Instituto Federal de Educação Ciência e Tecnologia do Ceará
Bacharelado em Engenharia de Computação
Disciplina: Engenharia de Software

Alana Silva Sales
Soraia Freire Batista
Raynara Maria Aurelio Coelho
Marcos Martenier Santos Oliveira
Kelly Letícia Nascimento de Morais

Relatório de Pesquisa e Análise: DevOps e Práticas de Integração e Entrega Contínua

Fortaleza
Junho de 2025

Alana Silva Sales
Soraia Freire Batista
Raynara Maria Aurelio Coelho
Marcos Martenier Santos Oliveira
Kelly Letícia Nascimento de Morais

Relatório de Pesquisa e Análise: DevOps e Práticas de Integração e Entrega Contínua

Trabalho apresentado ao curso de Engenharia de Computação do Instituto Federal de Educação Ciência e Tecnologia do Ceará como requisito parcial para obtenção de nota na disciplina de Engenharia de Software.

Professor responsável: César Olavo

Fortaleza

2025

Sumário

1. INTRODUÇÃO

Este relatório tem como objetivo explorar de forma aprofundada os conceitos, práticas e ferramentas relacionados à abordagem *DevOps* [2], [3], com ênfase especial nos processos de *Integração Contínua* (*CI*), *Entrega Contínua* (*CD*) [1] e *Implantação Contínua* (*CD*). Com o avanço das metodologias ágeis e a crescente necessidade por ciclos de entrega mais curtos, a adoção de práticas *DevOps* [2], [3] tem se mostrado essencial para garantir qualidade, estabilidade e agilidade no desenvolvimento de *software*.

A primeira parte deste relatório aborda a base teórica dos conceitos, suas aplicações e principais desafios. Na segunda parte, será apresentado um projeto prático com a implementação de um *pipeline CI/CD* [1], [3] utilizando uma das ferramentas analisadas, a ser definida conforme as especificidades do projeto desenvolvido.

2. INTRODUÇÃO AO DEVOPS

2.1 Origem e Contexto Histórico do DevOps

O termo *DevOps* [2], [3] surgiu da necessidade de superar os atritos provocados pela separação tradicional entre as equipes de desenvolvimento (*Dev*) e operações (*Ops*) de *software*. Historicamente, a área de desenvolvimento buscava entregar mudanças de forma rápida e constante, enquanto operações priorizavam estabilidade e segurança, gerando conflitos. Esse modelo, muitas vezes chamado de "*water-scrum-fall*" [12], levava a ciclos de entrega lentos, comunicação truncada e ambientes de produção instáveis.

O termo foi popularizado por Patrick Debois em 2009 durante a primeira conferência *DevOpsDays*, e desde então tem evoluído como uma prática cultural, técnica e organizacional [2].

2.2 O que é *DevOps*

DevOps [2], [3] é, antes de tudo, uma mudança de paradigma, uma abordagem cultural e prática que une princípios de *lean*, ágil, automação e monitoramento contínuo para integrar desenvolvimento e operações com o objetivo de entregar *software* com mais rapidez, qualidade e confiabilidade.

Segundo Bass et al. (2015) [3], *DevOps* pode ser entendido como "um conjunto de práticas destinadas a reduzir o tempo entre uma mudança ser feita em um sistema e essa mudança estar em produção, ao mesmo tempo garantindo alta qualidade".

2.3 Pilares da Cultura *DevOps*: O Modelo *CALMS*

O modelo *CALMS*, proposto por Jez Humble e colaboradores [2], [1], resume os cinco pilares fundamentais da cultura *DevOps*:

- ***Cultura (Culture)***: Enfatiza a colaboração entre equipes multidisciplinares, a transparência, a confiança e uma mentalidade de melhoria contínua.
- ***Automação (Automation)***: Abrange a automação de testes, *builds*, *deploys*, provisionamento de infraestrutura e monitoramento.
- ***Lean***: Foca na eliminação de desperdícios, na criação de fluxos de trabalho enxutos e em entregas iterativas.
- ***Medição (Measurement)***: Refere-se ao uso de métricas (como *lead time*, *change failure rate* e *MTTR*) para embasar decisões e otimizar processos.
- ***Compartilhamento (Sharing)***: Promove a disseminação de conhecimento e boas práticas entre todos os membros da equipe e da organização.

2.4 Benefícios da Adoção de *DevOps*

A incorporação de *DevOps* [2], [3] proporciona ganhos significativos em diversas áreas da engenharia de *software*:

- **Redução do *lead time*:** o tempo entre a concepção da funcionalidade e sua entrega em produção.
- **Maior frequência de *deploys*:** empresas como Amazon e Netflix realizam centenas de *deploys* por dia.
- **Menor taxa de falhas na produção:** práticas de teste automatizado e *rollback* reduzem riscos.
- ***Feedback* contínuo e melhoria incremental:** integração com monitoramento permite ciclos de melhoria baseados em dados reais.

Um estudo do DORA (*DevOps Research and Assessment*) mostra que equipes de alta performance que adotam *DevOps* entregam *software* 46x mais frequentemente, com 2.604x menor tempo de recuperação (*MTTR*) e 7x menor taxa de falhas em produção [13]. O que são dados relevantes e que instigam a adoção da prática e a familiarização com os recursos que a abordagem propõe.

