INSTITUTO FEDERAL DE MINAS GERAIS
CAMPUS SABARÁ
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

IAGO NEVES CALDEIRA

DEVOPS E INFRAESTRUTURA COMO CÓDIGO: UMA ABORDAGEM PARA ALTA DISPONIBILIDADE E ESCALABILIDADE NA AWS

Sabará, Minas Gerais
2026

IAGO NEVES CALDEIRA

DEVOPS E INFRAESTRUTURA COMO CÓDIGO: UMA ABORDAGEM PARA ALTA DISPONIBILIDADE E ESCALABILIDADE NA AWS

Trabalho de Conclusão de Curso apresentado ao Instituto Federal de Minas Gerais, Campus Sabará, como requisito parcial para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Carlos Alberto Severiano Júnior.

Sabará, Minas Gerais

2026

Caldeira, Iago Neves

C146d DevOps e infraestrutura como código [manuscrito]: uma abordagem para alta disponibilidade e escalabilidade na AWS. / Iago Neves Caldeira. - 2026.

38 f. : il.

Orientação: Prof. Dr. Carlos Alberto Severiano Júnior

Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Instituto Federal de Minas Gerais, *Campus* Sabará.

1. Software - Desenvolvimento. – Monografia. 2. Computação em nuvem. – Monografia. 3. Computadores - Medidas de segurança. – Monografia. 4. Tecnologia da informação. – Monografia. 5. Automação. – Monografia. I. Severiano Júnior, Carlos Alberto. II. Instituto Federal de Minas Gerais, *Campus* Sabará. III. Bacharelado em Sistemas de Informação. IV. Título.

CDU 004.4

César dos Santos Moreira / CRB6-2229
Biblioteca do IFMG *Campus* Sabará

INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
Minas Gerais
Campus Sabará

ATA DE DEFESA DO TRABALHO DE CONCLUSÃO DE CURSO de **IAGO NEVES CALDEIRA**

No dia 27 do mês de maio do ano de 2026, às 10 horas e 40 minutos, os professores: Carlos Alberto Severiano Junior, Luiz Guilherme Hilel Drumond Silveira e Bruno Nonato Gomes compareceram para defesa pública do Trabalho de Conclusão de Curso intitulado **DevOps e Infraestrutura Como Código: Uma Abordagem para Alta Disponibilidade e Escalabilidade na AWS**, requisito obrigatório para a obtenção do título de **Bacharel**. Após a apresentação e as observações dos membros da banca avaliadora, ficou definido que o trabalho foi considerado:

(X) Aprovado ( ) Reprovado.

O resultado final foi comunicado publicamente ao candidato pelo Professor Orientador. Nada mais havendo a tratar, o Professor Orientador a reunião e lavrou a presente ATA, que será assinada por todos os membros participantes da banca avaliadora.

Observações: O ALUNO DEVE REALIZAR AS CORREÇÕES SOLICITADAS PELOS COMPONENTES DA BANCA EXAMINADORA.

CARLOS ALBERTO SEVERIANO JUNIOR
Professor Orientador

LUIZ GUILHERME HILEL DRUMOND SILVEIRA
Membro da Banca Examinadora

BRUNO NONATO GOMES
Membro da Banca Examinadora

IAGO NEVES CALDEIRA
Aluno

RESUMO

Este trabalho investiga como as práticas de DevOps e a automação de infraestrutura podem ser aplicadas para garantir escalabilidade, resiliência e eficiência operacional em ambientes de nuvem. A pesquisa, de natureza aplicada e abordagem qualitativa, combina revisão bibliográfica com um estudo de caso conduzido em ambiente controlado da Amazon Web Services (AWS). A arquitetura proposta articula Infraestrutura como Código (IaC) com Terraform e Terragrunt; computação elástica por meio de Auto Scaling Groups (ASG) sobre instâncias Amazon EC2; balanceamento e verificação de saúde via Application Load Balancer (ALB); persistência relacional gerenciada com Amazon RDS PostgreSQL em modo Multi-AZ; mensageria assíncrona com Amazon SQS; e observabilidade contínua com Amazon CloudWatch. A validação foi realizada por meio de testes de escalabilidade ascendente e descendente, com observação direta de métricas, logs e dashboards. Os resultados demonstraram provisionamento padronizado, ajuste automático de capacidade em resposta a métricas de tamanho de fila, redução de janelas de indisponibilidade e otimização de custos pela contração da frota em períodos ociosos. Discutem-se ainda os trade-offs entre dependência de serviços gerenciados de um único provedor e alternativas autogerenciadas, bem como práticas complementares de teste de carga, engenharia do caos e GitOps. Conclui-se que a combinação de IaC, automação reativa a eventos e monitoramento contínuo entrega ganhos consistentes em disponibilidade, reprodutibilidade e eficiência, ainda que com limitações de ambiente de simulação e cargas sintéticas.

Palavras-chave: DevOps; infraestrutura como código; escalabilidade; alta disponibilidade; AWS; Terraform; Auto Scaling.

SUMÁRIO

1 INTRODUÇÃO

A entrega contínua de software em larga escala impõe um conjunto de exigências que sistemas tradicionais, operados de forma manual e segmentada, têm dificuldade de atender. Velocidade de implantação, confiabilidade de execução, capacidade de absorver variações abruptas de carga e respostas rápidas a falhas tornaram-se requisitos comuns mesmo a aplicações de porte modesto, especialmente quando hospedadas em ambientes públicos de nuvem. Como apontam Kim et al. (2016), a separação rígida entre desenvolvimento e operações cria pontos de atrito que se materializam em ciclos longos de entrega, retrabalho e instabilidade, sintomas que a cultura DevOps se propõe a tratar.

Esse cenário se intensificou na última década. Incidentes públicos de indisponibilidade em plataformas amplamente utilizadas pelo grande público, frequentemente relatados pela imprensa especializada, ilustram o custo concreto da fragilidade operacional. Em junho de 2025, por exemplo, o aplicativo de uma das maiores plataformas de entrega de alimentos do Brasil ficou fora do ar para usuários de dispositivos Android, gerando mais de três mil reclamações em pouco mais de uma hora e ocupando, no mesmo dia, lugar de destaque entre os termos mais buscados pelos brasileiros no Google (EXAME, 2025; TECHTUDO, 2025). Episódios dessa natureza traduzem, em termos práticos, o que a literatura técnica vem descrevendo há anos: à medida que sistemas crescem em complexidade e em base de usuários, qualquer ponto único de falha pode comprometer a operação inteira, com impacto direto sobre reputação e receita.

