliminou a variância operacional observada no cenário tradicional. Enquanto o processo manual apresentou oscilações significativas de tempo e risco de falha humana, o *pipeline* automatizado garantiu a execuções determinísticas, assegurando a previsibilidade e a estabilidade do processo de entrega conforme proposto.

Adicionalmente, um resultado de grande relevância foi a validação da segurança proativa. A integração do estágio de análise de vulnerabilidades materializou o conceito de *Shift-Left Security* definido pela Fortinet [12], elevando a maturidade de segurança do projeto para um modelo de *DevSecOps*.

5.2 Limitações do Trabalho

Apesar do êxito na validação do protótipo, este trabalho apresenta limitações decorrentes do escopo acadêmico e do ambiente experimental controlado:

- **Complexidade da Aplicação:** O estudo utilizou uma API minimalista. Em cenários corporativos reais, aplicações possuem dependências de estado (bancos de dados, *caches*) que exigiriam estratégias mais complexas de persistência de dados e *backups* durante o *deploy*, temas abordados como desafios de orquestração por Romero [24].
- **Escalabilidade da Infraestrutura:** A validação restringiu-se a uma única instância EC2 (*single node*). Não foram abordados mecanismos de alta disponibilidade, como balanceadores de carga ou grupos de auto-escalonamento, essenciais para sistemas de alto tráfego na nuvem AWS [3].
- **Métricas de Longo Prazo:** Embora o *Lead Time* tenha sido medido pontualmente, a avaliação contínua de outras métricas do DORA (como Tempo de Restauração de Serviço e Taxa de Falha de Mudança) sugeridas por Hall [14] demandaria um período de observação mais extenso em um ambiente produtivo real.

5.3 Trabalhos Futuros

Como desdobramento desta pesquisa, sugere-se a exploração de temas que expandam a robustez da arquitetura aqui proposta:

1. **Orquestração de Contêineres:** Evoluir a implantação de um servidor Docker isolado para um orquestrador como Docker Swarm ou Kubernetes. Isso permitiria o gerenciamento avançado de réplicas e recuperação automática (*self-healing*), ampliando a confiabilidade discutida por Matthias e Kane [22].

2. **Infraestrutura como Código (IaC) Avançada:** Aprofundar o uso de ferramentas para gerenciar a infraestrutura de forma imutável, prevenindo a deriva de configuração (*configuration drift*) conforme alertado por Morris [23].

3. **Estratégias Avançadas de Deploy:** Implementar técnicas como *Blue-Green Deployment* ou *Canary Releases*, que permitem testar novas versões com uma parcela reduzida de usuários, minimizando riscos de impacto no negócio descritos por Humble e Farley [15].

Em suma, este estudo reforça que a adoção de práticas *DevOps*, suportada por ferramentas como Docker e CI/CD, é um requisito indispensável para a engenharia de *software* contemporânea, provendo a base necessária para entregas ágeis, seguras e alinhadas aos objetivos de negócio.

Referências

[1] 4LINUX. **Diferença entre Containers e Máquinas Virtuais**. 4Linux, [s.d.]. Disponível em: https://www.4linux.com.br/diferenca-entre-containers-e-maquinas-virtuais. Acesso em: 14 out. 2025.

[2] ATLASSIAN. **What is continuous delivery?**. Atlassian, [s.d.]. Disponível em: https://www.atlassian.com/continuous-delivery. Acesso em: 13 out. 2025.

[3] AWS. **O que é o DevOps?**. [S. l.]: AWS, 2022. Disponível em: https://aws.amazon.com/pt/devops/what-is-devops/. Acesso em: 07 jun. 2022.

[4] AWS. **O que é Integração Contínua?**. Amazon Web Services, [s.d.]. Disponível em: https://aws.amazon.com/pt/devops/continuous-integration/. Acesso em: 13 out. 2025.

[5] AWS. **O que é Entrega Contínua?**. Amazon Web Services, [s.d.]. Disponível em: https://aws.amazon.com/pt/devops/continuous-delivery/. Acesso em: 13 out. 2025.

[6] BASS, Len; WEBER, Ingo; ZHU, Liming. **DevOps: A Software Architect's Perspective**. Boston: Addison-Wesley Professional, 2015.