2.5 Desafios e Resistências à Implementação

Apesar dos benefícios evidentes, a implementação de *DevOps* [2], [3] enfrenta diversas barreiras, muitas delas de natureza organizacional e técnica:

- **Mudança cultural:** Abandonar a mentalidade tradicional de “isso não é meu problema” e promover a colaboração entre equipes de desenvolvimento e operações exige um forte comprometimento da liderança e uma reestruturação de processos e mentalidades.
- **Legado tecnológico:** Sistemas antigos, monolíticos e mal documentados frequentemente dificultam a automação e a integração contínua, tornando a transição para práticas *DevOps* mais complexa.
- **Falta de capacitação:** Tanto os desenvolvedores (*devs*) quanto os profissionais de operações (*ops*) precisam dominar novas ferramentas, tecnologias de *pipelines* e conceitos como *infraestrutura como código*, demandando investimentos significativos em treinamento e desenvolvimento. Adicionando assim um investimento de tempo prévio ao desenvolvimento para adaptação da equipe.
- **Sobrecarga de ferramentas**: O "*tooling overload*", ou a proliferação excessiva de ferramentas, pode gerar mais complexidade do que benefícios se não for bem gerenciado, exigindo um planejamento cuidadoso e a escolha estratégica das soluções tecnológicas.

2.6 Ecossistema de Ferramentas

Embora *DevOps* [2], [3] não se resuma a ferramentas, elas são a base que torna possível operacionalizar os princípios [2]. O ecossistema pode ser dividido em várias camadas:

- **Controle de versão:** Ferramentas como *Git* [14], *GitHub* [5], *GitLab* [6] e *Bitbucket* [15] são essenciais para gerenciar o histórico de código e facilitar a colaboração entre os desenvolvedores.
- **Integração e entrega contínuas:** Plataformas como *Jenkins* [4], *GitLab CI/CD* [6], *GitHub Actions* [5] e *CircleCI*[16] automatizam os processos de *build*, teste e *deploy* [1].
- **Gerência de configuração:** Soluções como *Ansible* [17], *Puppet* [18] e *Chef* [19] permitem automatizar o provisionamento e a configuração de infraestrutura.
- **Conteinerização e orquestração:** *Docker* [8] e *Podman* [20] são utilizados para empacotar aplicações em *containers*, enquanto *Kubernetes* [21] orquestra o *deploy* e a gestão desses *containers* em larga escala.
- **Monitoramento e observabilidade:** Ferramentas como *Prometheus* [22], *Grafana* [23], *ELK Stack* (*Elasticsearch*, *Logstash*, *Kibana*) [24] e *Datadog* [25] fornecem visibilidade sobre o desempenho e a saúde dos sistemas em produção.
- **Segurança (*DevSecOps*)**: Ferramentas como *SonarQube* [26], *Snyk* [27] e *Trivy* [28] integram práticas de segurança diretamente nos *pipelines* de *CI/CD*, garantindo que vulnerabilidades sejam identificadas e corrigidas precocemente.

A escolha correta dessas ferramentas depende do contexto técnico, do tamanho da equipe e da maturidade do projeto.

3. INTEGRAÇÃO CONTÍNUA (*CI*)

3.1 Conceito e Fundamentos

A Integração Contínua (*CI*) [1] é uma prática essencial dentro do *DevOps* [2], [3] que consiste na integração frequente do código desenvolvido por diferentes membros da equipe em um repositório central. A cada integração, executa-se uma cadeia automatizada de testes, *builds* e verificações com o objetivo de detectar falhas o mais cedo possível. A ideia central da *CI* é minimizar a “integração de última hora”, que historicamente causa conflitos, *bugs* e retrabalho.

Essa prática se baseia no princípio de que quanto mais cedo um erro for detectado, menor será seu custo para correção. A *CI* contribui diretamente para a

redução do tempo de entrega, aumento da qualidade do *software* e melhoria na comunicação entre desenvolvedores.

3.2 Princípios Fundamentais da Integração Contínua

Segundo Martin Fowler (2006) [29], um dos primeiros defensores da *CI*, algumas das boas práticas fundamentais incluem:

- **Integração diária**: Desenvolvedores devem integrar mudanças no código ao menos uma vez ao dia.
- **Automação total do build**: A compilação e empacotamento do *software* deve ocorrer de forma automatizada.
- **Testes automatizados**: Toda alteração deve ser validada automaticamente por uma *suíte* de testes.
- ***Build* rápido e confiável**: *Builds* devem ser concluídos rapidamente e falhar apenas quando há problemas reais.
- **Ambiente limpo e isolado**: O ambiente de *build* deve ser idêntico ao ambiente de produção.

3.3 *CI* na Prática: Do *Commit* ao *Feedback*

O ciclo prático da *Integração Contínua* [1] pode ser resumido nas seguintes etapas:

1. O desenvolvedor realiza alterações no código-fonte.
2. As alterações são "*commitadas*" e enviadas ao repositório (ex: *Git* [14]).
3. O sistema de *CI* é acionado automaticamente.
4. Um processo de *build* compila o código e executa *scripts*.
5. A *suíte* de testes é executada [9].
6. O resultado é disponibilizado imediatamente para os desenvolvedores.

Esse ciclo, idealmente, não deve ultrapassar 10 a 15 minutos, permitindo ciclos de *feedback* extremamente rápidos.

3.4 Benefícios Estratégicos da *CI*

A adoção da *CI* [1] proporciona benefícios em três dimensões principais:

- **Técnica**:
    - Detecção precoce de erros e conflitos de *merge*.
    - Código sempre em estado funcional.
    - Documentação viva da evolução do projeto.

- **Organizacional**:
    - Comunicação mais fluida entre os membros da equipe.
    - Cultura de responsabilidade coletiva sobre o código.
    - Redução do retrabalho e aumento da produtividade.
- **Negócio**:
    - Entregas mais rápidas e confiáveis ao cliente.
    - Capacidade de adaptação a mudanças com agilidade.
    - Previsibilidade e menor custo de manutenção.

Esses benefícios são amplamente discutidos na literatura sobre *Continuous Integration*, incluindo trabalhos que abordam a detecção precoce de problemas, a melhoria da colaboração e a aceleração dos ciclos de *feedback* [30], [31], [32].

