UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS SOBRAL

CURSO DE GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

VINICIUS TEIXEIRA COSTA

AUTOMAÇÃO DE PROVISIONAMENTO DE UMA PLATAFORMA DE BORDA COM TERRAFORM E ANSIBLE: UMA ABORDAGEM DE INFRAESTRUTURA COMO CÓDIGO

SOBRAL

2026

VINICIUS TEIXEIRA COSTA

AUTOMAÇÃO DE PROVISIONAMENTO DE UMA PLATAFORMA DE BORDA COM TERRAFORM E ANSIBLE: UMA ABORDAGEM DE INFRAESTRUTURA COMO CÓDIGO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação do Campus Sobral da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia da Computação.

Orientador: Prof. Dr. Iális Cavalcante de Paula Júnior.

SOBRAL

2026

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

C876a Costa, Vinicius Teixeira.
AUTOMAÇÃO DE PROVISIONAMENTO DE UMA PLATAFORMA DE BORDA COM TERRAFORM E ANSIBLE : UMA ABORDAGEM DE INFRAESTRUTURA COMO CÓDIGO / Vinicius Teixeira Costa. – 2026.
60 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Sobral, Curso de Engenharia da Computação, Sobral, 2026.
Orientação: Prof. Dr. Iális Cavalcante de Paula Júnior.

1. Automação. 2. Plataforma de Borda. 3. Infraestrutura como código. 4. Terraform. 5. Ansible. I. Título.
CDD 621.39

VINICIUS TEIXEIRA COSTA

AUTOMAÇÃO DE PROVISIONAMENTO DE UMA PLATAFORMA DE BORDA COM TERRAFORM E ANSIBLE: UMA ABORDAGEM DE INFRAESTRUTURA COMO CÓDIGO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação do Campus Sobral da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia da Computação.

Aprovada em: 29 de janeiro de 2026

BANCA EXAMINADORA

Prof. Dr. Iális Cavalcante de Paula Júnior (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Me. Erick Aguiar Donato
Universidade Federal do Ceará (UFC)

Prof. Adriano Trajano Rodrigues
Universidade Federal do Ceará (UFC)

À minha família, por sua capacidade de acreditar em mim e investir em mim. Mãe, seu cuidado e dedicação foi que deram, em alguns momentos, a esperança para seguir. Pai, sua presença significou segurança e certeza de que não estou sozinho nessa caminhada.

AGRADECIMENTOS

Agradeço, primeiramente, ao meu pai Valdinar e à minha mãe Maria do Amparo pela construção de toda a base que me fez chegar até aqui. Agradeço por todo amor, suporte e compreensão que me foram desprendidos, sempre me proporcionando as melhores condições possíveis. Essa conquista é para vocês.

Agradeço ao meu irmão Vitor, por sempre me inspirar e motivar a ir em busca de novas conquistas. À minha avó, que com seu jeito carinhoso sempre perguntava sobre o andamento da faculdade, demonstrando sua torcida e preocupação constante com o meu futuro.

Agradeço à Universidade Federal do Ceará por me proporcionar um ensino superior de alta qualidade e a oportunidade de conhecer diversos amigos nesses anos de graduação. Ao Professor Ialis, meu orientador, agradeço não apenas pelos grandes conhecimentos técnicos compartilhados, mas também por ser um professor atencioso e estar sempre disposto a auxiliar.

Um agradecimento à Lizia, que foi fundamental nesta etapa final, sempre me motivando a não desistir e a concluir este trabalho.

Ao Levi, deixo meu muito obrigado pela parceria e mentoria técnica durante meu tempo no Instituto Atlântico. Seus ensinamentos foram vitais para meu desenvolvimento prático. Estendo minha gratidão à Adrielly, pelo incentivo crucial no momento em que decidi realizar a transição de carreira, seu apoio foi o ponto de partida para minha trajetória em DevOps.

Por fim, estendo minha gratidão aos colegas e equipes da Unimed Sobral, do Instituto Atlântico e do GREat. O convívio profissional e o aprendizado prático adquirido nessas instituições foram essenciais para o meu amadurecimento como profissional e enriqueceram a visão técnica aplicada neste trabalho.

RESUMO

A crescente demanda por agilidade e consistência na entrega de aplicações, especialmente em cenários de plataformas de borda, tem impulsionado a adoção de práticas de *Infraestrutura como Código* (IaC). A complexidade do gerenciamento manual de infraestruturas distribuídas e a necessidade de padronização para garantir a eficiência operacional e segurança frequentemente resultam em falhas e atrasos. Este trabalho propõe uma solução para esse desafio, explorando a automação do provisionamento de uma plataforma de borda utilizando as ferramentas Terraform e Ansible. O objetivo principal foi demonstrar a viabilidade e os benefícios de uma abordagem de IaC para construir, configurar e gerenciar de forma automatizada a infraestrutura de uma plataforma de borda. Para tanto, empregou-se o Ansible para a automação da configuração e implantação dos componentes necessários da aplicação, e o Terraform para o provisionamento da aplicação e seus *addons*. Como resultado, demonstrou-se a capacidade de implantar e gerenciar de forma eficiente e repetível o ambiente, reduzindo o tempo de provisionamento e minimizando erros humanos. A abordagem adotada visa otimizar o ciclo de desenvolvimento e operação, oferecendo uma infraestrutura robusta, escalável e de fácil manutenção para aplicações distribuídas.

Palavras-chave: Automação. Plataforma de Borda. Infraestrutura como código. Terraform. Ansible.

ABSTRACT

The increasing demand for agility and consistency in application delivery, especially in edge platform scenarios, has driven the adoption of Infrastructure as Code (IaC) practices. The complexity of manual management of distributed infrastructures and the need for standardization to ensure operational efficiency and security frequently result in failures and delays. This work proposes a solution to this challenge by exploring the automation of edge platform provisioning using Terraform and Ansible tools. The main objective was to demonstrate the feasibility and benefits of an IaC approach to automatically build, configure, and manage an edge platform infrastructure. To this end, Ansible was used for the automation of the necessary application component configuration and deployment, and Terraform for the provisioning of the application and its addons. As a result, the ability to efficiently and repeatably deploy and manage the environment was demonstrated, reducing provisioning time and minimizing human errors. The adopted approach aims to optimize the development and operations cycle, offering a robust, scalable, and easily maintainable infrastructure for distributed applications.

Keywords: Automation; Edge Platform; Infrastructure as Code; Terraform; Ansible.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CD	*Continuous Deployment*
CI	*Continuous Integration*
CLI	*Command Line Interface*
HCL	*HashiCorp Configuration Language*
IaC	*Infraestrutura como Código*
IP	Protocolo de Internet
JSON	*JavaScript Object Notation*
SSH	*Secure Shell*
TI	Tecnologia da Informação
VM	*Virtual Machine*
YAML	*YAML Ain't Markup Language*

SUMÁRIO

1 INTRODUÇÃO

O aumento da complexidade dos sistemas de *software* e a necessidade de oferecer serviços de forma ágil e eficaz estimularam o surgimento de novas estratégias para a gestão de infraestrutura. A emergência das tecnologias de computação em nuvem transformou radicalmente a maneira como os serviços de Tecnologia da Informação (TI) são implementados, proporcionando vantagens como diminuição de gastos, escalabilidade sob demanda, confiabilidade e segurança (DE DONNO; TANGE; DRAGONI, 2019; MARSTON et al., 2011; GAJBHIYE; SHRIVASTVA, 2014). No entanto, de acordo com Manvi e Shyam (2014), os maiores desafios associados com infraestrutura em ambiente *cloud* são o de gerenciamento de recursos, a utilização de ferramentas de Infraestrutura como Código (IaC), apresenta-se como uma alternativa capaz de facilitar o processo de provisionamento e gerenciamento de infraestrutura.

No caso da abordagem à plataforma de borda, deve-se observar que esse nível de infraestrutura desempenha um papel estratégico ao aproximar o processamento de dados dos dispositivos e usuários finais. Contudo, o provisionamento e a configuração desses ambientes ainda representam desafios consideráveis, principalmente quando executados de forma manual ou sem padronização. Dessa forma, com o uso combinado de ferramentas como Terraform e Ansible é possível ter uma solução robusta utilizada para automatizar a criação e facilitar o gerenciamento de infraestruturas distribuídas, alinhando-se às melhores práticas de IaC (HASHICORP, 2023; RED HAT, 2023).

Neste contexto, este projeto, se trata de uma automação do provisionamento de uma plataforma de borda, onde foram utilizadas as ferramentas, Ansible provisionamento e configuração da infraestrutura e Terraform para provisionamento da aplicação e dos *addons*. Esse trabalho busca não apenas validar a eficiência e a confiabilidade da automação no contexto do projeto, mas também explorar os benefícios proporcionados pela padronização dos processos e pela redução do tempo de provisionamento.

A pesquisa adota como estudo de caso o provisionamento de uma plataforma de borda simulada, onde serão analisados os benefícios da automação, como redução do tempo de provisionamento, mitigação de erros e maior previsibilidade das operações. A coleta de dados será realizada a partir da comparação entre cenários manuais e automatizados, buscando evidenciar as vantagens da solução proposta. Além de contribuir com a literatura acadêmica sobre automação e plataformas de borda.

1.1 Justificativa

Neste atual contexto de provisionamento e automação em plataformas de borda, observa-se que a instalação manual de requisitos e complementos, a configuração de ambientes e a integração das tecnologias envolvem múltiplas etapas que frequentemente demandam a colaboração de diferentes equipes, como desenvolvimento e infraestrutura. Esse processo manual é propenso a erros, retrabalho e inconsistências, além de prolongar significativamente o tempo de provisionamento e implantação.

Ao atender às necessidades do projeto, em um ambiente de desenvolvimento, identificou-se que a criação de clusters em máquinas virtuais, em inglês, *Virtual Machine* (VM) e a posterior configuração de componentes essenciais para o uso, tais como, Poetry, Docker, K3D CLI, kubectl CLI, Helm, e MQTT Client UI apresentavam desafios relacionados à eficiência. Esse cenário, baseado em processos parcialmente manuais, não apenas aumenta a suscetibilidade a falhas, mas também dificulta a padronização e a repetibilidade das operações.

Dessa forma, com a implementação de uma abordagem automatizada, fundamentada em *Infrastructure as Code* (IaC) com o uso das ferramentas Terraform e Ansible, surge como uma solução viável e estratégica para superar esses desafios, visto que, ela permite não apenas reduzir significativamente o tempo necessário para o provisionamento e configuração, mas também estabelecer um processo padronizado, previsível e escalável da plataforma de borda. Além disso, a integração de ferramentas modernas para o *deploy* de *addons* promove uma abordagem estruturada e uniforme no gerenciamento da plataforma.

A automação proposta busca não apenas resolver as dores imediatas do projeto e do time de desenvolvimento da borda, mas também tem como objetivo estabelecer um modelo replicável que pode ser aplicado em outros ambientes. Essa estratégia reforça os benefícios da Integração Contínua (*Continuous Integration* (CI)) e da Entrega Contínua (*Continuous Deployment* (CD)), promovendo a otimização dos processos e a redução de riscos, ao mesmo tempo que aumenta a qualidade, a eficiência e a previsibilidade das entregas.

1.2 Objetivos

1.2.1 Objetivo geral

Explorar de maneira detalhada como a automação do provisionamento de uma plataforma de borda, com uso de Terraform e Ansible, pode facilitar na implantação e contribuir com o gerenciamento da infraestrutura. Isso resulta em maior eficiência operacional, uniformidade nos ambientes e diminuição de erros manuais durante o processo de provisionamento.

1.3 Objetivos Específicos

1. Analisar os requisitos para automatizar o provisionamento da plataforma de borda.
2. Desenvolver uma abordagem de IaC utilizando as ferramentas Terraform e Ansible.
3. Validar a automação do provisionamento em um ambiente controlado.
4. Comparar o provisionamento manual e automatizado em termos de desempenho e confiabilidade.

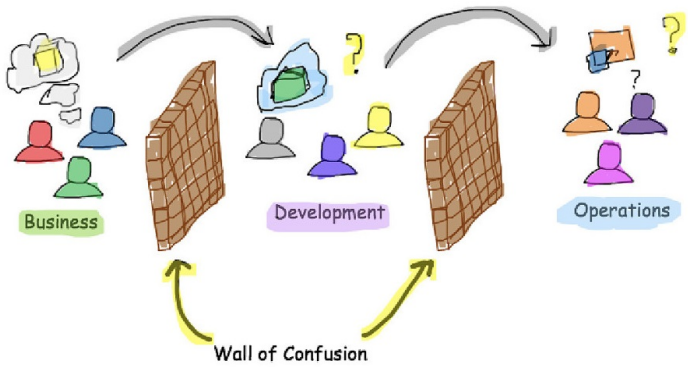
2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo será apresentado o embasamento teórico, coletado de referências literárias e utilizado para teorizar e fundamentar a temática deste trabalho. Serão abordados os conceitos de DevOps e entrega contínua, a evolução da gestão de infraestrutura, o papel estratégico das plataformas de borda, e os princípios e benefícios da Infraestrutura como Código (IaC), com foco nas ferramentas Terraform e Ansible.