É nesse contexto que o DevOps emerge como abordagem cultural e técnica para integrar desenvolvimento e operações, sustentada por automação, integração e entrega contínuas, monitoramento proativo e infraestrutura tratada como código (HUMBLE; FARLEY, 2010; KIM et al., 2016). Diante desse pano de fundo, este trabalho parte da seguinte questão de pesquisa: como práticas DevOps e a automação de infraestrutura podem ser aplicadas para garantir escalabilidade, resiliência e eficiência operacional em ambientes de nuvem?

Para investigar essa questão, propõe-se um estudo de caso em ambiente controlado, no qual uma aplicação web simples é implantada sobre uma arquitetura escalável na AWS, provisionada inteiramente por meio de Infraestrutura como Código (IaC). A escolha por uma aplicação propositalmente enxuta, uma API HTTP que interage com banco de dados

gerenciado e com uma fila de mensagens, busca isolar o foco da análise nos comportamentos da infraestrutura, e não na lógica de negócio.

Como objetivo geral, este trabalho propõe-se a analisar a aplicação de práticas DevOps em uma infraestrutura escalável na AWS, com ênfase nos efeitos de automação, elasticidade e observabilidade sobre disponibilidade e eficiência operacional. A partir desse objetivo geral, derivam-se os seguintes objetivos específicos:

- investigar os conceitos e práticas fundamentais da cultura DevOps aplicáveis à gestão de infraestrutura em nuvem, com base em literatura acadêmica recente;
- projetar e implementar uma arquitetura escalável na AWS utilizando Terraform e Terragrunt, articulando os serviços EC2, ALB, ASG, RDS e SQS;
- avaliar o comportamento da infraestrutura sob variações de carga, com observação direta de métricas, logs e alarmes do Amazon CloudWatch;
- analisar os benefícios e limitações da abordagem em termos de disponibilidade, eficiência operacional, custo e reprodutibilidade, contrastando com alternativas autogerenciadas.

O texto está organizado da seguinte forma. O Capítulo 2 apresenta o referencial teórico, revisando os pilares da cultura DevOps, princípios de Infraestrutura como Código, estratégias de escalabilidade, GitOps, engenharia do caos e testes de carga, com apoio em literatura acadêmica e técnica. O Capítulo 3 descreve a metodologia adotada e justifica o desenho da pesquisa. O Capítulo 4 detalha o estudo de caso, expondo a arquitetura proposta, as ferramentas envolvidas, a aplicação utilizada, a estratégia de testes e a análise comparativa de custos. O Capítulo 5 sintetiza as conclusões, e o Capítulo 6 indica direções para trabalhos futuros.

2 REFERENCIAL TEÓRICO

Este capítulo organiza os conceitos que sustentam a proposta. Parte-se de uma síntese histórica do DevOps, segue-se com os quatro pilares culturais e técnicos do movimento, composto por automação, integração e entrega contínuas, infraestrutura como código e gestão de configuração, e avança-se para tópicos correlatos relevantes para o estudo de caso: estratégias de escalabilidade, GitOps, observabilidade, engenharia do caos e testes de carga. A revisão privilegia literatura acadêmica de referência, complementada por documentação técnica oficial quando essa documentação é a fonte primária do conceito.

2.1 Origens e definição de DevOps

O termo DevOps foi cunhado em 2009, no encontro DevOpsDays organizado em Ghent (Bélgica) por Patrick Debois, e popularizou-se a partir da palestra “10+ Deploys per Day: Dev and Ops Cooperation at Flickr”, apresentada por John Allspaw e Paul Hammond na O’Reilly Velocity Conference do mesmo ano (KIM et al., 2016). Sua proposta central é a integração das equipes de desenvolvimento e de operações em torno de objetivos comuns de qualidade, velocidade e estabilidade. Humble e Farley (2010) descrevem, na obra seminal sobre entrega contínua, o pano de fundo que motivou esse movimento: ciclos manuais e estanques de release transformavam cada implantação em evento arriscado, com janelas de indisponibilidade e dependência intensa de coordenação humana.

Em definição amplamente referenciada, Kim et al. (2016, p. 4, tradução nossa) caracterizam DevOps como um conjunto de princípios e práticas que “permite às organizações de tecnologia entregar valor com qualidade, segurança e rapidez, alinhando desenvolvimento, operações, segurança da informação e demais áreas envolvidas no ciclo de vida do software”. A cultura DevOps, portanto, não se reduz a um conjunto de ferramentas; ela articula práticas técnicas (automação, IaC, CI/CD, monitoramento), princípios organizacionais (responsabilidade compartilhada, blameless postmortems, melhoria contínua) e métricas de desempenho de engenharia consolidadas pela pesquisa DORA (FORSGREN; HUMBLE; KIM, 2018).

2.2 Automação de processos

Automação é frequentemente apontada como o pilar mais visível do DevOps, mas é importante distinguir automação como meio e automação como fim em si. Humble e Farley

(2010) argumentam que toda atividade repetitiva, sujeita a erro humano ou que dependa de coordenação síncrona entre times, como compilação, execução de testes, provisionamento de ambientes, deploy, migração de banco de dados e configuração de rede, deve ser tratada como candidata à automação. O ganho não está apenas na velocidade: a automação dá consistência, transforma processos em artefatos versionáveis e converte conhecimento tácito em conhecimento explícito.

No domínio da infraestrutura, o efeito é particularmente notável. Quando um ambiente é provisionado a partir de um conjunto de scripts ou definições declarativas, a operação deixa de depender de notas em wiki, de procedimentos passo-a-passo executados manualmente ou de configurações implícitas em uma máquina "especial". O custo de recriar um ambiente cai drasticamente, e práticas como ambientes efêmeros, ambientes por pull request e disaster recovery automatizado tornam-se viáveis (MORRIS, 2020).

2.3 Integração e entrega contínuas

Integração Contínua (CI) é a prática de integrar frequentemente o trabalho de cada desenvolvedor em uma linha principal compartilhada, validando cada integração por meio de build automatizado e testes (HUMBLE; FARLEY, 2010). A justificativa é estatística: quanto maior o intervalo entre integrações, maior a probabilidade de conflitos e maior o custo de resolvê-los.

Entrega Contínua (CD) estende a CI ao tornar o software entregável a qualquer momento, isto é, qualquer versão que passe pelo pipeline está, por construção, em condição de ser publicada. Humble e Farley (2010) propõem o conceito de "deployment pipeline": uma representação automatizada do caminho que uma mudança percorre, do commit ao ambiente de produção, atravessando estágios sucessivos de validação. Parnin et al. (2017) catalogam dez adágios consolidados na prática de entrega contínua, entre os quais figuras como "se dói, faça com mais frequência" e "automatize tudo o que for repetitivo", que se tornaram aforismos do movimento.