3.5 Desafios na Implementação

Apesar de seus inegáveis benefícios, a *Integração Contínua* (*CI*) [1] apresenta obstáculos comuns em sua implementação:

- **Resistência à mudança de cultura**: Times acostumados a ciclos de desenvolvimento longos e integrações infrequentes podem resistir ao modelo incremental de *CI*, que exige colaboração constante e mentalidade de "código sempre funcional".
- **Testes frágeis ou inexistentes**: Sem uma suíte de testes automatizados robusta e confiável, a *CI* perde sua efetividade central. Testes que falham intermitentemente ou não cobrem adequadamente as funcionalidades podem minar a confiança no processo.
- **Builds lentos**: *Builds* que demoram excessivamente prejudicam o ciclo de *feedback* rápido, desmotivando a equipe e reduzindo a agilidade que a *CI* promete. O objetivo é que os *builds* sejam concluídos em poucos minutos [29].
- **Dificuldade na padronização de ambientes**: Diferenças significativas entre os ambientes de desenvolvimento local, *staging* e produção podem levar a falhas não detectadas durante a *CI*, que só se manifestam em estágios posteriores. A padronização, muitas vezes alcançada com conteinerização [8], é crucial para a confiabilidade.

3.6 *CI* e Controle de Versão

A prática da *Integração Contínua* (*CI*) [1] está diretamente ligada ao uso de sistemas de controle de versão, como o *Git*[14]. O controle de versão é a fundação que permite:

- Organização de *branches* e estratégias de *merge* (como *Git Flow*, *trunk-based development*).
- Disparo de *pipelines* a partir de eventos no repositório (*push*, *pull request*, *tag*).
- Registro histórico de alterações e rastreabilidade.

Plataformas como *GitHub* [5], *GitLab* [6], *Bitbucket* [15] e *Azure DevOps* integram nativamente recursos de *CI*, permitindo que repositórios acionem *pipelines* automaticamente, monitorando qualidade de código, cobertura de testes e *status* de *build* a cada *commit* realizado.

4. ENTREGA CONTÍNUA (*CD*) E IMPLANTAÇÃO CONTÍNUA (*CD*)

4.1 Conceito e Distinções Fundamentais

Entrega Contínua (*Continuous Delivery*) [1] e *Implantação Contínua* (*Continuous Deployment*) [1] são práticas que estendem a *Integração Contínua* (*CI*) [1] para os estágios finais do ciclo de vida do *software*. Ambas visam acelerar o processo de liberação de novas versões do sistema, porém com graus diferentes de automação.

A *Entrega Contínua* [1] é a capacidade de liberar *software* em produção a qualquer momento, de forma segura e com um nível elevado de automação. Contudo, a liberação efetiva para produção ainda pode depender de uma decisão humana.

A *Implantação Contínua*, por sua vez, elimina essa barreira. Todo código que passa pela esteira de testes automatizados e validações é automaticamente implantado no ambiente de produção, sem necessidade de intervenção manual [1].

Essas práticas não apenas reduzem o *time-to-market*, como também promovem a confiabilidade e a padronização dos processos de liberação de *software* [33][34][35][36][37][38].

4.2 Arquitetura de um *Pipeline* de *CD*

Um *pipeline* de *CD* [1] bem estruturado inclui várias etapas automatizadas, garantindo a qualidade e a eficiência do processo de liberação de *software*:

- ***Build***: Esta fase envolve a compilação e o empacotamento do código-fonte em artefatos executáveis ou *containers*[8]. É o momento em que o código se transforma em um produto pronto para ser testado e implantado.
- **Testes**: Uma *suíte* abrangente de testes é executada, incluindo testes unitários [9], de integração, de segurança (*DevSecOps*) [26], [27], [28] e de

performance. Essa etapa é crucial para garantir a funcionalidade, a estabilidade e a robustez da aplicação.

- **Empacotamento**: Após os testes, a aplicação é empacotada em formatos de fácil distribuição, como imagens *Docker* [8] ou arquivos de instalação, preparando-a para o *deploy*.
- ***Deploy* em ambientes intermediários**: O artefato é implantado em ambientes que simulam o ambiente de produção, como homologação, *QA* (*Quality Assurance*) e *staging*. Isso permite validações adicionais por equipes específicas antes da liberação final.
- ***Deploy* em produção**: A etapa final, onde a aplicação é implantada no ambiente de produção. Pode ser manual, no caso de *Entrega Contínua* [1], onde uma decisão humana final é necessária, ou automático, no caso de *Implantação Contínua*, onde a implantação ocorre sem intervenção manual após todas as validações terem sido bem-sucedidas [1].

Cada estágio desse *pipeline* pode conter verificações de qualidade e *checkpoints*, como a aprovação de um *Product Owner* ou a análise de métricas de desempenho e uso.

4.3 Benefícios da Adoção de *CD/CI*

A implementação de *Entrega Contínua* e *Implantação Contínua* [1] proporciona ganhos significativos em diversas frentes:

- ***Técnicos***:
  - Redução drástica nos erros de *deploy*, uma vez que o processo é automatizado e padronizado [33], [36].
  - Ambientes consistentes e reprodutíveis, o que minimiza o problema de "funciona na minha máquina" [8].
  - *Rollbacks* facilitados em caso de falha, permitindo rápida recuperação e menor impacto ao usuário [33].
- ***Operacionais***:
  - Diminuição da sobrecarga sobre as equipes de operações, que não precisam mais realizar *deploys* manuais e repetitivos [33], [36].
  - *Deploys* fora do horário comercial tornam-se seguros e automatizados, reduzindo a necessidade de intervenção humana em momentos críticos.
- ***Negócios***:
  - *Time-to-market* reduzido, permitindo que novas funcionalidades cheguem mais rapidamente aos usuários [33], [36], [37], [38].
  - Entregas contínuas de valor ao usuário, com atualizações frequentes e incrementais do *software*.
  - Reação mais rápida a mudanças e falhas em produção, possibilitando correções ágeis e adaptação às demandas do mercado.