2.1 Tecnologia da Informação Tradicional: Desafios na Gestão de Infraestrutura

No modelo tradicional de TI, existe comumente uma divisão entre os times de desenvolvimento e de operações. Enquanto o primeiro é responsável por criar, manter e evoluir os sistemas, o segundo trata de manter a estabilidade e operação das aplicações. De fato, essa separação de responsabilidades frequentemente gera conflitos: o time de desenvolvimento é incentivado a promover mudanças contínuas, o que pode causar instabilidade, enquanto o time de operações busca evitar alterações que possam comprometer a confiabilidade dos sistemas (SATO, 2014). Dessa forma, esse impasse entre as equipes é denominado por Shafer (2009) como "Wall of Confusion" (muro da confusão), onde representa a barreira que impede a colaboração entre os dois lados, conforme ilustrado na Figura 1.

Figura 1 – O Muro da Confusão (Wall of Confusion)



Fonte: VisionTemenos

A Figura 1 mostra bem esse muro, que na verdade é invisível, e acaba dividindo os times e gerando um atrito constante no ambiente. A verdade é que a raiz de tudo isso está nos objetivos bem diferentes de cada equipe: enquanto uma sonha em inovar e mudar, a outra busca

a todo custo manter a estabilidade (EDWARDS, 2010). Como bem apontam Kim et al. (2018), essa separação dificulta demais a empresa de alcançar seus objetivos maiores, e o resultado final são entregas de software com menos qualidade e clientes insatisfeitos.

Os autores apontam três vilões principais que fazem os departamentos de TI perderem a eficiência, não importa o tamanho da empresa:

1. A infraestrutura é um emaranhado complexo, mal explicado e que vive caindo: Isso significa que o pessoal vive "apagando incêndios"com soluções rápidas para o dia a dia, e as verdadeiras melhorias ficam sempre para um futuro que nunca chega.

2. Compensações que viram bola de neve: Um exemplo seria quando um gerente de produto promete uma função nova para disfarçar um atraso ou um erro, assim, isso joga a equipe em uma corrida para efetuar a entregar da nova demanda, muitas vezes sem tempo hábil.

3. Tudo fica mais pesado e difícil: Com o tempo, os problemas se acumulam, todo mundo fica sobrecarregado, a comunicação vira um arrasto e o número de tarefas só cresce. Dessa forma, até uma mudança pequena começa a dar um "frio na barriga", gerando um medo enorme e uma resistência a mexer em qualquer coisa.

De acordo com Sato (2014) essa deterioração nos processos culmina em um aumento da burocracia para o trânsito de código das fases de desenvolvimento e testes para o ambiente de produção. Com o tempo, essa burocracia excessiva reduz a frequência das implantações, os chamados *deploys*, o que, por sua vez, leva ao acúmulo de alterações. Consequentemente, cada nova entrega torna-se substancialmente mais complexa e inerentemente arriscada.

2.2 DevOps: Uma Transformação Cultural

A crescente complexidade dos sistemas e a busca por maior agilidade na entrega de software evidenciaram as limitações do modelo tradicional de TI, no qual as equipes de desenvolvimento e operações atuavam de forma isolada. Historicamente, os desenvolvedores eram responsáveis pela criação de aplicações, enquanto os profissionais de operações pela infraestrutura e execução. O modelo gerava gargalos, retrabalho e conflitos de interesse, principalmente quando mudanças rápidas e frequentes eram necessárias (Brikman, 2019; Hüttermann, 2012).

Nesse contexto, emerge o movimento DevOps, não como uma ferramenta ou cargo específico, mas como uma mudança cultural profunda que integra pessoas, processos e tecnologias em prol da entrega contínua de valor. DevOps é a junção dos termos *Development* e *Operations*, e representa uma abordagem colaborativa baseada nos princípios do desenvolvimento

ágil. Seu objetivo é reduzir o tempo entre a concepção de uma mudança e sua disponibilização em produção, promovendo automação, integração contínua, entrega contínua e *feedback* constante (Sharma, 2014; Brikman, 2019).

Como destacam Beyer et al. (2016), o DevOps basicamente redefine o papel da TI, acompanhando as aplicações em todas as suas fases. O grande salto aqui é a troca de interações manuais — aquelas que dependiam muito de pessoas e podiam causar erros — por automações poderosas. Essa abordagem também integra ferramentas que antes não se "conversavam" e, mais importante, nutre uma cultura de responsabilidade que é compartilhada por todos. Essa nova forma de pensar quebra as barreiras antigas entre os times, incentivando que desenvolvedores, testadores, analistas, administradores de sistemas e até os profissionais de segurança trabalhem lado a lado. Hüttermann (2012) reforça que, com o DevOps, cresce um senso de união e comprometimento em grupo, onde as pessoas sempre vêm antes dos processos e das ferramentas. Sato (2014) ressalta de forma contundente que o DevOps tem como objetivo superar as lacunas históricas entre desenvolvimento e operações. Isso é alcançado ao reunir profissionais com habilidades diferentes, porém complementares, em uma equipe coesa e multidisciplinar. Esse alinhamento estratégico não é apenas teórico — ele se traduz em entregas mais rápidas, menos erros e mais tempo dedicado à melhoria contínua. Pense nisso: mais tempo para testes aprofundados, ajustes de desempenho e refinamento das funcionalidades. Geerling (2015) complementa que essa colaboração impacta diretamente na qualidade e estabilidade dos sistemas, além de promover um ambiente de trabalho mais unido e eficiente.

O movimento DevOps ganhou notoriedade a partir de 2008, durante a Conferência Ágil de Toronto, no Canadá, quando Patrick Debois e Andrew Shafer discutiram informalmente a aplicação de princípios ágeis à infraestrutura. Em 2009, durante a conferência *Velocity*, John Allspaw e Paul Hammond apresentaram o caso da empresa Flickr, que realizava até dez implantações por dia com colaboração estreita entre desenvolvimento e operações, demonstrando os benefícios da integração contínua e da responsabilidade compartilhada (Kim et al., 2018).

Um dos exemplos mais marcantes da eficácia da cultura DevOps é o da Salesforce. Inicialmente voltada para serviços de CRM, a empresa começou a enfrentar sérias dificuldades para acompanhar o ritmo das entregas. Em 2007, conseguiu fazer apenas um grande lançamento, mesmo com uma equipe de engenharia maior do que no ano anterior, quando haviam ocorrido quatro. Diante desses desafios, a Salesforce iniciou, em 2009, uma transformação profunda baseada em práticas ágeis e DevOps. Até 2013, o tempo necessário para implantar novas

versões caiu de seis dias para apenas cinco minutos. Isso só foi possível graças à automação de testes — incluindo testes funcionais e destrutivos —, ao uso de ferramentas como o *Rouster* e à colaboração mais próxima entre as equipes de desenvolvimento, operações e segurança da informação (Kim et al., 2018). Com a automação assumindo o lugar de processos manuais e burocráticos, a empresa ganhou eficiência, reduziu riscos e aumentou significativamente a qualidade do serviço entregue.

A adoção de DevOps, portanto, não é apenas uma mudança técnica, mas uma transformação na maneira como o *software* é desenvolvido e mantido. Com seus princípios bem incorporados, as organizações passam a ter a capacidade de realizar centenas ou até milhares de implantações por dia, de forma mais previsível e confiável. Isso fortalece a inovação, a capacidade de adaptação ao mercado e a entrega contínua de valor ao cliente (Kim et al., 2018).

2.3 Plataformas de Borda *(Edge Computing)*

O crescimento exponencial na geração de dados por dispositivos inteligentes e a evolução tecnológica em ambientes industriais, como a Indústria 4.0, tornaram o processamento de informações próximo à sua origem uma necessidade crucial (Shi et al., 2016). A arquitetura tradicional de Computação em Nuvem (*Cloud Computing*), embora escalável, enfrenta limitações em cenários que exigem baixa latência e alta disponibilidade para dados gerados na periferia da rede. Nesse cenário, o paradigma de Computação de Borda emerge como um componente essencial.

2.3.1 Conceito e Arquitetura

A Computação de Borda é um modelo de processamento distribuído que estende as capacidades da Nuvem para a borda da rede, posicionando recursos de armazenamento, processamento e rede mais próximos de sensores, atuadores e usuários finais (Cao et al., 2020). Essa abordagem descentralizada visa minimizar a transferência de grandes volumes de dados para *data centers* distantes, atuando como um intermediador entre os dispositivos IoT e a infraestrutura central na Nuvem.

A arquitetura de comunicação da Computação de Borda é frequentemente organizada em três camadas distintas, garantindo o isolamento e a otimização do fluxo de dados (Cao et al., 2020; Rios, 2022):

1. Camada de Terminais (Dispositivos IoT): Engloba os dispositivos conectados, como sensores de rede de energia e atuadores industriais, que são os fornecedores ativos das informações.
2. Camada de Barreira (Nós de Borda/Névoa): É a camada intermediária, composta por *gateways*, *switches* e microcomputadores (*Raspberry Pi* ou servidores locais), que realiza o processamento local, a comunicação e o isolamento entre os terminais e a Nuvem.
3. Camada de Nuvem (*Cloud*): É o extremo oposto, onde se encontram servidores e dispositivos de armazenamento com alta capacidade computacional, reservada para análises complexas e armazenamento de longo prazo (Cao et al., 2020).

2.3.2 Características e Vantagens Estratégicas

A posição estratégica da Borda confere-lhe atributos que são vitais para o funcionamento de sistemas críticos e ambientes de IoT de grande escala, como o monitoramento de energia:

- Baixa Latência e Interações em Tempo Real: Por estar fisicamente próxima dos dispositivos IoT, a Borda reduz a distância de comunicação, diminuindo drasticamente a latência e permitindo interações em tempo real essenciais para o controle e monitoramento de ativos (Yi et al., 2015).
- Distribuição Geográfica e Escalabilidade: Os nós de Borda podem ser implantados em locais distintos (percepção de localização) para atender à demanda de uma rede de dispositivos inteligentes dispersos, oferecendo recursos de processamento e armazenamento distribuídos de forma escalável (Atlam et al., 2018).
- Otimização de Largura de Banda e Segurança: O processamento local preserva a largura de banda, transferindo apenas dados processados para a *cloud*. Além disso, a proximidade física e a capacidade de operar desconectada de sistemas externos facilitam a implementação de políticas de segurança mais robustas para dados sensíveis (Shi et al., 2016).
- Heterogeneidade e Interoperabilidade: A Borda é projetada para gerenciar e interligar diferentes dispositivos finais e protocolos, garantindo que componentes de fabricantes distintos possam se comunicar de forma eficaz (Yi et al., 2015).

2.3.3 Conexão com a Infraestrutura como Código

Apesar das vantagens, o gerenciamento de plataformas de borda é inerentemente complexo devido à distribuição geográfica e à heterogeneidade de *hardware* e *software* nos nós de barreira. No contexto desta pesquisa, o ambiente de borda, simulado por Máquinas Virtuais (VMs) que replicam ***Smartservers*** de coleta de energia, exige que o provisionamento e a configuração sejam repetíveis, consistentes e rápidos. A configuração manual desse *cluster* para a instalação de componentes essenciais (como *addons* de comunicação e coleta de dados) introduz risco de erros e atrasos. Portanto, o gerenciamento eficiente e em escala da Plataforma de Borda depende criticamente da adoção da abordagem *Infrastructure as Code*, utilizando ferramentas que permitam a automação completa do ciclo de vida da infraestrutura.

2.4 Infraestrutura como Código (IaC)

A automação, técnica que elimina a interferência humana em processos, assume um papel central na gestão de ambientes de TI, sendo a resposta mais eficaz para a crescente complexidade das infraestruturas. A intervenção manual no gerenciamento de recursos não só compromete a agilidade na implantação de *software* (BROWN; HELLERSTEIN, 2005), mas também é a causa principal da grande maioria dos problemas de disponibilidade em grandes *data centers* (OPPENHEIMER; GANAPATHI; PATTERSON, 2003).

Para Nelson-Smith (2013), a infraestrutura como código teve início em 2006, quando a *Amazon Web Services* lançou o *Elastic Compute Cloud* (EC2). A partir daquele momento, qualquer ideia para uma aplicação *web* poderia ser implementada em questão de semanas, fazendo com que pequenas empresas tivessem um crescimento acelerado, e mesmo sendo lideradas por desenvolvedores, tiveram que se preocupar com rotinas de equipes de operações, como adicionar máquinas idênticas, fazer *backup*, escalar banco de dados, entre outras. Como estas equipes eram pequenas e tinham que administrar ambientes complexos, começaram a surgir as primeiras ferramentas de gestão de configuração.

2.4.1 Evolução e Definição do Conceito

Para compreender o IaC, é fundamental distinguir seus escopos. Segundo Humble e Farley (2013), o ambiente é o conjunto de recursos (configuração de *hardware*, rede, sistema operacional e *middlewares*) que uma aplicação requer para rodar. Já a infraestrutura representa

todos os ambientes de uma organização e os serviços que os apoiam. A gestão eficiente dessa infraestrutura se tornou um desafio exponencial na era da virtualização, *containers* e Computação em Nuvem (Red Hat, 2020).

A capacidade de provisionar recursos sob demanda em um tempo drasticamente reduzido (Morris, 2016) elevou a demanda por rotinas de operação mais eficientes. Morris (2016) marca essa transição como a passagem da “idade do ferro”, na qual era baseada em *hardware* físico e trabalho manual para a “idade da nuvem” que é caracterizada pela abstração do *hardware* e pelo gerenciamento programático da infraestrutura.