Ferramentas que materializam CI/CD incluem Jenkins, GitLab CI/CD, GitHub Actions, CircleCI e ArgoCD (esta última no contexto de GitOps, discutido adiante). Para o presente estudo, importa menos a ferramenta específica e mais o princípio: a esteira de validação automatizada como artefato de primeira classe da engenharia.

2.4 Infraestrutura como Código

Infraestrutura como Código (Infrastructure as Code, IaC) é a prática de descrever e gerenciar a infraestrutura, como máquinas virtuais, redes, balanceadores, bancos gerenciados e regras de firewall, por meio de arquivos legíveis por humanos e processáveis por máquina, tratados como código-fonte: versionados em Git, revisados em pull requests, testados e implantados por pipelines (MORRIS, 2020; QUATTROCCHI; TAMBURRI, 2023).

Rahman, Mahdavi-Hezaveh e Williams (2019) conduziram um mapeamento sistemático sobre pesquisa em IaC, identificando IaC como pilar fundamental da entrega contínua e mostrando seu crescimento como tópico de pesquisa formal. Em estudo subsequente, Rahman et al. (2020) propõem uma taxonomia de defeitos em scripts de IaC, incluindo defeitos sintáticos, semânticos e de segurança, apontando que, à medida que IaC se generaliza, os mesmos rigores aplicados a software de aplicação precisam ser estendidos a essas definições.

Sob a ótica da arquitetura, IaC viabiliza três propriedades essenciais. Primeiro, reprodutibilidade: o mesmo conjunto de definições gera ambientes equivalentes em desenvolvimento, homologação e produção, mitigando a clássica diferença de comportamento entre ambientes. Segundo, versionamento: cada mudança na infraestrutura passa pelo controle de versão, com histórico, autoria e possibilidade de reversão. Terceiro, auditabilidade: o estado desejado da infraestrutura é explícito no repositório, o que facilita revisão, conformidade e investigação de incidentes.

Entre as ferramentas mais consolidadas figura o Terraform, desenvolvido pela HashiCorp, que adota uma abordagem declarativa: o engenheiro descreve o estado desejado e a ferramenta calcula o conjunto de operações necessárias para alcançá-lo, comparando com o estado atual mantido em um arquivo de estado (state file). O Terragrunt, por sua vez, atua como camada de orquestração sobre o Terraform, oferecendo mecanismos para manter configurações compartilhadas DRY (Don’t Repeat Yourself) e organizar múltiplos ambientes (ARTAC et al., 2017).

2.5 Gestão de configuração

A gestão de configuração diz respeito ao controle das definições internas das instâncias e dos serviços ao longo do ciclo de vida (MORRIS, 2020). Enquanto IaC, no sentido estrito, cuida da topologia, isto é, quais recursos existem e como se conectam, a gestão de configuração trata do interior de cada um: pacotes instalados, variáveis de ambiente,

certificados, regras de log. Ferramentas como Ansible, Puppet, Chef e SaltStack consolidaram-se nesse domínio.

Para o estudo de caso desta pesquisa, optou-se por uma forma mais leve: um script user-data executado pelo EC2 no momento do boot, responsável por instalar dependências, baixar artefatos e iniciar o serviço da aplicação. Essa decisão é discutida no Capítulo 4. Independentemente da ferramenta, três disciplinas se mantêm essenciais: versionar o conteúdo de configuração, aplicá-lo de forma idempotente e auditar continuamente o estado resultante.

2.6 Estratégias de escalabilidade

Escalabilidade refere-se à capacidade de um sistema de manter desempenho aceitável diante de variações de carga. Tradicionalmente, distinguem-se duas estratégias fundamentais:

- **Escalabilidade vertical (scale up): aumentar a capacidade de uma máquina já em operação, com mais CPU, mais memória e disco mais rápido. É simples conceitualmente, mas esbarra em limites físicos do hardware, custo crescente de forma não-linear e, em geral, exige reinício do serviço para aplicar mudanças significativas.**
- **Escalabilidade horizontal (scale out): adicionar mais máquinas pequenas trabalhando em paralelo. Requer que a aplicação tolere distribuição, preferencialmente sem estado nas instâncias, mas oferece elasticidade praticamente ilimitada, tolerância a falhas e custo proporcional ao uso.**

Em ambientes de nuvem, a escalabilidade horizontal é predominante por duas razões. Primeiro, o modelo econômico favorece consumo por demanda: contrata-se capacidade enquanto necessária, devolve-se quando ociosa. Segundo, a infraestrutura subjacente foi projetada para distribuição: zonas de disponibilidade, balanceadores, filas e serviços gerenciados pressupõem múltiplos consumidores e produtores.

A escalabilidade horizontal, por sua vez, pode ser disparada por diferentes métricas. As mais comuns em provedores públicos são:

- **Métricas de CPU:** escalar quando a utilização média de CPU em uma frota ultrapassa um patamar (por exemplo, 70%). É um proxy razoável para “a frota está atendendo no limite”, mas pode ser enganoso em workloads I/O-bound.
- **Métricas de memória:** relevantes para aplicações que mantêm cache em memória ou processam dados volumosos. Em AWS, exige instalação do CloudWatch Agent, pois a métrica de memória não é nativa do hipervisor.

- **Métricas de fila:** escalar com base no tamanho de uma fila de mensagens (por exemplo, ApproximateNumberOfMessagesVisible no SQS). Modelo particularmente robusto, pois o tamanho da fila reflete diretamente a defasagem entre produção e consumo de trabalho.
- **Métricas de aplicação (custom metrics):** latência de resposta, taxa de erros, taxa de requisições por segundo. Permitem políticas mais aderentes à semântica do sistema, mas exigem instrumentação adequada.
- **Escalonamento agendado (scheduled scaling):** útil quando padrões de uso são previsíveis (por exemplo, horário comercial), antecipando demanda em vez de reagir a ela.

Há ainda a distinção entre políticas reativas (que respondem a métricas atuais), preditivas (que utilizam séries históricas para prever demanda futura) e baseadas em alvo (target tracking, que mantém uma métrica próxima de um valor desejado). Cada estratégia tem trade-offs entre simplicidade, custo e qualidade de resposta.