Segundo o relatório "*State of DevOps*" do DORA (2021) [13], organizações com *pipelines CD* bem implementados são 1.5x mais propensas a atingir metas de negócio, além de apresentarem melhor satisfação do cliente.

4.4 Desafios e Obstáculos Técnicos

Apesar das vantagens, a implementação de *Entrega Contínua* (*CD*) e, especialmente, de *Implantação Contínua*, exige um alto grau de maturidade técnica e cultural [1]:

- **Qualidade dos testes**: Testes mal escritos ou incompletos representam um risco significativo, pois podem aprovar código com *bugs* graves que seriam então liberados diretamente para a produção. A confiança na automação depende da confiabilidade da suíte de testes [9].
- **Infraestrutura**: Há uma necessidade crescente de ambientes espelhados, consistentes e efêmeros, muitas vezes alcançados com a utilização de *containers* [8] e orquestradores como *Kubernetes* [21]. A gestão dessa infraestrutura pode ser complexa.
- **Segurança**: A agilidade do *Continuous Deployment* traz consigo um risco elevado de vazamento de falhas de segurança diretamente para o usuário final, caso não haja práticas robustas de *DevSecOps* [26], [27], [28] integradas ao *pipeline*.
- **Monitoramento e observabilidade**: Sem sistemas de monitoramento (como *Prometheus* [22] e *Grafana* [23]), *logs* detalhados e alertas proativos, falhas em produção podem passar despercebidas ou serem detectadas tardiamente, minimizando os benefícios da entrega rápida.

O sucesso do *CD/CD* depende não apenas da automação robusta, mas também da observabilidade contínua do sistema em produção, permitindo *feedback* rápido e ações corretivas ágeis.

4.5 Princípios Avançados

A otimização de *pipelines* de *Entrega Contínua* e *Implantação Contínua* [1] pode ser aprimorada pela adoção de conceitos avançados que visam maior robustez e flexibilidade:

- ***Build once, deploy many***: Este princípio assegura que o artefato gerado durante o processo de *build* [1] seja único e imutável, sendo utilizado em todos os ambientes – desde o desenvolvimento até a produção. Isso garante

consistência e elimina problemas frequentes de "funciona no meu ambiente" [8].

- ***Feature Toggles*** (ou *Feature Flags*): São mecanismos que permitem ativar ou desativar funcionalidades específicas da aplicação em tempo real, sem a necessidade de um novo *deploy*. Essa técnica reduz o risco das implantações, pois uma nova funcionalidade pode ser desativada instantaneamente caso apresente problemas, e permite testes A/B e lançamentos graduais para subconjuntos de usuários.
- ***Release Train***: Inspirado em metodologias ágeis escaláveis como *SAFe* (*Scaled Agile Framework*) [39], o modelo de *Release Train* estabelece um cronograma de entrega previsível com datas fixas. As equipes se sincronizam para que suas funcionalidades sejam incluídas em "trens" de *release* regulares, promovendo um fluxo contínuo de valor e coordenação em larga escala.

Esses conceitos avançados ajudam a modularizar o processo de entrega e tornam o *pipeline* mais resiliente, adaptável e alinhado às necessidades de negócios dinâmicas.

5. FERRAMENTAS DE *CI/CD*

5.1 Panorama Geral

O mercado de ferramentas de *Integração* e *Entrega Contínuas* (*CI/CD*) [1] é extenso e em constante evolução. A escolha da ferramenta ideal depende de diversos fatores: infraestrutura disponível, linguagem de programação usada, repositório de código, nível de automação desejado, além de critérios como facilidade de uso, escalabilidade, integração com serviços de nuvem e custo.

A seguir, apresentamos uma análise detalhada de três ferramentas amplamente utilizadas: *Jenkins* [4], *GitHub Actions* [5] e *GitLab CI/CD* [6].

5.2 *Jenkins*

a) Visão Geral

Jenkins [4] é uma das ferramentas de *CI/CD* [1] mais consolidadas do mercado. De código aberto e altamente extensível, permite a construção de *pipelines* complexos por meio de *scripts* e *plugins*.

b) Arquitetura

Jenkins [4] adota uma arquitetura mestre-*agente* (*master-agent*), onde o servidor principal orquestra a execução de tarefas delegadas a *agentes* (*runners*), que podem estar distribuídos local ou remotamente.

c) Vantagens

- Altamente customizável com mais de 1.500 plugins.
- Compatível com qualquer linguagem de programação.
- Forte comunidade e ampla documentação.

d) Desvantagens

- Curva de aprendizado acentuada.
- Interface gráfica pouco moderna.
- Exige maior manutenção e configuração manual.

e) Contexto Ideal

Jenkins [4] é recomendado para equipes que precisam de controle total sobre o *pipeline* e contam com recursos técnicos para configurar e manter a ferramenta.

5.3 GitHub Actions

a) Visão Geral

GitHub Actions [5] é uma solução de *CI/CD* [1] integrada diretamente ao *GitHub* [5], permitindo que *workflows* sejam definidos por arquivos *YAML* e executados automaticamente em resposta a eventos do repositório.

b) Arquitetura

É baseado em *runners* hospedados na nuvem (ou locais), disparados por gatilhos como *push*, *pull request* ou *release tags*.

c) Vantagens

- Integração nativa com repositórios *GitHub*[5].
- Fácil configuração e documentação intuitiva.
- Suporte a *containers*[8] e ambientes paralelos.

d) Desvantagens

- Limitado em termos de customização se comparado ao *Jenkins*[4].
- Pode gerar custos dependendo do volume de *builds*.

e) Contexto Ideal

GitHub Actions [5] é perfeito para projetos hospedados no *GitHub* [5] que buscam uma solução rápida, moderna e com baixa manutenção.