Infraestrutura como Código pode ser definida como a prática de gerenciar e provisionar infraestrutura por meio de código, em vez de processos manuais (Red Hat, 2020). IaC consiste no uso das práticas do desenvolvimento de *software* para definir, implementar, atualizar e destruir a infraestrutura, integrando essa abordagem à cultura DevOps (Brikman, 2022). O IaC transforma todo o conjunto de *scripts*, modelos, dependências e configurações em código-fonte, garantindo que o ambiente seja executável de forma eficiente e repetitiva (Artač et al., 2017; Hüttermann, 2012).

No modelo tradicional de gerenciamento de infraestrutura, um desenvolvedor que necessitava de um novo recurso, como uma VM, abria uma solicitação (*ticket*). Um operador de infraestrutura então atendia a essa solicitação, acessando um console de administração e realizando um processo manual de "apontar e clicar"(*point-and-click*) para provisionar o recurso. Esse modelo era funcional em ambientes de baixa rotatividade (*churn*), onde as VMs tinham um ciclo de vida de meses ou anos (HashiCorp, 2025).

A transição para a Computação em Nuvem tornou esse modelo obsoleto por três motivos principais (HashiCorp, 2025):

1. Ambientes *API-driven*: A Nuvem é predominantemente controlada por APIs, permitindo a automação.
2. Alta Elasticidade: O ciclo de vida dos recursos mudou de meses para semanas ou até mesmo dias.
3. Natureza Cíclica e Escalável: As infraestruturas precisam escalar (aumentar) para picos de demanda e encolher (diminuir) em períodos ociosos para otimização de custos, tornando o gerenciamento manual por *tickets* inviável.

Nesse novo paradigma, o IaC surge como a solução para capturar o processo de “apontar e clicar” de forma codificada (HashiCorp, 2025). A prática consiste no uso de código

para definir, implementar, atualizar e destruir a infraestrutura (Brikman, 2022), alinhando-se à cultura DevOps (Red Hat, 2020). O IaC transforma *scripts*, modelos e configurações em código-fonte, garantindo que o ambiente seja executável de forma eficiente e repetitiva (Artač et al., 2017; Hüttermann, 2012).

2.4.2 Benefícios da Infraestrutura como Código

A adoção do IaC permite que a infraestrutura seja gerenciada de forma segura, consistente, rastreável e repetível (HashiCorp, 2025). Os principais benefícios incluem (Nelson-Smith, 2013; Kim et al., 2018; Saito et al., 2017):

- Agilidade e Velocidade (Automação): O uso de código-fonte e controle de versão permite avançar ou retroceder a estados conhecidos da infraestrutura com rapidez, otimizando o tempo de implantação e permitindo que as equipes respondam rapidamente às necessidades do negócio.
- Repetibilidade e Replicação: O código garante que a recriação do ambiente seja idêntica em qualquer instância, eliminando o erro humano inerente aos processos manuais (Saito et al., 2017).
- Consistência e Rastreabilidade (Versionamento): O versionamento do código (Kim et al., 2018) permite o rastreamento das alterações (transparência), garantindo a uniformidade dos ambientes, o que é vital em plataformas distribuídas, como a Borda.
- Escalabilidade e Descarte de Ambientes: A modularidade e a automação facilitam o aumento da quantidade de servidores e permitem que a infraestrutura seja tratada como dinâmica, podendo ser facilmente destruída e recriada (*disposable*).
- Recuperação de Desastres: Ter a infraestrutura definida em código facilita a restauração completa do ambiente em caso de falha catastrófica (*Disaster Recovery*).
- Reusabilidade: O código pode ser compartilhado e reutilizado, padronizando a implantação de componentes (HashiCorp, 2025).

2.4.3 Princípios da Infraestrutura como Código

A codificação de infraestruturas complexas é uma tarefa desafiadora. A simples automação sem a adesão a um conjunto de princípios pode levar a problemas comuns, como a duplicação de código, alterações desorganizadas e o risco de mudanças triviais causarem efeitos colaterais catastróficos (Nelson-Smith, 2013).

Para mitigar esses riscos, Morris (2016) e Nelson-Smith (2013) estabelecem os seguintes pilares de design e desenvolvimento:

- Idempotência (Repetibilidade): Talvez o princípio técnico mais crítico. A idempotência assegura que uma configuração possa ser executada diversas vezes no mesmo *host*, mas as alterações só serão aplicadas caso sejam necessárias (Sato, 2014). Isso garante que o sistema sempre convirja para o estado desejado.
- Abordagem Declarativa: As especificações dos serviços devem focar no estado final desejado, tais quais como o "o quê" a infraestrutura deve ser, e não na sequência exata de comandos necessários para "como" alcançá-lo.
- Consistência: Elementos da infraestrutura com a mesma função devem ser funcionalmente idênticos. Qualquer mudança pós-provisionamento deve ser tratada e inserida como parte do código, evitando o *configuration drift* (desvio de configuração).
- Ambientes Descartáveis (*Disposable*): A infraestrutura deve ser dinâmica e projetada para ser facilmente criada, destruída ou substituída, o que aumenta a tolerância a falhas (Morris, 2016).
- Modularidade e Composição: Os serviços devem ser projetados de forma modular (pequenos e simples) para que possam ser integrados como "tijolos de construção" (building blocks) na criação de sistemas complexos, promovendo a reutilização de código (Nelson-Smith, 2013).
- Convergência: Os serviços devem estar com seu estado alinhado às suas políticas de configuração, convergindo para o funcionamento de um sistema geral.

2.4.4 Práticas Recomendadas de Implementação

A aplicação bem-sucedida dos princípios de IaC depende da adoção de práticas de engenharia de *software* no gerenciamento da infraestrutura. Riti (2018) e HashiCorp (2025) citam as seguintes práticas essenciais:

- Definir toda a infraestrutura em código: Tudo o que é necessário para o ambiente (DNS, espaço em disco, sistema operacional) deve ser definido em arquivos de código para tornar a infraestrutura programática e repetitiva.
- Versionar o código: O uso de um sistema de controle de versão, como o Git, é fundamental. Ele substitui a "tradição oral" (o conhecimento informal de "o que apontar e clicar") por um histórico incremental de quem mudou o quê (HashiCorp, 2025). Isso garante transparência,

documentação e a capacidade de reverter para estados anteriores (Riti, 2018).

– Realizar pequenas alterações (Iteração): Em vez de grandes mudanças em lote, deve-se priorizar alterações pequenas e frequentes, o que facilita o isolamento de problemas e a rápida correção.
– Testar o código: Como a infraestrutura é definida por código, ela pode (e deve) ser testada antes de ser aplicada em produção, aumentando a confiabilidade.

2.4.5 Categorização das Ferramentas e Componentes

O universo das ferramentas de IaC pode ser segmentado de acordo com a função que exercem no ciclo de vida da infraestrutura (Brikman, 2022; Instruct, 2017). Para fundamentar a arquitetura híbrida adotada neste trabalho, a Tabela 1 categoriza essas ferramentas e detalha como cada uma foi aplicada no projeto: o Ansible assumindo as camadas de infraestrutura base e configuração, enquanto o Terraform orquestra o deployment da aplicação.

Tabela 1 – Categorização das Ferramentas de IaC e Aplicação no Projeto

Categoria	**Descrição**	**Aplicação no Projeto de Borda**
Provisionamento (Infra)	Criação dos recursos subjacentes (VMs, redes).	Provisionamento do *cluster* de VMs com **Ansible**.
Gerência de Configuração	Instalação e gerenciamento de software em servidores já existentes.	Configuração do ambiente (Docker, K3D, Poetry) com **Ansible**.
Provisionamento (Aplicação)	Implantação de componentes de software (ex: *addons*, *deployments*).	*Deploy* dos *addons* e da aplicação na borda com **Terraform**.

Fonte: Elaborado pelo autor (2025), baseado em Brikman (2022) e Instruct (2017).

2.5 Ferramentas de Automação para IaC

Enquanto a seção anterior definiu os conceitos de IaC, esta seção detalha as ferramentas específicas escolhidas para implementar a automação da Plataforma de Borda: **Terraform** e **Ansible**. A escolha dessas duas ferramentas reflete a combinação estratégica de Provisionamento e Gerência de Configuração/Orquestração.

2.5.1 Terraform: Provisionamento Declarativo

O Terraform é uma ferramenta de código aberto (*open source*) desenvolvida pela HashiCorp, que permite a definição, o provisionamento e o gerenciamento do ciclo de vida da infraestrutura de forma modular e reutilizável (Brikman, 2019). Lançado em 2012 e desenvolvido em linguagem Go, o Terraform permite construir, modificar e versionar a infraestrutura de forma segura e eficiente.

A principal característica do Terraform é sua abordagem declarativa. A ferramenta utiliza uma linguagem própria, a *HashiCorp Configuration Language* (HCL), em arquivos com a extensão .tf. A HCL permite ao usuário focar no estado final desejado (o "o quê"), em vez de escrever a sequência de comandos para alcançá-lo (o "como") (Sabharwal; Pandey; Pandey, 2021).

2.5.1.1 Fluxo de Trabalho e Comandos Principais

O Terraform opera através de um fluxo de trabalho claro, baseado em comandos que gerenciam o ciclo de vida da infraestrutura (Brikman, 2019):

- ***terraform init*:** É o primeiro comando a ser executado em um diretório de trabalho. Ele inicializa o ambiente, baixando os plugins necessários (chamados *providers*) para interagir com as APIs dos provedores de nuvem ou serviços especificados (como AWS, Azure, ou, no caso deste projeto, o provedor local de virtualização).
- ***terraform plan*:** Este comando gera um plano de execução. Ele compara a infraestrutura atual (definida no arquivo de estado) com as configurações presentes nos arquivos .tf (o estado desejado). O resultado é uma lista detalhada de todas as ações que o Terraform tomará: quais recursos serão criados (adições), alterados (modificações) ou destruídos (remoções).
- ***terraform apply*:** Após a revisão do plano, este comando aplica as mudanças planejadas. Ele executa as ações descritas para que a infraestrutura real convirja para o estado definido no código.
- ***terraform destroy*:** Este comando remove toda a infraestrutura gerenciada pelo Terraform. Ele é particularmente útil para eliminar ambientes temporários, de teste ou de desenvolvimento, garantindo que os recursos sejam liberados e evitando custos desnecessários (Brikman, 2019).

2.5.1.2 O Arquivo de Estado (State File)

Um componente crítico do Terraform é o Arquivo de Estado (geralmente *terraform.tfstate*). Cada vez que o Terraform executa uma aplicação (*apply*), ele registra informações sobre a infraestrutura criada neste arquivo.

O state utiliza um formato *JavaScript Object Notation* (JSON) personalizado que mapeia os recursos definidos no código para os recursos reais na plataforma (seja na nuvem ou *on-premise*). Ele armazena informações detalhadas, como IDs de instâncias, endereços de IP e dependências entre recursos (Brikman, 2019). É este arquivo que o *terraform plan* utiliza como fonte da verdade sobre o "estado atual", permitindo que a ferramenta determine quais mudanças são necessárias.

2.5.1.3 Módulos e Reutilização

Para promover os princípios de IaC de modularidade e reutilização, o Terraform utiliza Módulos. Um módulo é um agrupamento de recursos (.tf arquivos) que pode ser tratado como um único "bloco de construção". Módulos podem ser compartilhados entre diferentes projetos ou equipes, garantindo consistência, reduzindo a duplicação de código e minimizando a probabilidade de erros.

2.5.2 Ansible: Gerência de Configuração e Orquestração

O **Ansible** é uma plataforma *open source* de automação de TI que atua no provisionamento, gerenciamento de configuração e implantação de aplicações. Concebido originalmente em 2012 por Michael DeHaan e desenvolvido em Python, o Ansible foi adquirido pela Red Hat em 2015, tornando-se uma referência no mercado devido à sua simplicidade e poder de escala (Heap, 2016). Atualmente, conta com uma vasta comunidade de colaboradores, sendo desenhado para suportar ambientes complexos, desde redes locais até nuvens híbridas e contêineres.

2.5.2.1 Arquitetura e Funcionamento

Diferente de outras ferramentas de automação que exigem a instalação de *software* cliente em cada máquina (arquitetura *agent-based*), o Ansible se destaca por sua arquitetura ***agent-less*** (sem agentes). Ele utiliza o protocolo SSH (*Secure Shell*) padrão para se comunicar com os sistemas remotos, eliminando a necessidade de gerenciar *daemons* ou agentes adicionais

nas máquinas-alvo. Essa abordagem reduz a complexidade de manutenção, simplifica o uso e aumenta a segurança e a compatibilidade com múltiplos sistemas operacionais (Heap, 2016).

O modelo de operação do Ansible é baseado no método ***Push***: a configuração é "empurrada"de um nó central para os *hosts* gerenciados.

A arquitetura básica é composta por dois elementos principais (Chang et al., 2016):

- **Nó de Controle (*Control Node*):** É a máquina onde o Ansible está instalado e de onde os comandos e *playbooks* são executados. Pode ser a estação de trabalho de um administrador ou um servidor de CI/CD.
- **Hosts Gerenciados (*Managed Hosts*):** São os servidores que recebem as configurações. Eles são organizados em um **Inventário**.

O **Inventário** é um componente fundamental que lista e agrupa os *hosts* sobre os quais o Ansible atuará. Ele pode ser definido de forma **estática** (arquivos de texto INI ou YAML com IPs e grupos fixos) ou **dinâmica** (*scripts* que consultam provedores de nuvem ou virtualizadores para obter a lista de máquinas em tempo real) (Chang et al., 2016).