2.7 GitOps

GitOps é uma evolução natural dos princípios de IaC, formalizada pela Weaveworks em 2017 (RICHARDSON, 2017) e posteriormente refinada como um conjunto de princípios pelo grupo de trabalho OpenGitOps, sob a Cloud Native Computing Foundation. Os quatro princípios são (BECK et al., 2021):

- **Declarativo:** o estado desejado do sistema é descrito de forma declarativa, sem prescrever como alcançá-lo.
- **Versionado e imutável:** esse estado é armazenado em um sistema de controle de versão (tipicamente Git), com histórico completo e imutável.
- **Aplicado automaticamente:** agentes especializados aplicam o estado desejado à infraestrutura, sem ação humana direta no ambiente de execução.
- **Continuamente reconciliado:** o agente compara periodicamente o estado real ao desejado e corrige divergências (drift).

Em termos práticos, GitOps converte o repositório Git na única fonte autorizada da verdade sobre o que deve estar em produção. Mudanças não são aplicadas via comando manual em servidores, mas por meio de pull requests revisados, mesclados e propagados automaticamente. O benefício duplo é evidente: ganha-se em auditoria (todo o histórico de

quem mudou o quê e quando está em commits) e em resiliência (em caso de incidente, pode-se reverter ao estado anterior conhecido por meio de um revert no Git).

2.8 Observabilidade e monitoramento

Observabilidade vai além de monitoramento clássico: enquanto este último responde a perguntas previamente conhecidas (“o serviço está no ar?”), observabilidade permite formular perguntas novas sobre o comportamento do sistema a partir dos sinais que ele produz (BEYER et al., 2016). Os três pilares clássicos da observabilidade são métricas, logs e traces, frequentemente complementados por eventos e dashboards.

Em ambientes nativos da AWS, o CloudWatch é o ponto central de coleta de métricas e disparo de alarmes. Cada serviço gerenciado expõe um conjunto de métricas padrão (utilização de CPU para EC2, número de mensagens visíveis para SQS, conexões ativas para RDS), e métricas customizadas podem ser publicadas pela aplicação. Os alarmes do CloudWatch monitoram essas métricas continuamente e disparam ações, como notificações via SNS, execução de funções Lambda e ajuste de capacidade no ASG, quando condições configuradas são satisfeitas.

2.9 Engenharia do caos

Engenharia do caos (Chaos Engineering) é a disciplina de submeter sistemas distribuídos a falhas controladas em ambiente de produção (ou próximo dela) com o objetivo de validar empiricamente a resiliência arquitetural (BASIRI et al., 2016). A prática nasceu na Netflix, com o Chaos Monkey, ferramenta que terminava instâncias EC2 aleatoriamente em produção para forçar que os sistemas fossem desenhados desde o início para tolerar a perda de nós.

Basiri et al. (2016) formalizam quatro princípios da prática: definir o estado estável do sistema, hipotetizar que esse estado se mantém na presença de falhas, introduzir variáveis que reflitam eventos do mundo real (falha de instância, latência de rede, perda de conexão com dependência externa) e tentar refutar a hipótese, examinando se há desvio comparado ao grupo de controle. A engenharia do caos, portanto, não é “quebrar coisas por diversão”: é uma experimentação científica disciplinada sobre sistemas em produção.

2.10 Testes de carga

Testes de carga (load testing) avaliam o comportamento de um sistema sob volumes crescentes de requisições, com o objetivo de mapear sua capacidade, identificar gargalos e validar políticas de escalabilidade. Ferramentas amplamente adotadas incluem JMeter, k6, Locust, Gatling e Artillery.

Distinguem-se subtipos: teste de carga propriamente dito (sustenta uma carga representativa por período prolongado), teste de estresse (extrapola até identificar o ponto de quebra), teste de pico (simula crescimento abrupto, como uma campanha de marketing), e teste de soak (carga moderada por horas ou dias, para revelar vazamentos de memória ou degradação cumulativa). Esses testes são essenciais para validar empiricamente as políticas de Auto Scaling: configurações teóricas de cooldown, thresholds e step adjustments só revelam seu comportamento real sob carga sintética que se aproxime da real.

2.11 Trabalhos relacionados

A literatura brasileira recente apresenta diversos esforços de aplicação de DevOps e IaC em contextos próximos ao deste trabalho. Universidade Federal de Uberlândia (2025) descreve a implementação de infraestrutura como código e integração CI/CD em ambiente de nuvem, com foco em reduzir trabalho manual e padronizar deploy. Gomes (2020) examina o processo de adoção da cultura DevOps em uma organização, articulando os pilares de cultura, automação, métrica e compartilhamento. Chaves (2020) propõe um modelo prático para implementar CI/CD, detalhando o ciclo automatizado de varredura de qualidade, testes e relatórios.

Univates (2022) descreve provisionamento automatizado de máquinas virtuais e contêineres com orquestração Kubernetes, e Instituto Federal do Sertão Pernambucano (2022) compara plataformas de nuvem para implementação de DevOps. Esses trabalhos, conjuntamente, sustentam a relevância do tema e ajudam a posicionar o presente estudo: enquanto parte da literatura local tem foco em Kubernetes ou em comparativos de plataformas, este trabalho concentra-se em uma stack baseada em serviços gerenciados da AWS, com ênfase em escalabilidade horizontal orientada a eventos.

No plano internacional, Artac et al. (2017) introduzem IaC no contexto de DevOps com foco em automação de provisionamento; Kumara et al. (2021) compilam, em revisão sistemática de literatura cinza, boas e más práticas de IaC; e Pahl, Sezen e Hofer (2026), em revisão sistemática, examinam o uso de inteligência artificial em todo o ciclo de IaC. Para o

escopo deste trabalho, esses estudos servem de pano de fundo para a discussão sobre maturidade, defeitos comuns e direções futuras de IaC.

3 METODOLOGIA

Esta pesquisa adota natureza aplicada, com abordagem qualitativa e finalidade exploratório-descritiva. O método é o estudo de caso conduzido em ambiente controlado, no qual se implementa, opera e avalia uma arquitetura escalável na AWS sob condições experimentais reproduzíveis. A escolha do estudo de caso justifica-se pelo objetivo central da pesquisa, examinar fenômenos contemporâneos em seu contexto real (YIN, 2018), e pela natureza eminentemente prática da questão de pesquisa, que demanda observação direta do comportamento da infraestrutura.

3.1 Etapas da pesquisa

O trabalho foi conduzido em quatro etapas principais. A primeira consistiu na revisão bibliográfica, com seleção de fontes em três níveis: livros de referência consolidados (HUMBLE; FARLEY, 2010; MORRIS, 2020; KIM et al., 2016; BEYER et al., 2016), artigos acadêmicos publicados em periódicos e conferências da área (BASIRI et al., 2016; RAHMAN; MAHDAVI-HEZAVEH; WILLIAMS, 2019; ARTAC et al., 2017; QUATTROCCHI; TAMBURRI, 2023) e documentação técnica oficial dos serviços empregados, quando essa documentação representa a fonte primária do conceito (AWS, 2024). Os critérios de inclusão privilegiaram trabalhos publicados nos últimos dez anos para temas em evolução rápida (GitOps, observabilidade) e obras seminais para temas consolidados (CI/CD, IaC).