5.4 GitLab CI/CD

a) Visão Geral

GitLab CI/CD [6] é um módulo nativo do *GitLab* [6] que permite a construção de *pipelines* robustos diretamente acoplados ao ciclo de vida do repositório, usando arquivos *.gitlab-ci.yml*.

b) Arquitetura

Utiliza executores (*runners*) configuráveis, podendo operar em nuvem, servidores dedicados ou ambientes locais. Os *jobs* são organizados em estágios sequenciais ou paralelos.

c) Vantagens

- Totalmente integrado com repositório, *issues*, *merge requests* e monitoramento.
- Interface gráfica clara e pipelines visuais.
- Suporte nativo a *Kubernetes* [21] e *containers* [8].

d) Desvantagens

- Pode apresentar lentidão em projetos muito grandes.
- Complexidade em configurações avançadas de cache e runners customizados.

e) Contexto Ideal

GitLab CI/CD [6] é ideal para equipes que utilizam o *GitLab* [6] como plataforma de *DevOps* [2], [3] completa, centralizando versionamento, *CI/CD* [1], gestão de projeto e segurança.

5.5 *Docker*

a) Visão Geral

Docker [8] é uma plataforma de código aberto que permite empacotar, distribuir e executar aplicações em **ambientes isolados** chamados *containers*.
Essencialmente, um *container* é uma unidade leve, portátil e autossuficiente que

inclui tudo o que o *software* precisa para funcionar: código, *runtime*, bibliotecas do sistema e configurações. O objetivo principal do *Docker* é eliminar o problema de "funciona na minha máquina", garantindo que a aplicação se comporte da mesma forma em qualquer ambiente que suporte *Docker*.

b) Arquitetura

A arquitetura do *Docker* é baseada em um **modelo cliente-servidor**. O *Docker daemon* (ou *dockerd*) é o servidor persistente que gerencia os *containers*, imagens, volumes e redes. Os usuários interagem com o *daemon* através do **cliente *Docker*** (*docker CLI*), que envia comandos via *API REST*. As **imagens *Docker*** são modelos de leitura que contêm as instruções para criar um *container*, e são construídas a partir de um *Dockerfile*. O **Registro *Docker*** (*Docker Hub* ou um registro privado) é um serviço para armazenar e compartilhar imagens *Docker*.

c) Vantagens

- **Portabilidade e Consistência**: Garante que a aplicação e suas dependências funcionem de forma idêntica em qualquer ambiente, do desenvolvimento à produção.
- **Isolamento**: *Containers* isolam as aplicações umas das outras e do sistema *host*, prevenindo conflitos de dependências.
- **Eficiência de Recursos**: *Containers* compartilham o *kernel* do sistema *host*, sendo muito mais leves e rápidos para iniciar do que máquinas virtuais.
- **Agilidade no Desenvolvimento**: Facilita a configuração de ambientes de desenvolvimento, testes e *deploy*, acelerando o ciclo de vida do *software*.
- **Escalabilidade**: Imagens *Docker* são a base para orquestradores como *Kubernetes* [21], permitindo escalar aplicações de forma eficiente.

d) Desvantagens

- **Complexidade para Iniciantes**: A curva de aprendizado inicial pode ser íngreme para quem não está familiarizado com conceitos de conteinerização.
- **Gerenciamento de Dados Persistentes**: Lidar com o armazenamento persistente de dados dentro de *containers* pode exigir configurações adicionais (uso de volumes).
- **Sobrecarga de Ferramentas**: Em ambientes complexos, pode ser necessário integrar o *Docker* com outras ferramentas de orquestração e monitoramento, aumentando a complexidade.
- **Segurança de Imagens**: É crucial garantir que as imagens utilizadas e construídas sejam seguras para evitar vulnerabilidades.

e) Contexto Ideal

Docker [8] é ideal para qualquer equipe que busca padronizar seus ambientes, garantir a portabilidade de suas aplicações e otimizar os processos de *build* e *deploy* dentro de um *pipeline CI/CD* [1]. É particularmente útil em arquiteturas de microsserviços, em projetos que precisam de ambientes consistentes entre desenvolvedores e para facilitar o *deploy* em ambientes de nuvem ou *on-premise*. Sua adoção é fundamental para quem planeja utilizar orquestradores de *containers* como *Kubernetes* [21] em escala de produção.

5.6 Justificativa da Escolha para o Projeto Prático

Para a parte prática deste trabalho, a ferramenta *GitHub Actions* [5] foi escolhida para a implementação do *pipeline CI/CD* [1]. Essa decisão levou em consideração diversos critérios, especialmente a **f**alta de experiência prévia da equipe com ferramentas de automação de *pipeline*, e a compatibilidade com o *stack* tecnológico do projeto, que inclui *React.js*[7] para o *frontend* e *Docker* [8] para conteinerização.

Os principais motivos para a escolha de *GitHub Actions* [5] são:

- **Integração direta com o repositório**: O projeto prático será desenvolvido em um repositório *GitHub* [5], e a integração nativa do *GitHub Actions* simplifica a configuração e o gerenciamento dos *workflows* diretamente a partir do código-fonte.
- **Facilidade na criação de *pipelines* com *YAML***: A definição dos *pipelines* é feita por meio de arquivos *YAML*, que são intuitivos e possuem uma sintaxe clara, facilitando a aprendizagem e a colaboração da equipe.
- **Baixa curva de aprendizado**: Dada a pouca experiência da equipe com *CI/CD*, a interface amigável e a vasta documentação do *GitHub Actions* [5] promovem uma adoção mais rápida e eficiente, permitindo que todos os membros contribuam ativamente para a automação.
- **Infraestrutura gerenciada (*runners* na nuvem)**: A utilização de *runners* hospedados na nuvem [5] elimina a complexidade operacional de configurar e manter a infraestrutura de *CI/CD*, permitindo que a equipe foque no desenvolvimento e na lógica do *pipeline*.
- **Compatibilidade com o *stack* tecnológico**: A ferramenta é totalmente compatível com o ambiente do projeto, suportando a construção de aplicações em *React.js* [7], a execução de testes automatizados com *Jest* [9] e a criação e *deploy* de *containers Docker* [8], conforme detalhado na seção do projeto prático.