2.5.2.2 Playbooks e Linguagem YAML

A automação no Ansible é descrita utilizando **YAML** (*YAML Ain't Markup Language*), um formato de serialização de dados legível por humanos, focado na simplicidade de leitura e escrita (Fonseca; Simoes, 2007).

Os arquivos principais de automação são denominados ***Playbooks***. O *Playbook* é o objeto de mais alto nível que define o **estado desejado** do sistema, contendo uma ou mais *Plays*. Cada *Play* mapeia um grupo de *hosts* a uma lista de tarefas (*tasks*) (Hochstein; Moser, 2017).

A anatomia de um projeto Ansible envolve os seguintes conceitos-chave:

- ***Tasks* (Tarefas):** São as menores unidades de ação. Cada *task* chama um ***Module*** específico para realizar uma operação (ex: instalar um pacote, copiar um arquivo, reiniciar um serviço).
- ***Modules* (Módulos):** São os programas "binários de ação" que o Ansible envia para o *host* remoto para executar as tarefas. O Ansible possui centenas de módulos nativos para interagir com sistemas Linux, Windows, redes e APIs de Nuvem.
- ***R*oles (Papéis):** Para promover a reutilização e organização, as *Roles* permitem agrupar *tasks*, variáveis, arquivos e *templates* em uma estrutura de diretórios padronizada. Isso facilita o compartilhamento de automações complexas entre diferentes projetos.

– **Idempotência:** Um princípio central do Ansible. A maioria dos módulos verifica o estado atual do sistema antes de agir. Se o sistema já estiver no estado desejado (ex: o pacote “Docker” já está instalado), o Ansible não realiza nenhuma alteração. Isso garante segurança na execução repetida de *playbooks* (Pilla, 2020).

2.5.2.3 Aplicação na Plataforma de Borda

No contexto deste trabalho, o Ansible desempenha um papel híbrido e fundamental para orquestração do ambiente. Atuando por meio de *playbooks* em duas etapas sequenciais. Ele é utilizado primeiramente na instalação dos *requirements* — como Docker, K3D CLI e Terraform - garantindo que todas as dependências e ferramentas de linha de comando estejam presentes para o **provisionamento da infraestrutura**. Em seguida, executa o **provisionamento do *cluster***, automatizando a criação e configuração do ambiente de borda sobre a infraestrutura previamente preparada.

Dessa forma, garantindo que todas as instâncias do ambiente de desenvolvimento estejam idênticas e prontas para receber a aplicação (que será implantada posteriormente via Terraform).

3 MATERIAIS E MÉTODOS

3.1 Tipo de Pesquisa

Para a realização deste trabalho, a metodologia foi estruturada com base em critérios de natureza, objetivos e procedimentos técnicos, conforme detalhado nas subseções a seguir.

3.1.1 Definição da pesquisa quanto aos seus objetivos

Os objetivos de pesquisa deste trabalho enquadram-se como **exploratórios** e **descritivos**. É classificada como **exploratória**, pois busca promover a familiaridade e aprimorar as ideias sobre o tema central: a viabilidade da automação completa de plataformas de borda com a combinação de Terraform e Ansible (Botelho e da Cruz, 2013). Segundo Gil (2002), estas pesquisas têm como foco aprimorar ideias ou descobrir intuições. É também **descritiva**, pois o objetivo é descrever detalhadamente o processo de implementação, o fluxo de automação desenvolvido e a relação entre as variáveis analisadas.

3.1.2 Definição da pesquisa quanto à natureza da abordagem

Este trabalho utiliza uma abordagem de pesquisa de natureza **quali-quantitativa**. O componente **qualitativo** se manifesta na análise dos benefícios da adoção da Infraestrutura como Código, buscando entender o fenômeno em profundidade (Botelho e da Cruz, 2013). O componente **quantitativo** é utilizado na fase de análise dos resultados, com foco na medição de variáveis específicas, como a comparação do tempo de provisionamento entre o método manual e o automatizado.

3.1.3 Definição da pesquisa quanto aos procedimentos técnicos

O procedimento técnico utilizado neste estudo é classificado como **Estudo de Caso**, sendo complementado pela **Pesquisa Bibliográfica**. A Pesquisa Bibliográfica, conforme apresentado no Capítulo 2, serviu de base para o embasamento teórico. Já o **Estudo de Caso** é a estratégia de investigação principal. Para **Robert K. Yin** (2015), o estudo de caso é uma investigação empírica que estuda um fenômeno contemporâneo em profundidade e dentro de seu contexto real. O estudo de caso foi realizado no âmbito do projeto, focando na automação do ambiente de desenvolvimento de uma plataforma de borda, validando a solução de automação

dentro de um cenário de aplicação real e específico.

3.2 Materiais e Tecnologias Utilizadas

Esta seção detalha os recursos de *hardware* e *software* que compuseram o ambiente de estudo de caso, bem como as ferramentas de automação selecionadas para a implementação da abordagem IaC.

3.2.1 Arquitetura do Ambiente de Borda (Materiais)

O ambiente de desenvolvimento simulou a arquitetura de *Edge Computing* (Computação de Borda) na qual a solução de coleta de dados de energia seria implantada. A plataforma de borda real consiste em *Smartservers* dedicados que realizam a aquisição de amostras da rede elétrica.

- **Bancada de Simulação:** Foi utilizada uma bancada física contendo componentes elétricos que simulam a rede da empresa contratante, fornecendo dados em tempo real para o sistema.
- **Hosts de Borda Virtualizados:** As Máquinas Virtuais (VMs) foram utilizadas para simular os *Smartservers* de borda. Estas VMs funcionam como nós de processamento distribuído, responsáveis por executar o *cluster* e os componentes de coleta de dados, alinhando-se aos princípios de baixa latência do *Edge Computing*.
- **Plataforma de Virtualização:** A infraestrutura de virtualização (ex: *VirtualBox*) foi a camada base, sobre a qual o **Ansible** atuou para configurar as VMs.

3.2.2 Ferramentas de Automação e Componentes (Tecnologias)

A automação foi implementada através da combinação estratégica de duas ferramentas centrais do universo IaC, cujas funções se complementam no fluxo de trabalho:

3.2.2.1 Ansible

O Ansible foi empregado para a etapa inicial de **Gerência de Configuração**. Sua arquitetura *agent-less* facilitou a criação do *cluster* de VMs e a instalação de todos os pré-requisitos, garantindo a idempotência no processo de *setup* do ambiente.

3.2.2.2 Terraform

O Terraform foi utilizado para a fase de **Provisionamento da Aplicação**, atuando após o Ansible. Sua natureza declarativa e capacidade de gerenciar o estado da aplicação foram cruciais para o *deploy* consistente dos *addons* (via provedores como *Kubernetes* ou *Helm*) e dos componentes da aplicação sobre o *cluster* de borda.

3.2.2.3 Componentes-Chave da Plataforma

Os seguintes componentes foram instalados e configurados nas VMs de borda para viabilizar a coleta e o processamento de dados:

- **Docker:** Utilizado como motor de *containerização* para isolar e executar a aplicação e os *addons* (Docker Inc., 2024).
- **K3D:** Uma distribuição leve do *Kubernetes* (K8s) para rodar *clusters* em *Docker*. Foi escolhido por sua eficiência e baixa demanda de recursos, ideal para simular o ambiente de borda (K3D Project, 2024).
- **Poetry:** Gerenciador de dependências e empacotamento em Python, utilizado para garantir que o *software* de coleta de dados da aplicação seja executado em um ambiente consistente (Poetry Project, 2024).
- **MQTT Client UI:** Interface ou *client* utilizado para monitoramento e validação da comunicação de dados entre a bancada simuladora e o *cluster* de borda.

3.3 Desenvolvimento da Solução

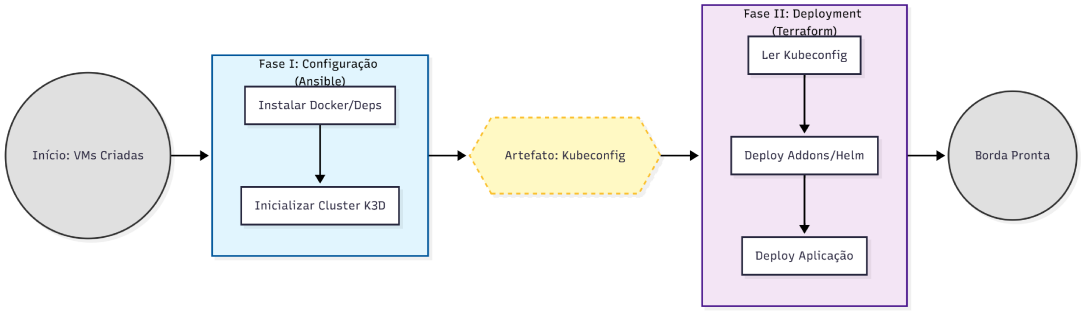
O desenvolvimento da solução de *Infrastructure as Code* (IaC) foi concebido para eliminar a intervenção manual no ciclo de vida da Plataforma de Borda, desde a criação da infraestrutura base até o *deploy* da aplicação. A principal característica da metodologia adotada é a utilização de uma **arquitetura híbrida** que combina o poder de provisionamento do **Ansible** com a capacidade de *deployment* de aplicação do **Terraform**, otimizando o fluxo de trabalho.

Visando a reprodutibilidade deste estudo e seguindo as práticas de *Open Source*, todo o código-fonte desenvolvido para a automação (*scripts* Terraform e *playbooks* Ansible) encontra-se disponível publicamente em um repositório de controle de versão, acessível através do endereço: https://github.com/viniciuscosta12/automacao-borda/.

3.3.1 Arquitetura da Automação Proposta

A arquitetura de automação foi dividida em um fluxo de trabalho sequencial e automatizado, garantindo que as dependências fossem resolvidas antes da execução da próxima etapa. O fluxo demonstra como o provisionamento e o *deployment* são desacoplados entre as duas ferramentas. A Figura 2 ilustra esta sequência de execução (Hochstein; Moser, 2017; Brikman, 2019).

Figura 2 – Fluxo de Automação Híbrida: Ansible e Terraform



Fonte: Elaborado pelo autor (2025).

O fluxo é executado em duas fases principais:

- **Fase I: Configuração e Orquestração (Ansible):** Após a criação manual das Máquinas Virtuais (VMs), o Ansible atua na padronização do ambiente. Ele se conecta aos *hosts* via SSH para instalar e configurar os pré-requisitos essenciais de *software* (Docker, K3D CLI, Poetry) e inicializar o *cluster*.
- **Fase II: Deployment da Aplicação (Terraform):** Após a infraestrutura estar pronta e configurada, o Terraform é acionado. Ele utiliza *providers* específicos (como *Kubernetes* ou *Helm*) para realizar o *deploy* dos *addons* e dos componentes da aplicação sobre o *cluster* K3D, finalizando a entrega de valor.

Essa separação de responsabilidades permite que cada ferramenta utilize sua especialidade (Ansible para sistemas operacionais e Terraform para o ciclo de vida de recursos de alto nível), garantindo eficiência e rastreabilidade em cada etapa do processo.

A seleção das ferramentas de orquestração não foi arbitrária, mas pautada na complementaridade de suas funções e na estratégia de unificação dos ambientes. O Ansible foi eleito para a camada de configuração do sistema operacional devido à sua natureza *agentless*, ideal para preparar nós de borda heterogêneos sem a necessidade de instalar agentes prévios.

Já a escolha do Terraform para o gerenciamento do ciclo de vida do *cluster* e das

aplicações foi motivada, primordialmente, pela necessidade de padronização. Visto que o Terraform já era a ferramenta consolidada para o *deploy* da aplicação central e dos *addons* no cluster da nuvem (*Cloud*), estender seu uso para a borda garantiu uma paridade tecnológica. Essa decisão permitiu reutilizar módulos e lógicas de provisionamento já validados, reduzindo a curva de aprendizado da equipe e mitigando erros de discrepância entre os ambientes de desenvolvimento, nuvem e borda.

3.3.2 Etapa 1: Provisionamento e Configuração do Cluster com Ansible

A primeira fase da automação foi orquestrada integralmente pelo Ansible, dividida em dois fluxos de trabalho sequenciais para garantir a integridade do ambiente. O primeiro fluxo, executado pelo playbook `requirements.yml`, atuou na camada de Gerência de Configuração, padronizando o sistema operacional e instalando as dependências críticas (Docker, K3D CLI, Helm). Uma vez garantida a conformidade do ambiente, o segundo fluxo, com o uso do *playbook* `create_cluster.yml`, foi acionado para efetuar o provisionamento da infraestrutura do cluster Kubernetes, expondo as portas necessárias para a comunicação com a bancada de simulação.

3.3.2.1 Configuração de Dependências (Setup do Ambiente)

O primeiro *Playbook*, denominado `requirements.yml`, assumiu a função de Gerência de Configuração. Ele foi responsável por padronizar o ambiente de execução, garantindo que todas as ferramentas CLI (*Command Line Interface*) e *runtimes* estivessem presentes e nas versões corretas.

As tarefas incluíram a instalação do Docker (motor de execução), K3D, Kubectl, Helm, MQTTX e Poetry. Um ponto estratégico desta etapa foi a instalação automatizada do próprio Terraform via Ansible, preparando o ambiente para a execução da Fase II da automação.