[7] BUCHANAN, Ian. **História do DevOps: Como as equipes de desenvolvimento e operações se uniram para solucionar disfunções no setor**. Atlassian, [s.d.]. Disponível em: https://www.atlassian.com/br/devops/what-is-devops/history-of-devops. Acesso em: 06 out. 2025.

[8] DEVOPEDIA. **DevOps**. Devopedia, 2022. Disponível em: https://devopedia.org/devops. Acesso em: 13 out. 2025.

[9] DOCKER. **Docker overview**. Docker Documentation, [s.d.]. Disponível em: https://docs.docker.com/get-started/docker-overview/. Acesso em: 15 out. 2025.

[10] DOCKER, Inc. **Docker: Develop Faster, Run Anywhere**. Docker, Inc., [s.d.]. Disponível em: https://www.docker.com/. Acesso em: 14 out. 2025.

[11] FORSGREN, Nicole; HUMBLE, Jez; KIM, Gene. **Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations**. Portland, OR: IT Revolution Press, 2018.

[12] FORTINET. **O que é segurança Shift Left?**. [S. l.]: Fortinet, 2025. Disponível em: https://www.fortinet.com/br/resources/cyberglossary/shift-left-security. Acesso em: 16 dez. 2025.

[13] GIL, Antonio Carlos. *Como Elaborar Projetos de Pesquisa*. 6. ed. São Paulo: Atlas, 2019.

[14] HALL, Tom. **DORA metrics: How to measure Open DevOps success**. Atlassian, 2023. Disponível em: `https://www.atlassian.com/devops/frameworks/dora-metrics`. Acesso em: 13 out. 2025.

[15] HUMBLE, Jez; FARLEY, David. **Entrega Contínua: Como Entregar Software de Forma Rápida e Confiável**. Porto Alegre: Bookman Editora, 2014.

[16] HUMBLE, Jez. **What is Continuous Delivery?**. continuousdelivery.com, [s.d.]. Disponível em: `https://continuousdelivery.com/`. Acesso em: 13 out. 2025.

[17] KAWAGUCHI, Stephen. **The Wall of Confusion - DevOps Essentials**. Level Up Coding (Medium), 2020. Disponível em: `https://levelup.gitconnected.com/the-wall-of-confusion-623057a4dd26`. Acesso em: 06 out. 2025.

[18] KIM, Gene; BEHR, Kevin; SPAFFORD, George. **O Projeto Fênix: Um Romance sobre TI, DevOps e Sobre Ajudar sua Empresa a Vencer**. Rio de Janeiro: Alta Books, 2014.

[19] KIM, Gene; HUMBLE, Jez; DEBOIS, Patrick; WILLIS, John. **The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations**. Portland, OR: IT Revolution Press, 2016.

[20] KIM, Gene. **The Three Ways: The Principles Underpinning DevOps**. IT Revolution, 2012. Disponível em: `https://itrevolution.com/articles/the-three-ways-principles-underpinning-devops/`. Acesso em: 10 out. 2025.

[21] KOHGADAI, Ajmal. **Building a DevSecOps culture and shifting security left**. Red Hat Blog, 2021. Disponível em: `https://www.redhat.com/en/blog/building-devsecops-culture-and-shifting-security-left`. Acesso em: 28 jan. 2026.

[22] MATTHIAS, Karl; KANE, Sean P. **Docker: Up & Running: Shipping Reliable Containers in Production**. 2. ed. Sebastopol, CA: O'Reilly Media, 2018.

[23] MORRIS, Kief. **Infrastructure as Code**: Dynamic Systems for the Cloud Age. 2. ed. Sebastopol: O'Reilly Media, 2020.

[24] ROMERO, Daniel. **Containers com Docker: Do desenvolvimento à produção**. São Paulo: Casa do Código, 2015.

[25] SATO, Danilo. **DevOps na Prática: entrega de software confiável e automatizada**. São Paulo: Casa do Código, 2014.

[26] TRYHACKME. **DevSecOps Learning Path**. [S. l.]: TryHackMe, [2024]. Disponível em: `https://tryhackme.com/path/devsecops`. Acesso em: 16 set. 2025.

[27] UNITY. **O que é CI/CD?**. Unity, [s.d.]. Disponível em: `https://unity.com/pt/topics/what-is-ci-cd`. Acesso em: 13 out. 2025.