Essa escolha estratégica visa maximizar a eficiência do aprendizado da equipe e garantir a implementação de um *pipeline* de *CI/CD* robusto, alinhado às práticas modernas de desenvolvimento de *software*.

5.6.1 Projeto: API de Previsão do Tempo com CI/CD

O projeto desenvolvido no âmbito da disciplina de Engenharia de Software (IFCE – 2025.1) focou na materialização prática dos conceitos de *DevOps* [2], [3], *Integração Contínua* (*CI*) e *Entrega Contínua* (*CD*) [1] através da construção de uma *API* de Previsão do Tempo. Esta aplicação *web*, totalmente funcional, permite aos usuários consultar a previsão climática atual para diversas cidades, consumindo dados diretamente de uma *API* pública (*OpenWeatherMap* [11]). O principal objetivo foi simular um ciclo de desenvolvimento de *software* moderno, abrangendo desde a concepção e codificação até a automação de testes, conteinerização e *deploy.*

A arquitetura da solução é composta por um *frontend* dinâmico desenvolvido em *React.js* [7], uma biblioteca *JavaScript* amplamente utilizada para construção de interfaces de usuário, e estilizado com *CSS Modules*. Ele interage de forma assíncrona com a *API* externa (*OpenWeatherMap* [11]) para obter os dados. Para garantir a responsividade e a gestão do *estado* da aplicação, foram amplamente utilizados os *React Hooks* (*useState* e *useEffect)*, um recurso padrão do *React*[7]. A funcionalidade central da *API* é acessível através da rota */weather?city=nome_da_cidade*, que retorna um objeto *JSON* detalhado, contendo a temperatura atual, sensação térmica, umidade relativa do ar, uma descrição textual do clima e um ícone correspondente. Um dos pontos cruciais do desenvolvimento foi a implementação de tratamento de erros robusto, garantindo que a aplicação lide elegantemente com cenários como cidades inexistentes ou falhas na comunicação com a *API* externa, fornecendo *feedback* claro ao usuário.A qualidade do código foi assegurada pela implementação de testes automatizados utilizando Jest e React Testing Library. Esses testes abrangem a renderização dos componentes, o comportamento da aplicação em diferentes cenários (dados válidos, inválidos e erros), e incluem o *mocking* da API para garantir a reprodutibilidade e o isolamento dos testes, um pilar essencial da garantia de qualidade em um ambiente de CI.

A qualidade do código foi assegurada pela implementação de testes automatizados utilizando *Jest* [9], um *framework* de testes *JavaScript*, e *React Testing Library*, uma biblioteca focada em testar componentes *React* da perspectiva do usuário. Esses testes abrangem a renderização dos componentes, o comportamento da aplicação em diferentes cenários (dados válidos, inválidos e erros), e incluem o *mocking* da *API* para garantir a reprodutibilidade e o isolamento dos testes, um pilar essencial da garantia de qualidade em um ambiente de *CI*.

A conteinerização foi uma etapa fundamental, com a aplicação sendo empacotada em imagens *Docker* [8], uma plataforma líder para desenvolver, enviar e executar aplicações em *containers*. Isso não apenas assegura a portabilidade da solução, mas também padroniza o ambiente de execução, eliminando problemas de compatibilidade e facilitando o *deploy*. A imagem *Docker* do *frontend* é configurada

para ser servida por um *servidor Nginx* [10], um *servidor web* e *proxy reverso* de alta performance, otimizando a entrega dos arquivos estáticos.

A espinha dorsal da automação do ciclo de vida do desenvolvimento é o *pipeline* de *CI/CD* [1], [3] implementado com *GitHub Actions* [5]. Este *pipeline* é dividido em duas etapas principais: *Integração Contínua* (*CI*) e *Entrega Contínua*(*CD*) [1]. No estágio de *CI* (*ci.yml*), cada alteração no código é automaticamente submetida a *análise estática* (*Lint*), execução de todos os testes automatizados e o *build* da aplicação *frontend*. Uma vez que essas etapas são concluídas com sucesso, o *pipeline* de *Entrega Contínua* (*CD*) (*cd.yml*) é acionado, realizando o *build* da imagem *Docker* [8] da aplicação, o *push* dessa imagem para um registro de *containers* (como o *Docker Hub*) e, por fim, o *deploy* automático da nova versão para um ambiente de homologação (utilizando plataformas como *Render* ou *Vercel*). Essa automação garante que cada alteração validada no repositório seja rapidamente disponibilizada, simulando um ambiente de produção real.

A metodologia de desenvolvimento adotada foi o *Scrum*, com a equipe dividida em funções específicas para otimizar o trabalho colaborativo. A Kelly foi responsável pelo desenvolvimento da *interface* (*UI*), garantindo a responsividade e a integração visual com o *estado* da aplicação. A Raynara focou na lógica de consumo da *API* [11], gerenciamento de *estado* e na implementação de testes de unidade e integração. Martenier cuidou da infraestrutura de *CI/CD* com *GitHub Actions* [5], da criação do *Dockerfile* e da configuração do *deploy* automático. A Alana dedicou-se à elaboração da documentação técnica, seguindo o padrão ABNT, e ao detalhamento do funcionamento da aplicação. Por fim, a Soraia foi encarregada do relatório teórico sobre *DevOps* [2], [3] e *CI/CD* [1], justificando as ferramentas utilizadas e preparando a apresentação final do projeto. Essa abordagem não só promoveu a especialização dos membros, mas também incentivou a colaboração e o aprendizado prático em diversas frentes da engenharia de *software* moderna. Ao final, o projeto não apenas resultou em uma aplicação funcional, mas também em um conjunto robusto de documentação técnica e relatórios, consolidando o aprendizado e demonstrando a aplicação prática dos conceitos de engenharia de *software* em um cenário completo de desenvolvimento e *deploy*.