A Figura 3 apresenta o trecho do código responsável pela verificação e instalação do Docker. Para garantir a robustez e a idempotência, foram utilizados identificadores estratégicos de controle de fluxo: a diretiva `register` captura o resultado do comando de verificação na variável `docker_check` as instruções `ignore_errors` e `failed_when` são empregadas para evitar que o playbook seja interrompido caso o Docker não seja encontrado inicialmente. Por fim, a condicional `when` avalia o código de retorno (rc) armazenado, assegurando que a tarefa de instalação seja executada apenas se a ferramenta estiver de fato ausente no sistema.

Figura 3 – Trecho do Playbook Ansible para Instalação do Docker

```yaml
- name: Check if Docker is installed
  shell: command -v docker
  register: docker_check
  ignore_errors: yes
  failed_when: false

- name: Install docker
  apt:
    name: docker.io
    state: present
  when: docker_check.rc != 0

- name: Start and enable the docker service
  systemd:
    name: docker
    state: started
    enabled: yes
```

Fonte: Elaborado pelo autor (2025).

3.3.2.2 Provisionamento do Cluster K3D

Após a preparação das dependências, o *playbook* `create_cluster.yml` assume a responsabilidade pelo provisionamento da infraestrutura. A Figura 4 detalha a lógica de orquestração implementada para esta etapa. Inicialmente, observa-se o padrão de verificação de idempotência nas duas primeiras tarefas: o módulo `command` tenta identificar a versão do `k3d`, registrando o resultado na variável `k3d_installed`; caso o código de retorno indique ausência da ferramenta (`rc != 0`), a condicional `when` autoriza a instalação via *shell script*.

A tarefa principal, 'Create k3d cluster', utiliza o módulo `command` para instanciar o cluster denominado 'app-edge'. Destacam-se três pontos técnicos cruciais neste bloco:

1. **Mapeamento de Portas:** As *flags* `-p` realizam o *bind* das portas do *LoadBalancer* do contêiner (80, 443, 1883) para portas altas no *host* (9080, 9443, 1883), viabilizando a comunicação externa com a bancada simuladora sem conflitos com serviços locais.
2. **Parametrização:** O uso da sintaxe Jinja2 em `{{ num_agents }}` permite que a quantidade de nós agentes seja definida dinamicamente via variáveis, conferindo escalabilidade ao *script*.
3. **Controle de Estado:** O parâmetro `creates` dentro da diretiva `args` atua como um 'guarda', impedindo que o Ansible tente recriar o cluster se o diretório de dados especificado já existir. Isso garante a idempotência da execução, evitando falhas por duplicidade.

Figura 4 – Tarefa Ansible de Criação do Cluster K3D

```yaml
tasks:
  - name: Check if k3d is installed
    command: k3d version
    register: k3d_installed
    ignore_errors: yes

  - name: Install k3d
    shell: |
      curl -s https://raw.githubusercontent.com/k3d-io/k3d/main/install.sh | bash
    when: k3d_installed.rc != 0

  - name: Create k3d cluster
    command: >
      k3d cluster create "app-edge"
      -p "9080:80@loadbalancer"
      -p "9443:443@loadbalancer"
      -p "1883:1883@loadbalancer"
      --agents {{ num_agents }}
    args:
      creates: /var/lib/rancher/k3d/k3s/app-edge # Prevents recreation if the cluster already exists
```

Fonte: Elaborado pelo autor (2025).

3.3.2.3 Extração e Configuração do Acesso (Kubeconfig)

A última etapa crítica realizada pelo Ansible foi a extração e configuração do arquivo `kubeconfig`, essencial para que tanto o Terraform (na fase seguinte) quanto os administradores possam interagir com o cluster. A Figura 5 detalha essa sequência de tarefas de pós-provisionamento.

Inicialmente, o módulo `file` assegura a existência do diretório `~/.kube`. A lógica central reside na captura das credenciais: o módulo `command` executa a instrução `kubectl config view -raw`, armazenando a saída padrão (*stdout*) na variável `kubeconfig_output` através da diretiva `register`.

Em seguida, o módulo `copy` utiliza o conteúdo dessa variável (acessado via sintaxe Jinja2 `{{ }}`) para materializar os arquivos de configuração no disco, enquanto o módulo `file` é reutilizado para ajustar as permissões de proprietário e leitura (modo '0644'). Para garantir a persistência e a usabilidade do ambiente, o módulo `lineinfile` injeta a variável de ambiente `KUBECONFIG` no arquivo `.bashrc` do usuário, e um link simbólico é criado para o binário do `kubectl`, eliminando a necessidade de privilégios de superusuário (*sudo*) para operações rotineiras.

Figura 5 – Tarefa Ansible de Criação do Cluster K3D

```yaml
- name: Check if the ~/.kube directory exists
  file:
    path: "{{ lookup('env', 'HOME') }}/.kube"
    state: directory
    mode: '0755'

- name: Capture kubectl configuration and save to a file
  command: "kubectl config view --raw"
  register: kubeconfig_output

- name: Save kubeconfig to file
  copy:
    content: "{{ kubeconfig_output.stdout }}"
    dest: "{{ lookup('env', 'HOME') }}/.kube/k3d_kubeconfig.yaml"
    mode: '0644'

- name: Save kubeconfig to 'config' file
  copy:
    content: "{{ kubeconfig_output.stdout }}"
    dest: "{{ lookup('env', 'HOME') }}/.kube/config"
    mode: '0644'

- name: Adjust kubeconfig file permissions
  file:
    path: "{{ lookup('env', 'HOME') }}/.kube/k3d_kubeconfig.yaml"
    owner: "{{ lookup('env', 'USER') }}"
    group: "{{ lookup('env', 'USER') }}"
    mode: '0644'

- name: Configure KUBECONFIG environment variable for user
  lineinfile:
    path: "{{ lookup('env', 'HOME') }}/.bashrc"
    line: "export KUBECONFIG={{ lookup('env', 'HOME') }}/.kube/k3d_kubeconfig.yaml"
    create: yes

- name: Ensure kubectl can run without sudo
  file:
    path: "{{ lookup('env', 'PATH').split(':')[0] }}/kubectl"
    state: link
    src: "/usr/local/bin/kubectl"
    mode: '0755'
```

Fonte: Elaborado pelo autor (2025).

3.3.2.4 Estrutura de Tarefas: Preparação do Ambiente

O primeiro *Playbook* (`requirements.yml`) é focado na instalação de dependências. O Quadro 2 detalha as tarefas executadas para garantir que o nó de controle e os nós de borda possuam as ferramentas necessárias.

Tabela 2 – Tarefas do Playbook `requirements.yml` (Configuração)

Tarefa / Componente	**Ação Automatizada**
System Dependencies	Instalação de pacotes base: `curl`, `wget`, `unzip` e `libpq-dev`.
Docker Runtime	Verificação, instalação e habilitação do serviço `docker` no sistema.
K3D & Kubectl CLI	Download e instalação dos binários de linha de comando para gerenciamento do Kubernetes.
Helm Package Manager	Instalação do gerenciador de pacotes Helm para futuros *deployments*.
Terraform CLI	Download, descompactação e instalação do binário do Terraform para a Fase II.
Poetry (Python)	Instalação do gerenciador de dependências e configuração do `PATH`.

Fonte: Elaborado pelo autor (2025).

3.3.2.5 Estrutura de Tarefas: Provisionamento do Cluster

O segundo *Playbook* (`create_cluster.yml`) é responsável pela criação efetiva da infraestrutura do *cluster*. O Quadro 3 resume as etapas de orquestração realizadas.

Tabela 3 – Tarefas do Playbook `create_cluster.yml` (Provisionamento)

Tarefa / Ação	**Ação Automatizada**
Cluster Check	Verificação se o *cluster* já existe para garantir a idempotência.
Create Cluster	Execução do comando `k3d cluster create` com mapeamento de portas (80, 443, 1883).
Kubeconfig Extraction	Extração das credenciais de acesso do *cluster* (`kubectl config view`).
Access Configuration	Criação do arquivo `~/.kube/config` e ajuste de permissões de usuário.
Environment Setup	Exportação da variável `KUBECONFIG` no `.bashrc` para uso imediato.

Fonte: Elaborado pelo autor (2025).

3.3.3 Etapa 2: Provisionamento dos Serviços de Borda e Aplicação com Terraform

A segunda fase da automação é orquestrada pelo **Terraform**. Nesta etapa, a ferramenta assume o controle para realizar o ciclo de vida não apenas da aplicação principal (*App*), mas de todo o ecossistema de serviços de mensageria e integração necessários para a plataforma de borda. O código foi estruturado para realizar não apenas o *deploy*, mas também a orquestração da construção dos contêineres, demonstrando a flexibilidade da ferramenta.

O Terraform foi estruturado de forma modular para gerenciar o *deploy* de quatro componentes críticos, conforme detalhado na documentação do projeto:

1. **Kafka (Strimzi Operator):** Plataforma de streaming de eventos distribuída.
2. **MirrorMaker 2:** Ferramenta para replicação de dados entre o *cluster* de borda e a nuvem.
3. **MQTT Broker (EMQX):** Intermediário de mensagens para dispositivos IoT e sensores.
4. **Edge Compute Apps (App):** A aplicação de negócio para processamento de energia.

3.3.3.1 Configuração dos Provedores e Autenticação

A integração efetiva entre as fases de automação é materializada na configuração dos provedores (*providers*) do Terraform. A Figura 6 demonstra o conteúdo do arquivo `main.tf`, onde se estabelece a conexão com a infraestrutura provisionada anteriormente.

O bloco `provider "kubernetes"` é configurado para ler o arquivo de autenticação em `~/.kube/config` e utilizar o contexto específico `k3d-app-edge`, ambos gerados pelo Ansible. Essa amarração explicita garante que o Terraform gerencie exclusivamente o *cluster* de desenvolvimento criado, mitigando riscos de interação com outros ambientes.

Da mesma forma, o `provider "helm"` é instanciado herdando essas credenciais, habilitando a orquestração de pacotes. Observa-se também a declaração dos provedores auxiliares `local` e `null`, fundamentais para a execução de *scripts* de *build* locais (detalhados na seção seguinte), e o bloco `terraform` que fixa a versão mínima do provedor Helm (`>= 2.1.0`), assegurando a estabilidade e reprodutibilidade da automação.

Figura 6 – Configuração dos Provedores Kubernetes e Helm com Contexto K3D

```
provider "kubernetes" {
  config_path    = "~/.kube/config"
  config_context = "k3d-app-edge"
}

provider "helm" {
  kubernetes {
  config_path    = "~/.kube/config"
  config_context = "k3d-app-edge"
  }
}

provider "local" {}

provider "null" {}


terraform {
  required_providers {
    helm = {
      source  = "hashicorp/helm"
      version = ">= 2.1.0"
    }
  }
}
```

Fonte: Elaborado pelo autor (2025).

3.3.3.2 Deployment dos Serviços de Infraestrutura (Addons)

Antes da implantação da aplicação principal, o Terraform provisiona os serviços de suporte essenciais. A Figura 7 ilustra a declaração de recursos do tipo `helm_release`, utilizados para gerenciar o ciclo de vida dos pacotes Helm (cert-manager e EMQX Operator) dentro do *cluster*.

Para o componente **cert-manager**, responsável pela gestão de certificados TLS, observa-se o uso do bloco `set` para injetar a configuração `installCRDs = true`. Isso é vital para que as *Custom Resource Definitions* (CRDs) sejam criadas automaticamente, evitando falhas de inicialização. Além disso, o parâmetro `create_namespace` automatiza a organização lógica do *cluster*.

Na sequência, declara-se o operador do **MQTT (EMQX)**. O destaque técnico deste bloco é o meta-argumento `depends_on`, que estabelece uma dependência explícita: o Terraform é instruído a aguardar o provisionamento completo do `cert-manager` antes de iniciar a instalação do `emqx-operator`. Essa orquestração previne condições de corrida (*race conditions*) e assegura que a infraestrutura de segurança esteja operacional antes que a camada de mensageria seja

instanciada.

Figura 7 – Exemplo de Recurso *Helm Release* para *Deployment* de Serviços

```
31  resource "helm_release" "cert_manager" {
32    name = "cert-manager"
33
34    repository       = "https://charts.jetstack.io"
35    chart            = "cert-manager"
36    namespace        = "cert-manager"
37    create_namespace = true
38
39    set = [ {
40      name  = "installCRDs"
41      value = "true"
42      type = "auto"
43    } ]
44  }
45
46  resource "helm_release" "emqx_operator" {
47    name = "emqx-operator"
48
49    repository       = "https://repos.emqx.io/charts"
50    chart            = "emqx-operator"
51    namespace        = "emqx-operator-system"
52    create_namespace = true
53
54    depends_on = [ helm_release.cert_manager ]
55  }
56
```

Fonte: Elaborado pelo autor (2025).

3.3.3.3 Orquestração de Build e Dependências

Uma limitação nativa do Terraform é o foco no provisionamento de infraestrutura, não no processo de *build* de *software*. Para superar essa barreira e manter a centralização da automação, utilizou-se o recurso `null_resource` em conjunto com o provisionador `local-exec`.

O `null_resource` atua como um recurso lógico que não gerencia um objeto de infraestrutura real, como uma VM, mas participa do grafo de dependências do Terraform, permitindo o acionamento de gatilhos (*triggers*) arbitrários. Já o `local-exec` é o mecanismo que instrui o Terraform a executar comandos de *shell* diretamente na máquina onde a automação está sendo processada (o *host* local), em vez de executá-los no servidor de destino.