A segunda etapa concentrou-se no desenho da arquitetura. Partindo dos requisitos de elasticidade, alta disponibilidade, desacoplamento e observabilidade, mapeou-se cada requisito a um conjunto de serviços gerenciados da AWS, conforme detalhado no Capítulo 4. Esse mapeamento foi documentado em diagramas e em uma matriz requisito-serviço, garantindo rastreabilidade entre as decisões de projeto e os objetivos do trabalho.

A terceira etapa correspondeu à implementação. A infraestrutura foi descrita em Terraform organizado em módulos reutilizáveis e orquestrada por Terragrunt, com separação clara entre módulos (definições genéricas) e environments (instâncias paramétricas). O estado do Terraform foi armazenado remotamente em bucket S3 versionado e criptografado, com lock de concorrência via DynamoDB, padrão consolidado pela documentação oficial. A aplicação foi desenvolvida em Node.js com Express.js e provisionada nas instâncias EC2 por meio de script user-data idempotente.

A quarta etapa consistiu na validação. Foram conduzidos dois testes principais: scale-up, induzido pela publicação de mensagens consecutivas no SQS, e scale-down, induzido pelo consumo dessas mensagens e pela manutenção de fila vazia por período suficiente. Em ambos os casos, observaram-se: (i) o disparo dos alarmes do CloudWatch, (ii) a atuação do ASG na adição ou remoção de instâncias, (iii) o tempo decorrido entre o disparo do alarme e a entrada efetiva de capacidade no balanceador, e (iv) o impacto sobre métricas auxiliares como utilização de CPU, tráfego de rede e saldo de créditos de CPU. Para fins de análise comparativa de custos, foram modelados dois cenários: máquina ligada 24 horas por dia versus frota elástica via ASG, com base nos preços públicos de instâncias EC2 em região us-east-1, conforme apresentado no Capítulo 4.

3.2 Materiais e ferramentas

O conjunto de ferramentas mobilizado foi o seguinte:

- **Provedor de nuvem:** Amazon Web Services (AWS), região us-east-1, com uso dos serviços EC2, ALB, ASG, RDS PostgreSQL, SQS, CloudWatch, S3 e DynamoDB.
- **Infraestrutura como Código:** Terraform e Terragrunt, com providers oficiais da AWS.
- **Aplicação:** Node.js (LTS), Express.js, SDK oficial da AWS para JavaScript.
- **Configuração de instância:** script user-data em Bash, executado pelo cloud-init no boot.
- **Observabilidade:** Amazon CloudWatch para métricas e alarmes; logs estruturados das aplicações.
- **Controle de versão:** Git, com fluxo baseado em pull requests.

3.3 Critérios de avaliação

Os resultados foram analisados sob quatro dimensões correlacionadas aos objetivos específicos: disponibilidade (presença de instâncias saudáveis atendendo tráfego durante todo o teste), elasticidade (capacidade do ASG de adicionar e remover instâncias em resposta a métricas, com tempos compatíveis com a janela de demanda), reprodutibilidade (capacidade de destruir e recriar todo o ambiente a partir do código, obtendo configuração equivalente) e eficiência de custo (estimativa de custo do cenário elástico comparado ao cenário de máquina

sempre ligada). A análise é qualitativa, sustentada por evidências quantitativas (métricas e estimativas) sempre que aplicável.

3.4 Limitações metodológicas

Reconhecem-se três limitações principais. Primeiro, o ambiente é controlado: a carga é sintética e gerada por scripts, não reflete integralmente padrões reais de tráfego de produção. Segundo, a aplicação é deliberadamente simples, uma API que troca dados com um banco e uma fila, o que facilita o isolamento da análise, mas restringe a generalização para sistemas com múltiplos microsserviços, dependências externas e estado distribuído. Terceiro, a arquitetura é fortemente acoplada a serviços gerenciados de um único provedor (AWS), o que é discutido criticamente no Capítulo 4 e endereçado como direção de trabalho futuro no Capítulo 6.

4 ESTUDO DE CASO

Este capítulo descreve a implementação e a avaliação da arquitetura proposta. Inicia-se pela visão geral da solução e pela aplicação utilizada; segue-se com o detalhamento de cada serviço e o papel que desempenha; apresentam-se a estratégia de Infraestrutura como Código, as políticas de escalabilidade e a malha de observabilidade; e encerra-se com os testes de escalabilidade, a análise de custos e a discussão de trade-offs entre AWS e alternativas.

4.1 Visão geral da solução

A arquitetura foi projetada para satisfazer quatro requisitos não-funcionais centrais: elasticidade (capacidade de ajustar recursos à demanda), alta disponibilidade (tolerância a falhas de instância individual), desacoplamento (resiliência a picos por meio de processamento assíncrono) e observabilidade (visibilidade contínua sobre o estado do sistema). A Figura 1 apresenta a topologia geral e o fluxo de uma requisição.