6. CONCLUSÃO

A adoção de práticas *DevOps* [2], [3] e *pipelines* de *CI/CD* [1] se consolida como uma estratégia essencial para equipes de desenvolvimento que buscam agilidade, confiabilidade e inovação contínua. Ao compreender os pilares culturais e técnicos do *DevOps*, bem como as diferenças entre *Integração*, *Entrega* e *Implantação Contínuas*, equipes se capacitam a entregar mais valor com menos retrabalho e menor risco. A aplicação prática das ferramentas analisadas fortalece o entendimento dos conceitos e permite sua adaptação à realidade de projetos reais, como demonstrado neste trabalho.

O projeto prático desenvolvido, uma *API* de Previsão do Tempo com *CI/CD*, resultou em um produto funcional e demonstrou a aplicabilidade dos conceitos discutidos. O desenvolvimento foi realizado ao longo de aproximadamente um mês, com tarefas leves distribuídas semanalmente, o que reforçou a abordagem do paradigma incremental [40], permitindo entregas contínuas e *feedback* constante. O *feedback* da equipe de desenvolvimento reforça o sucesso da abordagem e a escolha das tecnologias, mesmo diante da inexperiência inicial de alguns membros com certas ferramentas.

Em relação à Integração Contínua com *GitHub Actions* [5], o retorno foi majoritariamente positivo. Um dos membros, Martenier, que já possuía experiência prévia com conceitos de *backend* e *CI/CD* em outros contextos, não encontrou dificuldades. Contudo, ao ser questionado sobre o uso específico de *GitHub Actions*, ele esclareceu que nunca havia utilizado a ferramenta anteriormente. Apesar disso, sua percepção foi de que a ferramenta é "legal, bem útil", por "deixar o projeto mais organizado" e "evitar erro besta indo pra *branch* principal", uma vez que "obriga a gente a revisar o que foi feito antes de juntar com o resto". Essa observação destaca a eficácia do *GitHub Actions* em promover a qualidade do código e a colaboração, cumprindo o objetivo de facilitar a revisão e o *feedback* rápido.

No desenvolvimento do *frontend* com React.js [7], Kelly, que nunca havia trabalhado com *front web* antes, considerou a experiência "super tranquila" e "bem intuitiva". Embora tenha enfrentado uma pequena dificuldade com a responsividade, conseguiu resolver facilmente. Sua familiaridade e preferência pelo Git Flow [14] contribuíram para a organização do trabalho. A facilidade de aprendizado do *React* para iniciantes em desenvolvimento *web* foi um ponto forte.

Quanto à integração com a API [11], Raynara destacou que a plataforma da *API* externa era "bem documentada e já tinha alguns exemplos de como implementar o código e era bem completa", o que facilitou significativamente o consumo dos dados e a implementação das funcionalidades.

Em suma, a combinação da metodologia *DevOps* com ferramentas como *GitHub Actions*, *React.js* e uma *API* bem documentada permitiu que a equipe, mesmo com

níveis variados de experiência, desenvolvesse e implantasse um projeto de forma eficaz e satisfatória, comprovando os benefícios da automação e das práticas de *CI/CD* em um cenário prático de engenharia de *software*.

6.1 RESULTADOS OBTIDOS

A implementação prática do projeto culminou no desenvolvimento de uma aplicação de previsão do tempo com uma interface de usuário intuitiva e funcional, conforme ilustrado nas Figuras 1 e 2. O objetivo foi validar os conceitos de *DevOps* e *CI/CD* em um cenário real, resultando em um sistema que permite aos usuários buscar informações meteorológicas de diferentes localidades.

Figura 1 - Interface do site ao pesquisar clima de Fortaleza

Tela da aplicação web resultante do projeto prático, mostrando a previsão do tempo obtida para a cidade de Fortaleza, incluindo temperatura, umidade, condições do tempo e coordenadas geográficas.

Figura 2 - Interface do site ao pesquisar o clima de São Paulo

Tela da aplicação web resultante do projeto prático, mostrando a previsão do tempo obtida para a cidade de São Paulo, incluindo temperatura, umidade, condições do tempo e coordenadas geográficas.

Referência : autores

7. REFERÊNCIAS

[1] HUMBLE, Jez; FARLEY, David. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Boston: Addison-Wesley, 2010.

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

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

[4] JENKINS. *Jenkins Documentation*. Disponível em: https://www.jenkins.io/doc/. Acesso em: 29 jun. 2025.

[5] GITHUB. *GitHub Docs - Actions*. Disponível em: https://docs.github.com/en/actions. Acesso em: 29 jun. 2025.

[6] GITLAB. *GitLab Docs - CI/CD*. Disponível em: https://docs.gitlab.com/ee/ci/. Acesso em: 1 jul. 2025.

[7] REACT. *Documentação Oficial do React*. Disponível em: https://react.dev/docs. Acesso em: 15 jul. 2025.

[8] DOCKER. *Docker Documentation*. Disponível em: https://docs.docker.com/. Acesso em: 15 jul. 2025.

[9] JEST. *Jest Documentation*. Disponível em: https://jestjs.io/docs/getting-started. Acesso em: 15 jul. 2025.

[10] NGINX. *NGINX Documentation*. Disponível em: https://nginx.org/en/docs/. Acesso em: 15 jul. 2025.