Essa abordagem permitiu injetar a etapa de compilação dos contêineres dentro do

próprio ciclo de vida de provisionamento. Dessa forma, antes de qualquer tentativa de *deployment* no *cluster*, o *script* garante que a versão mais recente da aplicação seja construída e enviada ao *container registry*, que é o repositório que armazena, gerencia e distribui imagens de contêiner de forma segura e versionada, e assim, assegurando que a infraestrutura provisionada reflita sempre o código mais atual.

A Figura 8 ilustra este processo, evidenciando a flexibilidade da ferramenta em integrar tarefas de *build* ao fluxo de infraestrutura.

Figura 8 – Orquestração do *Build* da Aplicação via Terraform

```
resource "null_resource" "deploy_edge_apps" {
  provisioner "local-exec" {
    environment = {
      DOCKER_SERVER      = var.docker_server
      DOCKER_USERNAME    = var.docker_username
      DOCKER_PASSWORD    = var.docker_password
    }

    command = "sudo -E ${var.EDGE_PLATFORM_PATH}/app-edge-platform/scripts/build-containers.sh -d -c ${var.cluster_name}"
  }
}
```

Fonte: Elaborado pelo autor (2025).

3.3.3.4 Deployment via Helm

Após o sucesso do *build* e a criação do *namespace* dedicado, o Terraform utiliza o recurso `helm_release` para implantar a aplicação no *cluster*. A Figura 9 detalha a declaração deste recurso, destacando quatro estratégias de configuração:

1. **Chart Local:** O parâmetro `chart` aponta para um diretório local (`${path.module}/helm/`) em vez de um repositório remoto. Isso permite que a definição da infraestrutura evolua no mesmo *commit* do código da aplicação, facilitando o versionamento.
2. **Injeção de Valores:** A função `file()` é utilizada no bloco `values` para carregar dinamicamente as configurações do arquivo `values.yaml`. Isso desacopla os parâmetros da lógica de implantação.
3. **Resiliência:** O parâmetro `timeout` foi elevado para 600 segundos, garantindo tempo hábil para o *pull* das imagens Docker em ambientes com latência de rede, evitando falhas prematuras.
4. **Orquestração:** O meta-argumento `depends_on` força o Terraform a aguardar a criação do recurso `kubernetes_namespace.app`, assegurando que o *namespace* alvo exista antes de qualquer tentativa de instalação pelo Helm.

Figura 9 – *Deployment* da Aplicação utilizando Recurso *Helm Release*

```
46  resource "helm_release" "app" {
47    name = "app"
48
49    chart               = "${path.module}/helm/"
50    namespace           = "app"
51
52    values = [file("${path.module}/helm/values.yaml")]
53
54    timeout    = 600
55    depends_on = [ kubernetes_namespace.app ]
56  }
57
```

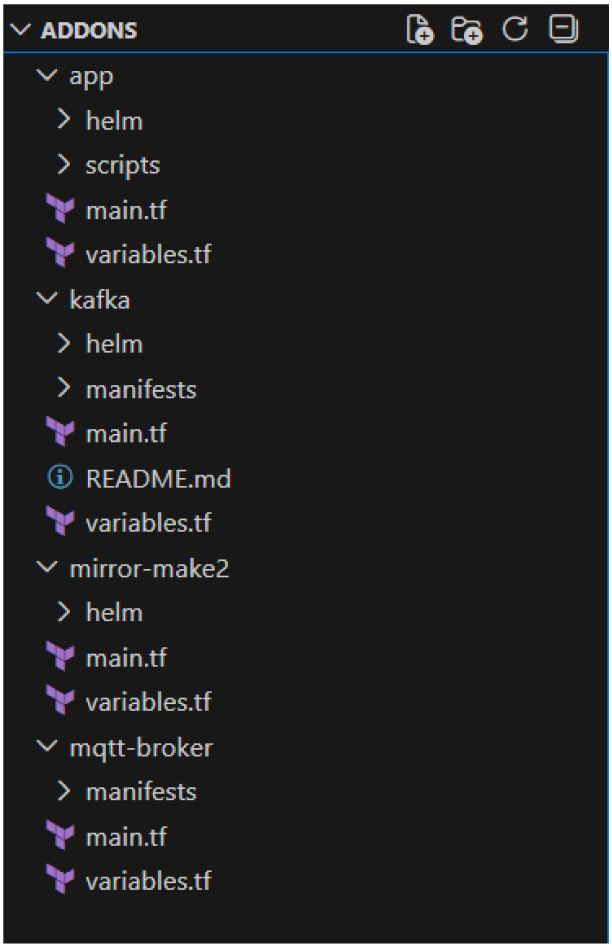
Fonte: Elaborado pelo autor (2025).

3.3.3.5 Estrutura Modular e Escalabilidade

Uma das principais vantagens da abordagem adotada foi a modularização arquitetural dos componentes. O padrão de implementação detalhado anteriormente para a aplicação foi replicado para todos os serviços auxiliares (*addons*) da plataforma, incluindo Kafka, MirrorMaker2 e MQTT Broker.

A Figura 10 detalha a estrutura de diretórios do projeto Terraform, projetada para garantir o **isolamento lógico** de cada serviço. Observa-se que cada componente reside em seu próprio diretório, contendo não apenas os arquivos essenciais de IaC (`main.tf` para recursos e `variables.tf` para parametrização), mas também subdiretórios específicos para seus artefatos: a pasta `helm` armazena os *charts* locais; `manifests` contém definições Kubernetes em YAML puro; e `scripts` guarda automações auxiliares em *Shell*. Essa organização granular assegura que a evolução ou manutenção de um serviço específico não gere efeitos colaterais indesejados no restante da plataforma.

Figura 10 – Estrutura Modular de Diretórios dos *Addons* no Terraform



Fonte: Elaborado pelo autor (2025).

Ao final da execução do comando `terraform apply`, a Plataforma de Borda transita de um estado de infraestrutura vazia para um ambiente operacional completo, com a infraestrutura provisionada e a aplicação de coleta de dados em execução, concluindo o fluxo de automação proposto.

3.3.3.6 Parametrização com uso de variáveis

A flexibilidade e a segurança da solução são asseguradas pela parametrização dos módulos. A Figura 11 ilustra o arquivo `variables.tf` do componente de aplicação, que atua como a interface de entrada de dados para o Terraform.

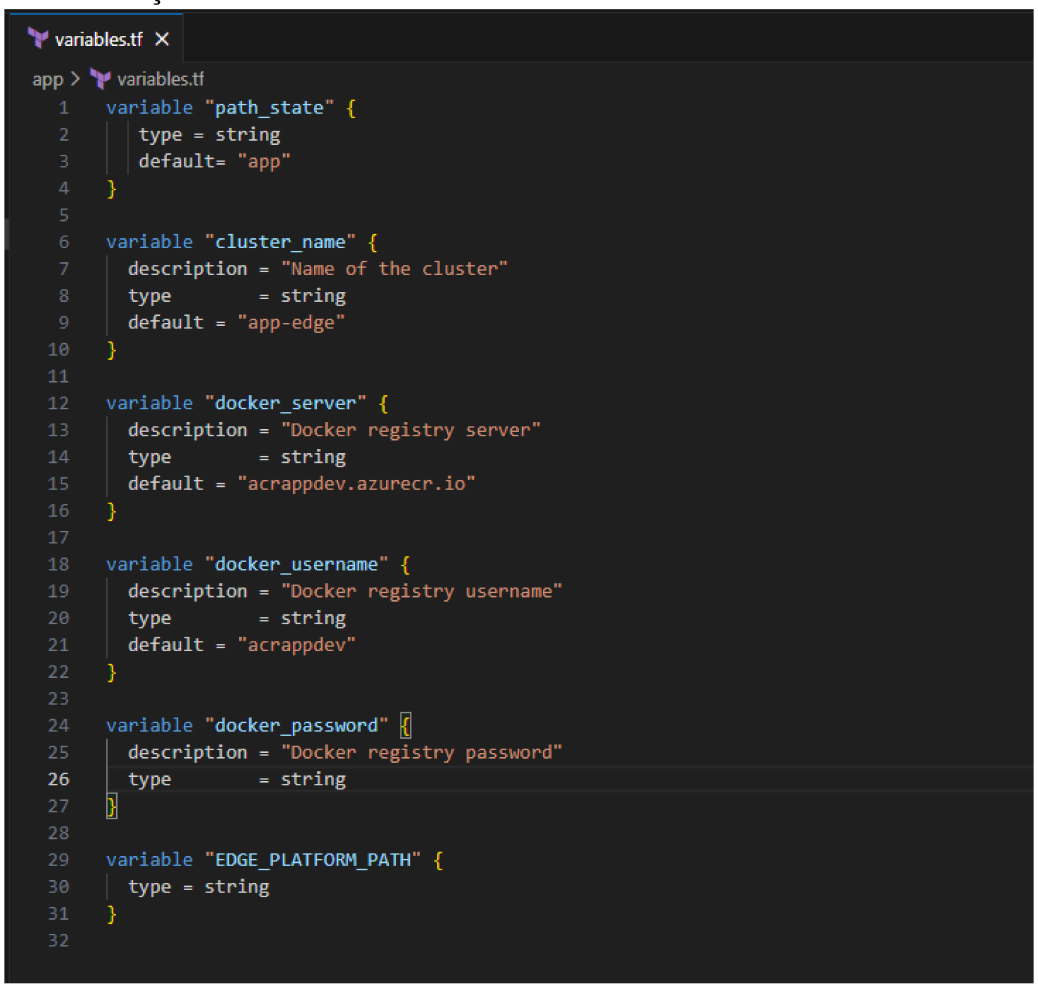
A declaração explícita de variáveis traz três benefícios técnicos imediatos observados no código:

1. **Tipagem Forte:** O uso da propriedade `type = string` impõe restrições de tipo, prevenindo erros de execução causados pela inserção de dados em formatos inválidos.

2. **Segurança de Credenciais:** Variáveis sensíveis, como `docker_username` e `docker_password`, são declaradas mas não necessariamente *hardcoded* (inseridas diretamente) no código. Isso permite que seus valores sejam injetados em tempo de execução via variáveis de ambiente ou arquivos de segredo (`.tfvars`), mantendo as credenciais fora do controle de versão.
3. **Documentação como Código:** O campo `description` fornece contexto imediato sobre o propósito de cada parâmetro, facilitando a manutenção por outros desenvolvedores.

Além disso, variáveis como `cluster_name` e `EDGE_PLATFORM_PATH` garantem que o mesmo código possa ser reutilizado em diferentes ambientes ou caminhos de diretório apenas alterando-se os parâmetros de entrada, sem necessidade de refatoração da lógica principal.

Figura 11 – Definição de Variáveis Parametrizáveis no Terraform



Fonte: Elaborado pelo autor (2025).

3.3.3.7 Execução do Deployment

O gerenciamento do ciclo de vida da aplicação é orquestrado através do *workflow* fundamental do Terraform, composto por etapas sequenciais de inicialização e aplicação. O processo inicia-se com o comando `terraform init`, etapa crítica onde a ferramenta analisa o código para identificar, baixar e configurar os *plugins* dos provedores necessários (neste caso, *Kubernetes* e *Helm*), além de inicializar o *backend* de estado.

Após a preparação do ambiente, a materialização da infraestrutura ocorre através do comando `terraform apply`. Nesta fase, o Terraform constrói um grafo de dependências de todos os recursos declarados e executa as chamadas de API necessárias ao *cluster* K3D para atingir o estado desejado. Para fins de automação total e execução em *scripts* sem interação humana, utilizou-se a *flag* `-auto-approve`, conforme detalhado na sequência de comandos do Código 1.

Código-fonte 1 – Sequência de Comandos para *Deployment*

```
1 # Inicializa os providers (Helm/Kubernetes) e backend
2 $ terraform init
3
4 # Aplica o plano de execucao e realiza o deploy
5 $ terraform apply -auto-approve
```

Após a finalização do fluxo Terraform, a Plataforma de Borda está totalmente operacional. A Figura 12 comprova o sucesso da automação, exibindo o estado dos *Pods* e Serviços rodando no *cluster* K3D, prontos para processar os dados da bancada simuladora.