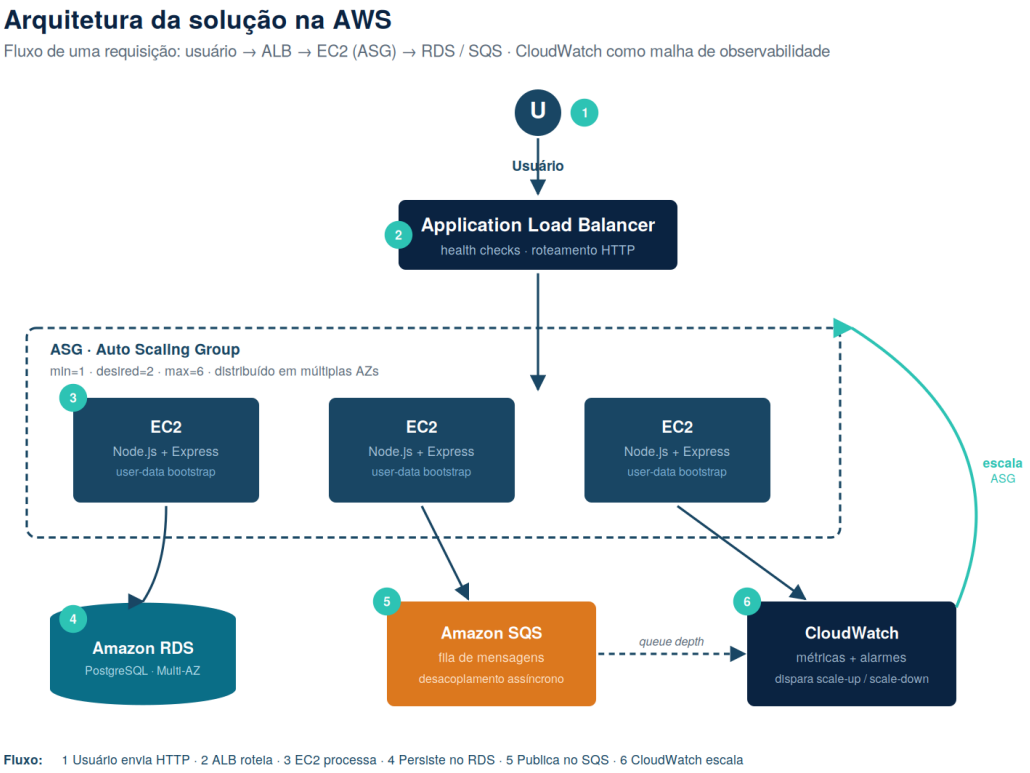

Figura 1 - Arquitetura da solução na AWS e fluxo de requisição
Fonte: Elaborado pelo autor.

O fluxo de uma requisição percorre seis etapas. O usuário envia uma requisição HTTP que chega ao Application Load Balancer (ALB), o qual a encaminha a uma instância EC2 saudável dentro do Auto Scaling Group (ASG). A instância, executando a aplicação Node.js, processa a requisição e, conforme a rota acionada, lê ou escreve no Amazon RDS, publica ou consome mensagens no Amazon SQS. O Amazon CloudWatch observa continuamente as métricas, em especial o tamanho da fila do SQS, e, ao cruzar limiares configurados, dispara alarmes que instruem o ASG a adicionar ou remover instâncias, fechando o ciclo de retroalimentação que confere elasticidade ao sistema.

A Tabela 1 sintetiza o mapeamento entre requisitos e serviços, evidenciando a rastreabilidade entre as decisões de projeto e os objetivos do trabalho.

Tabela 1 - Mapeamento entre requisitos não-funcionais e serviços da AWS

Requisito	Serviço(s) AWS	Como é atendido
Elasticidade	ASG + CloudWatch	ajuste automático da frota por métricas
Alta disponibilidade	ALB + múltiplas AZs	tráfego só a instâncias saudáveis
Desacoplamento	SQS	processamento assíncrono com buffer
Persistência confiável	RDS PostgreSQL	banco gerenciado, backups, Multi-AZ
Observabilidade	CloudWatch	métricas, dashboards e alarmes
Reprodutibilidade	Terraform + Terragrunt	infraestrutura versionada e modular

Fonte: Elaborado pelo autor.

4.2 A aplicação

A aplicação foi desenvolvida em Node.js com o framework Express.js e desenhada deliberadamente para ser simples, leve e sem estado (stateless). A ausência de estado nas instâncias é condição necessária para a escalabilidade horizontal: como qualquer requisição pode ser atendida por qualquer instância, o balanceador pode distribuir tráfego livremente e o ASG pode terminar instâncias sem perda de sessão. Todo estado relevante reside fora das instâncias: no RDS (dados persistentes) e no SQS (trabalho pendente).

A aplicação expõe quatro rotas, cada uma exercitando um aspecto da arquitetura:

- **Rota / (raiz):** serve uma página HTML mínima com links para as demais rotas, funcionando como painel de controle do experimento e permitindo acionar manualmente os fluxos.
- **Rota /send:** publica uma mensagem na fila do Amazon SQS. É o gatilho do experimento de scale-up: o acúmulo de mensagens eleva a métrica de profundidade de fila e induz o ASG a provisionar novas instâncias.

- **Rota /consume:** consome e remove mensagens da fila. É o gatilho do experimento de scale-down: ao esvaziar a fila e mantê-la vazia por tempo suficiente, induz a contração da frota.
- **Rota /dbtest:** estabelece conexão com o Amazon RDS e executa uma operação simples de leitura/escrita, validando a integridade e a operacionalidade do banco a partir de qualquer instância.

Para garantir execução contínua e recuperação automática, a aplicação é registrada como serviço gerenciado pelo systemd, configurado para reiniciá-la automaticamente em caso de falha. Combinado ao health check do ALB, esse arranjo cobre duas camadas de resiliência: o systemd recupera o processo dentro da instância, e o ALB remove do rodízio qualquer instância que não responda corretamente às verificações de integridade na rota de saúde.

4.2.1 O papel do banco de dados na arquitetura

Uma observação pertinente é que, em um experimento centrado em elasticidade orientada a fila, o banco de dados poderia parecer um componente secundário. Não é o caso, e vale explicitar três funções concretas que o Amazon RDS desempenha aqui.

Primeiro, o RDS representa o estado persistente compartilhado que torna a frota verdadeiramente intercambiável. Como nenhuma instância guarda dados localmente, é o banco que centraliza a informação que precisa sobreviver ao ciclo de vida efêmero das máquinas. Quando o ASG termina uma instância em um scale-down, nenhum dado é perdido, justamente porque o estado vive no RDS, e não na máquina. Essa separação entre computação efêmera e estado durável é o que viabiliza, na prática, o padrão de instâncias descartáveis (cattle, não pets).

Segundo, o RDS é o componente que exercita a dimensão de alta disponibilidade de dados, complementar à alta disponibilidade de computação fornecida pelo ALB e pelo ASG. Configurado em modo Multi-AZ, o RDS mantém uma réplica síncrona em outra zona de disponibilidade e promove failover automático em caso de falha da instância primária. Assim, a arquitetura tolera não apenas a perda de uma instância de aplicação, mas também a falha da própria infraestrutura de banco.

Terceiro, a rota /dbtest serve como sonda de conectividade que valida, a cada instância nova provisionada pelo ASG, que o acesso ao banco, incluindo regras de security group, credenciais e rede, foi corretamente configurado pelo script user-data. Em outras palavras, o banco também participa do experimento como ponto de verificação da correção do

provisionamento automatizado. Por essas razões, o RDS não é um apêndice da arquitetura, mas o elemento que confere sentido à elasticidade: sem estado externo confiável, instâncias descartáveis seriam inviáveis.

4.3 Detalhamento dos componentes

4.3.1 Amazon EC2 e o script user-data

As instâncias Amazon EC2 são a camada de execução da aplicação. Foram utilizadas instâncias do tipo t3.micro (família de uso geral, com modelo de créditos de CPU adequado a workloads de baixa utilização média com picos ocasionais). Cada instância é provisionada a partir de um launch template que referencia uma AMI base e injeta um script user-data, executado uma única vez no primeiro boot pelo cloud-init.