[11] OPENWEATHERMAP. *Current Weather Data API*. Disponível em: https://openweathermap.org/current. Acesso em: 15 jul. 2025.

[12] WEST, Dave. *Os pragmáticos venceram? A norma é o Water-Scrum-Fall*. InfoQ Brasil, 11 jan. 2012. Disponível em: https://www.infoq.com/br/news/2012/01/water-scrum-fall/. Acesso em: 15 jul. 2025.

[13] DORA. *Accelerate State of DevOps Report 2024*. Disponível em: https://dora.dev/research/2024/dora-report/. Acesso em: 15 jul. 2025.

[14] GIT. *Git Documentation*. Disponível em: https://git-scm.com/doc. Acesso em: 15 jul. 2025. [15] BITBUCKET. *Bitbucket Documentation*. Disponível em: https://support.atlassian.com/bitbucket-cloud/. Acesso em: 15 jul. 2025.

[16] CIRCLECI. *CircleCI Documentation*. Disponível em: https://circleci.com/docs/. Acesso em: 15 jul. 2025.

[17] ANSIBLE. *Ansible Documentation*. Disponível em: https://docs.ansible.com/. Acesso em: 15 jul. 2025.

[18] PUPPET. *Puppet Documentation*. Disponível em: https://puppet.com/docs/. Acesso em: 15 jul. 2025.

[19] CHEF. *Chef Documentation*. Disponível em: https://docs.chef.io/. Acesso em: 15 jul. 2025.

[20] PODMAN. *Podman Documentation*. Disponível em: https://podman.io/docs/. Acesso em: 15 jul. 2025.

[21] KUBERNETES. *Kubernetes Documentation*. Disponível em: https://kubernetes.io/docs/. Acesso em: 15 jul. 2025.

[22] PROMETHEUS. *Prometheus Documentation*. Disponível em: https://prometheus.io/docs/. Acesso em: 15 jul. 2025.

[23] GRAFANA. *Grafana Documentation*. Disponível em: https://grafana.com/docs/. Acesso em: 15 jul. 2025.

[24] ELASTIC. *Elastic Stack Documentation*. Disponível em: https://www.elastic.co/guide/index.html. Acesso em: 15 jul. 2025.

[25] DATADOG. *Datadog Documentation*. Disponível em: https://docs.datadoghq.com/. Acesso em: 15 jul. 2025.

[26] SONARQUBE. *SonarQube Documentation*. Disponível em: https://docs.sonarsource.com/sonarqube/. Acesso em: 15 jul. 2025.

[27] SNYK. *Snyk Documentation*. Disponível em: https://docs.snyk.io/. Acesso em: 15 jul. 2025.

[28] TRIVY. *Trivy Documentation*. Disponível em: https://trivy.dev/docs/. Acesso em: 15 jul. 2025.

[29] FOWLER, Martin. *Continuous Integration*. MartinFowler.com, 2006. Disponível em: https://martinfowler.com/articles/continuousIntegration.html. Acesso em: 15 jul. 2025.

[30] ZEET. *20 Benefits of Continuous Integration for Engineering Teams*. Zeet.co, 16 jan. 2024. Disponível em: https://zeet.co/blog/benefits-of-continuous-integration. Acesso em: 15 jul. 2025.

[31] TIERPOINT. *Top 12 Benefits of Continuous Integration*. TierPoint, LLC, 2 ago. 2023. Disponível em: https://www.tierpoint.com/blog/benefits-of-continuous-integration/. Acesso em: 15 jul. 2025.

[32] APIUMHUB. *What are the benefits of continuous integration?*. Apiumhub. Disponível em: https://apiumhub.com/tech-blog-barcelona/benefits-of-continuous-integration/. Acesso em: 15 jul. 2025.

[33] OCTOPUS. *Continuous Deployment: Benefits, Pros/cons, Tools And Tips*. Octopus. Disponível em: https://octopus.com/devops/continuous-deployment/. Acesso em: 15 jul. 2025.

[34] BROWSERSTACK. *Continuous Delivery vs Continuous Deployment: Core Differences*. BrowserStack. Disponível em: https://www.browserstack.com/guide/continuous-delivery-vs-continuous-deployment. Acesso em: 15 jul. 2025.

[35] ATLASSIAN. *Continuous integration vs. delivery vs. deployment*. Atlassian. Disponível em: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment. Acesso em: 15 jul. 2025.

[36] AWS. *Benefits of continuous delivery - AWS Documentation*. AWS. Disponível em: https://docs.aws.amazon.com/whitepapers/latest/practicing-continuous-integration-continuous-delivery/benefits-of-continuous-delivery.html. Acesso em: 15 jul. 2025.

[37] RESEARCHGATE. *A STUDY AND ANALYSIS OF CONTINUOUS DELIVERY, CONTINUOUS INTEGRATION IN SOFTWARE DEVELOPMENT ENVIRONMENT*. ResearchGate. Disponível em: https://www.researchgate.net/publication/354720705_A_STUDY_AND_ANALYSIS_OF_CONTINUOUS_DELIVERY_CONTINUOUS_INTEGRATION_IN_SOFTWARE_DEVELOPMENT_ENVIRONMENT. Acesso em: 15 jul. 2025.

[38] CIT. *The effect of using continuous integration, delivery and deployment on the software systems development process in the cloud technology environment*. Computer-Integrated Technologies: Education, Science,

Production. Disponível em: https://cit.lntu.edu.ua/index.php/cit/article/view/628. Acesso em: 15 jul. 2025.

[39] LEFFINGWELL, Dean. *SAFe Reference Guide: Scaled Agile Framework for Lean Software and Systems Engineering*. Scaled Agile, Incorporated, 2016.

[40] SOMMERVILLE, Ian. *Engenharia de Software*. 10. ed. São Paulo: Pearson Prentice Hall, 2019. p. 48-49.