Figura 12 – Ambiente de Borda Operacional (*Cluster* e Aplicação Ativos)

```
ubuntu@ubuntu:~/tcc/automacao-borda$ kubectl get po -A
NAMESPACE             NAME                                                READY   STATUS      RESTARTS       AGE
cert-manager          cert-manager-7dfcddcdd5-h7xs7                       1/1     Running     1 (159m ago)   10h
cert-manager          cert-manager-cainjector-58d74bf4f5-9lzrq            1/1     Running     0              10h
cert-manager          cert-manager-webhook-6db4c65b5d-v829s               1/1     Running     0              10h
emqx-operator-system  emqx-operator-controller-manager-7667898b57-j454s    1/1     Running     0              10h
kafka                 app-edge-kafka-cluster-entity-operator-bbfcbd8bf-jrlq9   3/3   Running     0              11h
kafka                 app-edge-kafka-cluster-kafka-0                      1/1     Running     0              11h
kafka                 app-edge-kafka-cluster-schema-registry-0            1/1     Running     0              11h
kafka                 app-edge-kafka-cluster-zookeeper-0                  1/1     Running     0              11h
kafka                 mirror-maker-2-mirrormaker2-0                       1/1     Running     90 (19s ago)   10h
kafka                 strimzi-cluster-operator-6f86cd854f-ggnsg           1/1     Running     1 (160m ago)   11h
kube-system           coredns-ccb96694c-q69zp                             1/1     Running     0              11h
kube-system           helm-install-traefik-crd-d9bjw                      0/1     Completed   0              11h
kube-system           helm-install-traefik-nldfd                          0/1     Completed   2              11h
kube-system           local-path-provisioner-5cf85fd84d-lvbnx             1/1     Running     0              11h
kube-system           metrics-server-5985cbc9d7-tdkf5                     1/1     Running     0              11h
kube-system           svclb-emqx-dashboard-f6e4f9c5-6brpn                 1/1     Running     0              10h
kube-system           svclb-emqx-dashboard-f6e4f9c5-cqgcz                 1/1     Running     0              10h
kube-system           svclb-emqx-dashboard-f6e4f9c5-q47dd                 1/1     Running     0              10h
kube-system           svclb-emqx-dashboard-f6e4f9c5-qbt7l                 1/1     Running     0              10h
kube-system           svclb-emqx-listeners-3f0b1dee-8x4hk                 4/4     Running     0              10h
kube-system           svclb-emqx-listeners-3f0b1dee-j9fwj                 4/4     Running     0              10h
kube-system           svclb-emqx-listeners-3f0b1dee-pdjq7                 4/4     Running     0              10h
kube-system           svclb-emqx-listeners-3f0b1dee-prqvp                 4/4     Running     0              10h
kube-system           svclb-traefik-8e8d17c7-4qdlr                        2/2     Running     0              11h
kube-system           svclb-traefik-8e8d17c7-fz6zd                        2/2     Running     0              11h
kube-system           svclb-traefik-8e8d17c7-h6ljt                        2/2     Running     0              11h
kube-system           svclb-traefik-8e8d17c7-qwfdz                        2/2     Running     0              11h
kube-system           traefik-5d45fc8cc9-mpmw8                            1/1     Running     0              11h
mqtt                  emqx-core-6db4dc768d-0                              1/1     Running     0              10h
```

Fonte: Elaborado pelo autor (2025).

3.3.3.8 Documentação como Código e Guia de Uso

Alinhado aos pilares da cultura DevOps, que preconiza a colaboração e o compartilhamento de conhecimento, a solução de automação não se limitou apenas aos *scripts* de execução. Foi desenvolvida uma documentação técnica completa, seguindo o paradigma de Documentação como Código (*Documentation as Code*).

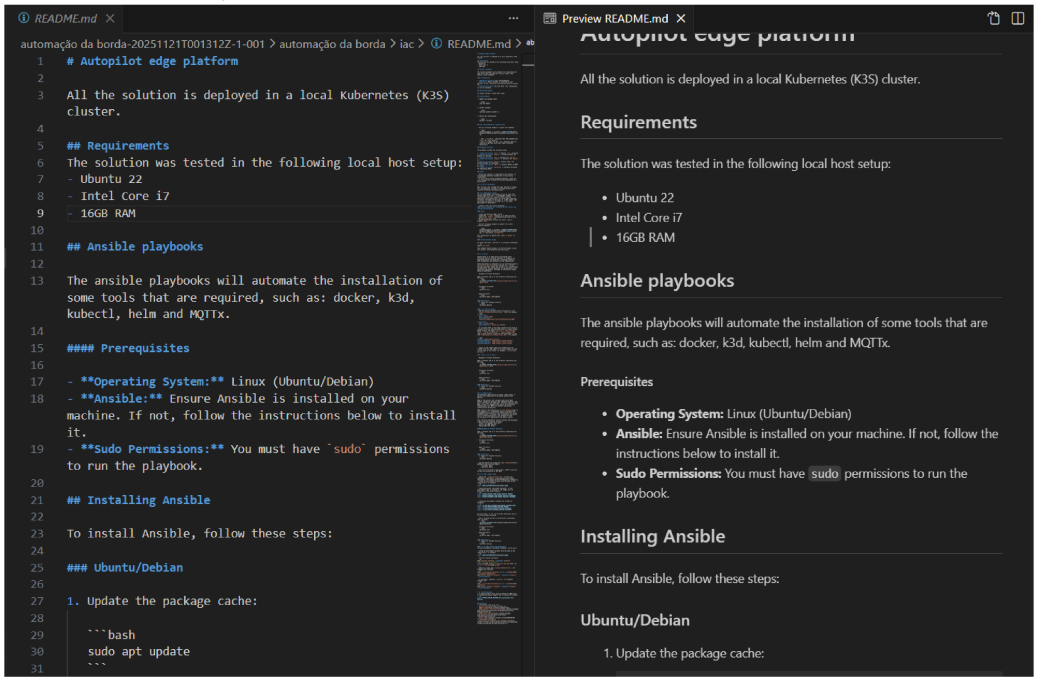
Um arquivo `README.md` foi criado na raiz do repositório do projeto para servir como o ponto de entrada único para desenvolvedores e operadores. Conforme ilustrado na Figura 13, este documento descreve de forma clara:

- **Pré-requisitos do Host:** Especificação do sistema operacional (Ubuntu/Debian) e recursos de *hardware* necessários.
- **Instalação do Ansible:** Instruções passo a passo para preparar a máquina de controle.
- **Execução dos Playbooks:** Comandos exatos para rodar o *setup* de dependências e a criação do *cluster* K3D.
- **Deploy com Terraform:** Guia de navegação entre os módulos (Kafka, MQTT, App) e os comandos de *init* e *apply*.
- **Troubleshooting:** Seção dedicada à resolução de problemas comuns, como erros de chaveiro do Python (*keyring*) ou permissões de *sudo*.

Essa documentação garante que a solução seja sustentável e que novos membros da equipe possam reproduzir o ambiente de desenvolvimento sem depender de conhecimento

prévio.

Figura 13 – Documentação do Ambiente de Desenvolvimento (README.md)



Fonte: Elaborado pelo autor (2025).

4 RESULTADOS

Neste capítulo, são apresentados os principais resultados obtidos com a implementação da automação de provisionamento da Plataforma de Borda. A análise considera os dados técnicos extraídos das execuções automatizadas (*logs* do Ansible e do Terraform) e realiza um comparativo direto com o método manual anteriormente utilizado no projeto.

Os resultados apontam melhorias significativas em aspectos críticos como tempo de provisionamento, padronização da infraestrutura (eliminação de *configuration drift*) e confiabilidade do processo. Também são discutidos os ganhos operacionais obtidos com a modularização dos componentes.

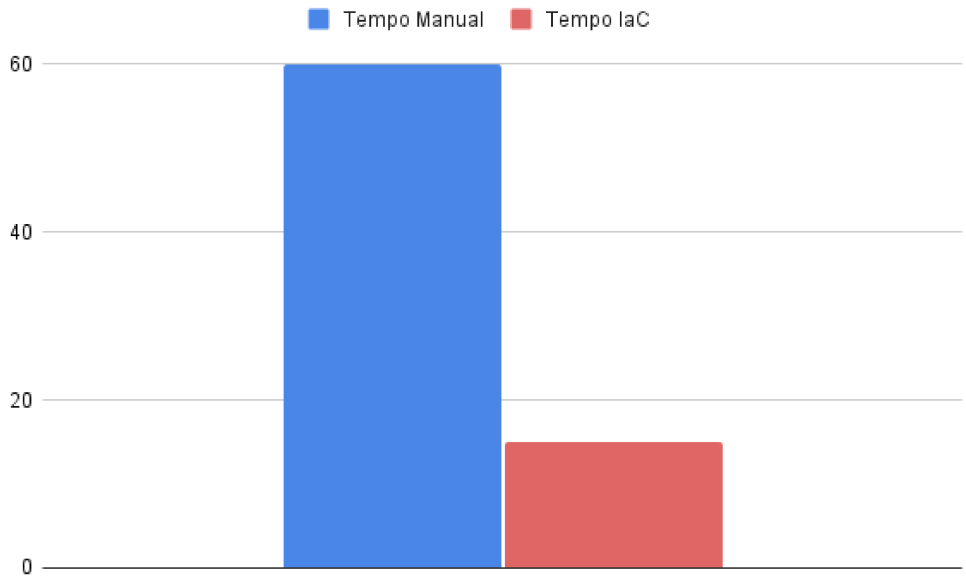
4.1 Desempenho e Otimização no Processo de Provisionamento

Antes da automação, a preparação do ambiente de desenvolvimento envolvia a configuração manual de máquinas virtuais, instalação de dependências e configuração de rede. Relatos do projeto indicam que o *setup* completo de um ambiente funcional podia levar horas, dependendo da familiaridade do operador com as ferramentas e da ocorrência de erros humanos durante a digitação de comandos.

Com a adoção da abordagem de IaC (Ansible + Terraform), esse tempo foi drasticamente reduzido. Após o disparo do comando inicial, o processo de configuração e *deploy* passou a ser executado de forma autônoma.

A Figura 14 apresenta o tempo total de execução registrado em um cenário de teste padrão (criação do *cluster* e *deploy* dos *addons*), totalizando aproximadamente **15 minutos**. A maior parte deste tempo é dedicada ao download de imagens e binários, o que demonstra que o gargalo deixou de ser a operação humana e passou a ser a largura de banda da rede.

Figura 14 – Comparativo de Tempo de Provisionamento (Manual vs. Automatizado)



Fonte: Elaborado pelo autor (2025).

A automação eliminou a latência cognitiva, tempo gasto pensando ou consultando documentação, e a necessidade de intervenções manuais em tarefas repetitivas.

4.2 Impacto na Qualidade e Confiabilidade do Sistema

O processo manual anterior resultava frequentemente em falhas de configuração, onde diferenças sutis entre as máquinas de desenvolvimento (versões diferentes do Docker ou do K3D) causavam erros difíceis de rastrear. A ausência de validações sistemáticas tornava o ambiente frágil.

Após a automação, a confiabilidade do sistema aumentou de forma notável devido a dois fatores principais garantidos pelas ferramentas escolhidas:

- **Idempotência (Ansible):** A garantia de que o *script* pode ser rodado múltiplas vezes sem quebrar o ambiente. Se uma configuração já está correta, ela não é alterada.
- **Gestão de Estado (Terraform):** O arquivo de estado (*tfstate*) garante que o Terraform saiba exatamente o que está rodando no *cluster*, detectando e corrigindo desvios automaticamente.

A Tabela 4 evidencia a mudança de paradigma na qualidade da entrega.

Tabela 4 – Impacto na Qualidade: Método Manual vs. Método IaC

Critério	**Método Manual (Antigo)**	**Método Automatizado (Novo)**
Consistência	Baixa (Varia conforme o operador)	Alta (Definida em código)
Rastreabilidade	Inexistente (Comandos soltos)	Total (Versionamento Git)
Resolução de Erros	Reativa (*Debug* manual)	Proativa (Falha rápida no *script*)
Segurança	Senhas expostas no terminal	Credenciais via variáveis de ambiente

Fonte: Elaborado pelo autor (2025).

4.3 Análise Comparativa dos Resultados Pré e Pós-Implantação

Com o objetivo de mensurar os impactos da automação, foi realizada uma comparação técnica entre os cenários.

Antes da implantação, os principais desafios técnicos incluíam:

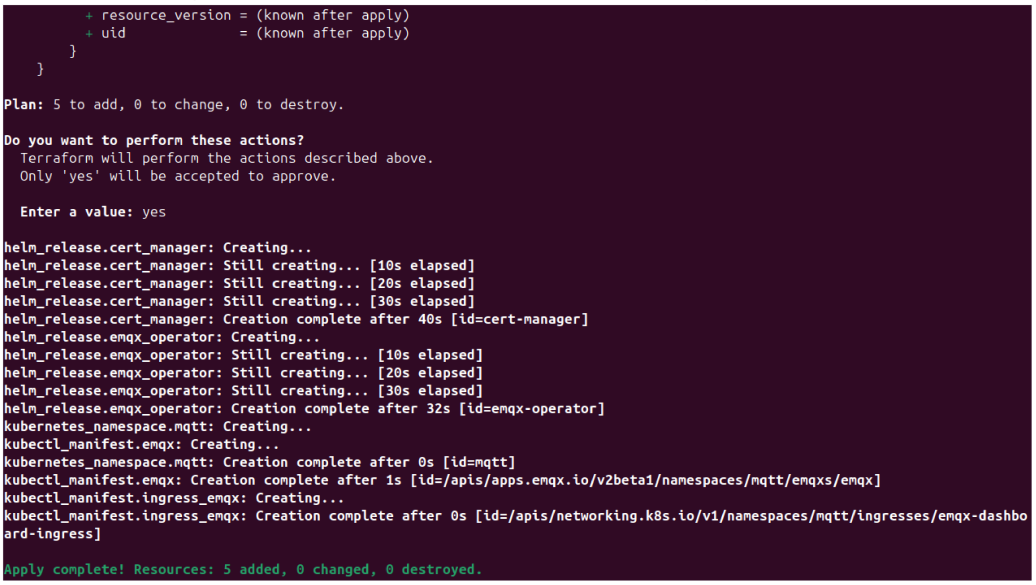
- **Imprevisibilidade:** Não havia garantia de que dois ambientes criados manualmente seriam idênticos.
- **Documentação Obsoleta:** O "passo a passo" manual frequentemente ficava desatualizado em relação ao *software* real.
- **Centralização do Conhecimento Técnico:** Apenas membros específicos da equipe sabiam resolver problemas complexos de instalação.