O script user-data cumpre o papel de gestão de configuração leve: instala as dependências (runtime Node.js e bibliotecas), baixa o artefato da aplicação, escreve o arquivo de unit do systemd e inicia o serviço. Por ser declarativo e versionado junto ao restante da IaC, ele garante que toda instância nasça idêntica, propriedade essencial para que o ASG possa adicionar capacidade sem intervenção manual.

4.3.2 Application Load Balancer

O Application Load Balancer (ALB) atua como ponto único de entrada do tráfego HTTP. Além de distribuir requisições entre as instâncias do target group associado ao ASG, executa verificações periódicas de integridade (health checks) em uma rota dedicada (por exemplo, /health). Uma instância é considerada saudável quando responde dentro do tempo limite e com o código de status esperado (tipicamente 200 OK). Caso uma instância falhe em um número consecutivo de verificações, é marcada como não-saudável e deixa de receber tráfego, sendo posteriormente substituída pelo ASG. Esse mecanismo isola falhas: um problema em uma instância não se propaga ao usuário final, que continua sendo atendido pelas instâncias remanescentes.

4.3.3 Auto Scaling Group

O Auto Scaling Group (ASG) é o coração da elasticidade. Ele mantém a frota de instâncias EC2 dentro de limites configurados, neste trabalho, mínimo de 1, capacidade desejada de 2 e máximo de 6, distribuindo-as por múltiplas zonas de disponibilidade. A Figura 2 ilustra seu funcionamento.

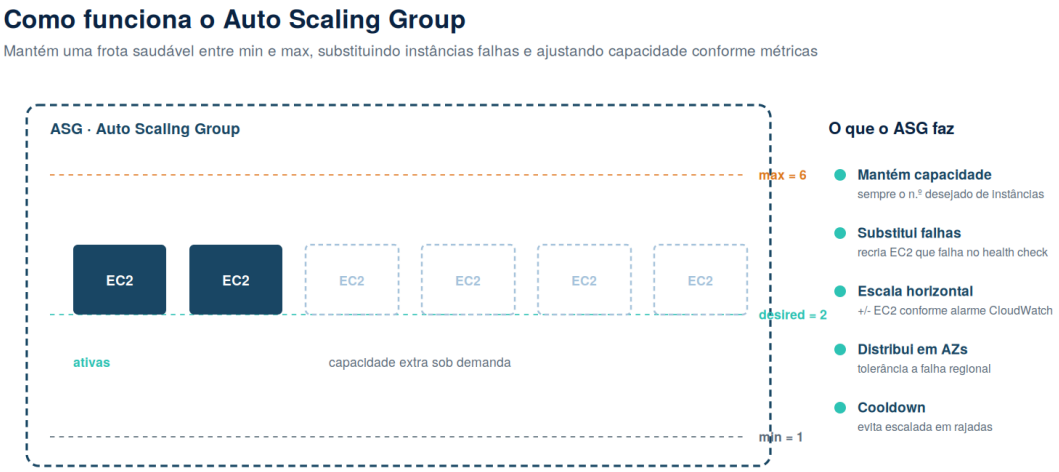

Figura 2 - Funcionamento do Auto Scaling Group
Fonte: Elaborado pelo autor.

O ASG desempenha quatro funções simultâneas. Mantém a capacidade desejada, repondo instâncias que sejam terminadas. Substitui instâncias que falhem nos health checks, terminando-as e provisionando substitutas. Escala horizontalmente, adicionando ou removendo instâncias, em resposta a alarmes do CloudWatch. E distribui as instâncias entre zonas de disponibilidade, conferindo tolerância a falhas de uma zona inteira.

4.3.4 Amazon RDS

Para persistência, adotou-se o Amazon RDS na engine PostgreSQL. Como serviço gerenciado, o RDS abstrai tarefas operacionais como aplicação de patches, backups automáticos, snapshots e replicação. Quando configurado em modo Multi-AZ, mantém uma réplica síncrona em zona distinta, com failover automático. O acesso ao banco é restrito por security groups, de modo que apenas as instâncias do ASG podem se conectar, reduzindo a superfície de ataque. As funções do RDS na arquitetura foram detalhadas na seção 4.2.1.

4.3.5 Amazon SQS

O Amazon SQS é o serviço de mensageria que viabiliza o desacoplamento entre produção e consumo de trabalho. Utilizou-se a fila do tipo Standard, que oferece alta vazão, ordenação best-effort e entrega ao menos uma vez (at-least-once). O SQS introduz duas propriedades arquiteturais valiosas. Primeiro, atua como buffer: picos de produção acumulam-se na fila sem derrubar os consumidores, que processam no próprio ritmo. Segundo, sua métrica de profundidade de fila é um sinal de escalabilidade naturalmente

robusto, pois reflete diretamente a defasagem entre o que está sendo produzido e o que está sendo consumido. A Figura 3 contrasta a abordagem síncrona acoplada com a abordagem assíncrona orientada a eventos.

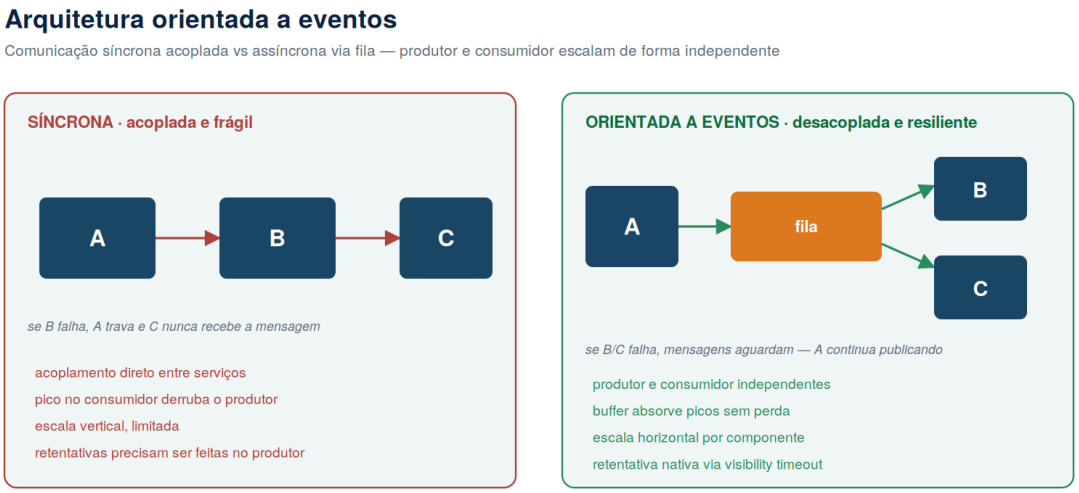

Figura 3 - Comunicação síncrona acoplada versus arquitetura orientada a eventos
Fonte: Elaborado pelo autor.