Após a implantação da solução com Ansible e Terraform, os resultados observados indicam:

- **Redução de 100%** na necessidade de digitação de comandos manuais para configuração de dependências.
- **Reprodutibilidade Garantida:** O ambiente pode ser destruído e recriado em minutos, incentivando testes mais agressivos.
- **Documentação Viva:** O próprio código (*Playbooks* e arquivos *.tf*) serve como documentação atualizada da infraestrutura.

A Figura 15 ilustra a consolidação dos benefícios técnicos alcançados após a execução de um comando bem sucedido.

Figura 15 – Confirmação de Sucesso e Consistência na Execução do Terraform



Fonte: Elaborado pelo autor (2025).

Esses resultados demonstram que a automação teve impacto direto na eliminação de falhas recorrentes de configuração humana, além de promover a padronização necessária para escalar a Plataforma de Borda.

5 CONSIDERAÇÕES FINAIS

Cada vez mais a tecnologia se torna onipresente no cotidiano, impulsionando uma demanda acelerada por sistemas confiáveis e ágeis. Para atender a essa demanda, metodologias de desenvolvimento evoluíram rapidamente, mas a gestão da infraestrutura tradicional frequentemente não acompanhou o mesmo ritmo, gerando gargalos e conflitos entre as equipes de desenvolvimento e operações.

A utilização da metodologia DevOps, central neste trabalho, torna possível que essas equipes trabalhem de forma colaborativa. Com o uso de ferramentas de automação, a infraestrutura deixa de ser um obstáculo estático e torna-se escalável e dinâmica, atendendo com velocidade as necessidades das aplicações de borda.

Em relação aos objetivos específicos propostos na seção 1.3, considera-se que foram plenamente alcançados. Inicialmente, a análise dos requisitos permitiu mapear as necessidades críticas da plataforma de borda, fundamentando a escolha das tecnologias. A abordagem de Infraestrutura como Código foi desenvolvida com sucesso mediante a integração entre Ansible e Terraform, garantindo o provisionamento da aplicação de seus *addons* na borda. A validação da automação ocorreu em ambiente controlado, comprovando a eficácia dos *playbooks* e *scripts* criados. Por fim, a comparação entre os métodos evidenciou a superioridade do modelo automatizado frente ao manual, destacando-se o ganho em desempenho e a mitigação de falhas humanas.

5.1 Conclusões do Estudo

O presente trabalho evidenciou que a automação do provisionamento de uma Plataforma de Borda, utilizando a arquitetura híbrida de **Ansible** e **Terraform**, é uma resposta eficaz aos desafios de latência e padronização.

Entre os diversos benefícios da adoção da IaC observados neste estudo de caso, destacam-se:

- **Eficiência Operacional:** A redução drástica no tempo de provisionamento comprovou que tarefas repetitivas e propensas a erro humano devem ser delegadas a *scripts* automatizados.
- **Recuperação de Desastres e Resiliência:** Uma vez que a infraestrutura completa está definida em código, foi possível evidenciar a capacidade de recuperação rápida. Em caso de falha crítica no ambiente de desenvolvimento, basta reexecutar os *playbooks* para

que, em poucos minutos, a infraestrutura esteja em execução novamente, garantindo a continuidade do negócio.

- **Documentação Viva e Versionamento:** Ao utilizar IaC, a documentação básica do ambiente torna-se automática, visto que o próprio código reflete o estado real da infraestrutura. Manter esse código versionado garante que alterações mal sucedidas não causem danos permanentes, havendo sempre a possibilidade de *rollback* para um estado conhecido e funcional.
- **Idempotência como Garantia de Qualidade:** A repetitividade de execução das ferramentas assegurou que alterações manuais indesejadas (o chamado *configuration drift*) fossem automaticamente corrigidas ou sobrescritas nas execuções subsequentes, mantendo a integridade do ambiente.

Retomando os objetivos específicos, conclui-se que os requisitos de automação foram atendidos, a abordagem modular foi validada com sucesso e a comparação de desempenho favoreceu inequivocamente o modelo automatizado.

5.2 Limitações e Trabalhos Futuros

É importante ressaltar que a validação foi realizada em um ambiente de desenvolvimento virtualizado (VMs). Embora funcional, a execução em *hardware* físico de borda pode apresentar desafios adicionais de conectividade e recursos que não foram explorados neste escopo.

Como sugestão para a continuidade e evolução desta pesquisa, propõe-se:

- **Integração com CI/CD:** Acoplar a solução a uma esteira de Integração Contínua, automatizando o disparo do provisionamento a cada *commit*.
- **Testes em Cenário Real:** Validar a solução em dispositivos físicos (*Smartservers*) para analisar o comportamento em campo.
- **Implementação de Observabilidade:** Incluir o *deploy* de ferramentas de monitoramento para acompanhar a saúde do *cluster* em tempo real.

A Infraestrutura como Código ainda possui um vasto campo para expansão, especialmente em nossa região. Espera-se que este trabalho levante o interesse e fomente discussões a respeito do tema, incentivando profissionais da área de operações a realizarem experimentos e futuras implantações que modernizem o cenário tecnológico local.

REFERÊNCIAS

ARTAč, M.; BOROVšAK, M.; TASIč, G. J. Devops based infrastructure as code in healthcare. In: **2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)**. [*S. l.*: *s. n.*], 2017. p. 422–427.

ATLAM, H. F.; WALTERS, R. J.; WILLS, G. B. Fog computing and the internet of things: A review. **Sensors**, MDPI, Basel, v. 18, n. 2, p. 586, fev. 2018.

BEYER, B.; JONES, C.; PETOFF, J.; MURPHY, N. R. **Engenharia de Confiabilidade do Google: Como o Google administra seus sistemas de produção**. São Paulo: Novatec, 2016.

BOTELHO, J. a. M.; CRUZ, V. A. G. **Metodologia Científica**. São Paulo: Pearson Education do Brasil, 2013.

BRIKMAN, Y. **Terraform: Up & Running**. 2. ed. Sebastopol, CA: O'Reilly Media, 2019.

BRIKMAN, Y. **Infrastructure as Code: An In-depth Guide to IaC Patterns and Practices**. 2nd. ed. [*S. l.*]: O'Reilly Media, 2022.

BROWN, A. F.; HELLERSTEIN, J. L. An evaluation of the auto-bashing approach to configuration management. **Cluster Computing**, v. 8, p. 325–339, 2005.

CAO, K.; LIU, Y.; MENG, G.; SUN, Q. An overview on edge computing research. **IEEE Access**, IEEE, v. 8, p. 85714–85728, maio 2020.

CHANG, C. *et al.* **Automation with Ansible Student Workbook (Role)**. 2016. Material de curso/Workbook.

Docker Inc. **What is Docker? / Running containers - Docker Docs**. 2024. Disponível em: https://docs.docker.com/get-started/docker-overview/. Acesso em: 18 nov. 2025.

DONNO, M. D.; TANGE, K. P.; DRAGONI, N. Foundations and evolution of modern computing paradigms: Cloud, iot, edge, and fog. **IEEE Access**, v. 7, p. 150936–150948, 2019. Disponível em: https://doi.org/10.1109/ACCESS.2019.2947652.

EDWARDS, D. **What is DevOps?** 2010. `dev2ops.org`. Acesso em 9 jul. 2025. Disponível em: http://dev2ops.org/2010/02/what-is-devops/.

FONSECA, R.; SIMÕES, A. Alternativas ao xml: Yaml e json. In: **Actas da Conferência Nacional, XATA2007**. Lisboa: [*S. n.*], 2007. ISBN 978-972-99166-4-9.

FORSGREN, N.; HUMBLE, J.; KIM, G. **Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations**. [*S. l.*]: IT Revolution Press, 2018.

GAJBHIYE, A.; SHRIVASTVA, K. M. P. D. Cloud computing: Need, enabling technology, architecture, advantages and challenges. In: **2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence)**. IEEE, 2014. p. 1–7. Disponível em: https://doi.org/10.1109/CONFLUENCE.2014.6949261.

GEERLING, J. **Ansible for DevOps: Server and configuration management for humans**. Canadá: Leanpub, 2015.

GIL, A. C. **Como elaborar projetos de pesquisa**. 4. ed. São Paulo: Atlas, 2002.

HASHICORP. **HashiCorp 2023 Year in Review: Product Innovation**. 2023. Acessado em 28 de maio de 2025. Disponível em: https://www.hashicorp.com/blog/hashicorp-2023-year-in-review-product-innovation.

HashiCorp. **Infrastructure as Code: What Is It? Why Is It Important?** 2025. Publicado originalmente em 2018. Disponível em: [https://www.hashicorp.com/resources/what-is-infrastructure-as-code]. Acesso em: 27 out. 2025.

HAT, R. **Ansible Automates 2023 - Red Hat**. 2023. Acessado em 28 de maio de 2025. Disponível em: https://www.redhat.com/en/engage/ansible-automates-2023.

HEAP, M. **Ansible: From Beginner to Pro**. [*S. l.*]: Apress, 2016. ISBN 978-1-4842-1660-6.

HOCHSTEIN, L.; MOSER, R. **Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way**. [*S. l.*]: O'Reilly Media, Inc., 2017. ISBN 978-1-4919-7977-8.

HUMBLE, J.; FARLEY, D. **Entrega Contínua: Como Entregar Software de Forma Rápida e Confiável**. Rio de Janeiro: Alta Books, 2013.

HÜTTERMANN, M. **DevOps for Developers: Integrate development and operations, the agile way**. New York: Apress, 2012.

HüTTERMANN, M. **DevOps for Developers**. [*S. l.*]: Apress, 2012.

INSTRUCT. **Guia definitivo de Infraestrutura Ágil**. 2017. Acesso em: 01 dez. 2025. Disponível em: https://instruct.com.br/wp-content/uploads/2017/05/guia-completo-infra-agil.pdf.

K3D Project. **k3d: Little helper to run CNCF's k3s in Docker**. 2024. Disponível em: https://k3d.io/. Acesso em: 18 nov. 2025.

MANVI, S. S.; SHYAM, G. K. **Cloud Computing: Concepts and Technologies**. CRC Press, 2021. ISBN 9780367554613. Disponível em: https://www.taylorfrancis.com/books/mono/10.1201/9781003093671/cloud-computing-sunilkumar-manvi-gopal-shyam.

MARSTON, S.; LI, Z.; BANDYOPADHYAY, S.; ZHANG, J.; GHALSASI, A. Cloud computing — the business perspective. **Decision Support Systems**, v. 51, n. 1, p. 176–189, 2011. Disponível em: https://doi.org/10.1016/j.dss.2010.12.006.

MORRIS, K. **Infrastructure as Code: Managing infrastructure with VAGRANT, CHEF, DOCKER, and other tools**. [*S. l.*]: O'Reilly Media, 2016.

NELSON-SMITH, M. **Infrastructure as Code: Managing Servers in the Cloud**. [*S. l.*]: O'Reilly Media, 2013.

OPPENHEIMER, D.; GANAPATHI, A.; PATTERSON, D. A. Why do internet services fail, and what can be done about it? In: **Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (SITS '03)**. [*S. l.*: *s. n.*], 2003.

PILLA, D. **Automação Ansible, do CLI para uma solução mais Enterprise**. 2020. Acesso em: 01 dez. 2025. Disponível em: https://zallpy.com/ar10022020.html.

Poetry Project. **Poetry - Python dependency management and packaging made easy**. 2024. Disponível em: https://python-poetry.org/. Acesso em: 18 nov. 2025.

Red Hat. **What is infrastructure as code (IaC)?** 2020. Disponível em: [https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac]. Acesso em: 27 out. 2025.

RIOS, M. S. **Computação de borda: uma avaliação do processamento de dados para aplicações de IoT**. Dissertação (Dissertação de Mestrado) – Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, 2022.

RITI, E. **DevOps for Dummies**. [*S. l.*]: John Wiley & Sons, 2018.

SABHARWAL, N.; PANDEY, S.; PANDEY, P. Understanding terraform programming constructs. In: **Infrastructure-as-Code Automation Using Terraform, Packer, Vault, Nomad and Consul**. [*S. l.*]: Springer, 2021. p. 47–83.

SAITO, K.; LEE, J.; WU, C. Configuration management with ansible. In: **2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)**. [*S. l.*: *s. n.*], 2017. p. 3114–3119.

SATO, D. **DevOps: Na prática: Entrega de software confiável e automatizada**. São Paulo: Casa do Código, 2014.

SHAFER, A. C. **Agile Infrastructure: A Story in Three Acts**. San Jose: Velocity, 2009.

SHARMA, S. **DevOps For Dummies: IBM Limited Edition**. New Jersey: John Wiley & Sons, 2014.

SHI, W.; CAO, J.; ZHANG, Q.; LI, Y.; XU, L. Edge computing: Vision and challenges. **IEEE Internet of Things Journal**, IEEE, v. 3, n. 5, p. 637–646, out. 2016.

Vision Temenos. **DevOps + SAFe = DevOps?** 2018. Acesso em 9 jul. 2025. Disponível em: https://www.visiontemenos.com/blog/devops-safe-devops.

YI, S.; LI, C.; LI, Q. A survey of fog computing: Concepts, applications and issues. In: **Proceedings of the Workshop on Mobile Big Data (MoBiData)**. Atlanta, GA: ACM, 2015. p. 37–42.

YIN, R. K. **Estudo de Caso: Planejamento e Métodos**. 5. ed. Porto Alegre: Bookman, 2015.