4.3.6 Amazon CloudWatch

O Amazon CloudWatch é a malha de observabilidade da arquitetura. Coleta métricas dos demais serviços e dispara alarmes que acionam as políticas de escalabilidade. As principais métricas monitoradas neste trabalho foram:

- **ApproximateNumberOfMessagesVisible (SQS):** número aproximado de mensagens disponíveis na fila aguardando processamento. É a métrica primária que dispara o escalonamento, pois traduz diretamente a carga de trabalho pendente.
- **CPUUtilization (EC2/ASG):** percentual médio de utilização de CPU da frota, usado como métrica auxiliar para validar o dimensionamento e identificar saturação de processamento.
- **CPUCreditBalance e CPUCreditUsage (instâncias T):** saldo e consumo de créditos de CPU das instâncias da família burstable, importantes para verificar se a frota não está esgotando seu orçamento de burst sob carga sustentada.
- **NetworkIn / NetworkOut e pacotes de rede:** tráfego de entrada e saída, úteis para correlacionar volume de requisições com consumo de recursos.

- **HealthyHostCount (ALB):** número de instâncias consideradas saudáveis pelo balanceador, indicador direto da disponibilidade efetiva do serviço.

Cada métrica pode ser estatisticamente agregada (média, soma, máximo, percentil) sobre janelas de tempo configuráveis, e os alarmes do CloudWatch avaliam se essas agregações cruzam limiares por um número de períodos consecutivos antes de mudar de estado e disparar ações, desenho que evita reações a ruídos transitórios.

4.4 Infraestrutura como Código

Toda a infraestrutura foi descrita em Terraform e orquestrada por Terragrunt, em organização modular. Os módulos genéricos (em modules/) descrevem recursos reutilizáveis, como ASG, ALB, RDS e SQS, parametrizados por variáveis. Os ambientes (em environments/production/us-east-1/) instanciam esses módulos com valores concretos. O arquivo terragrunt.hcl na raiz centraliza configurações compartilhadas, como a definição do backend remoto, evitando repetição entre ambientes.

O fragmento a seguir ilustra a definição declarativa do Auto Scaling Group, na qual se observam os limites de capacidade, a referência ao launch template e a associação ao target group do balanceador.

modules/asg/main.tf

```
resource "aws_autoscaling_group" "app" {
  name                = var.name
  min_size            = 1
  desired_capacity    = 2
  max_size            = 6
  vpc_zone_identifier = var.subnet_ids
  target_group_arns   = [aws_lb_target_group.app.arn]
  health_check_type   = "ELB"

  launch_template {
    id      = aws_launch_template.app.id
    version = "$Latest"
  }
}
```

O estado do Terraform, arquivo que mapeia o que a ferramenta conhece sobre a infraestrutura real, foi mantido remotamente, e não localmente. Esse ponto é frequentemente subestimado, mas é decisivo para trabalho em equipe: o estado foi armazenado em um bucket S3 versionado e criptografado, com lock de concorrência implementado via tabela DynamoDB. O versionamento do S3 permite retornar a estados anteriores; o lock do DynamoDB impede que duas execuções simultâneas corrompam o estado ao escrever ao

mesmo tempo. O Terragrunt automatiza a geração dessa configuração de backend de forma DRY.

Esse arranjo é também a base prática para a adoção de GitOps, discutida na seção a seguir.

4.5 GitOps e os benefícios da IaC

A organização da infraestrutura em código versionado abre caminho para o fluxo GitOps, no qual o repositório Git se torna a fonte única da verdade sobre o estado desejado da infraestrutura. A Figura 4 ilustra esse fluxo.

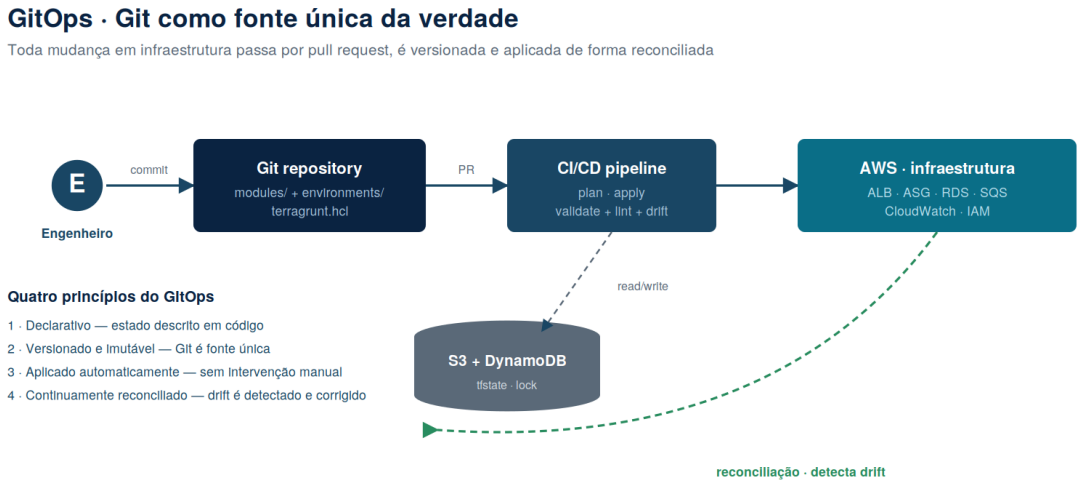

Figura 4 - Fluxo GitOps: Git como fonte única da verdade
Fonte: Elaborado pelo autor.

Sob esse modelo, nenhuma mudança é aplicada manualmente no ambiente: toda alteração é proposta via pull request, revisada por pares, mesclada e então propagada automaticamente pela esteira de CI/CD, que executa terraform plan e terraform apply. Os benefícios concretos dessa abordagem são:

- **Auditabilidade total:** o histórico de commits registra quem alterou o quê, quando e por quê, constituindo trilha de auditoria nativa.
- **Reversão simples:** em caso de incidente, reverter a infraestrutura ao último estado conhecido equivale a um git revert, em vez de uma intervenção manual sob pressão.

- **Colaboração segura:** o lock de estado e o fluxo de pull request impedem que engenheiros sobrescrevam o trabalho uns dos outros.
- **Detecção de drift:** a comparação contínua entre o estado descrito no Git e o estado real expõe divergências introduzidas por mudanças fora do processo.
- **Reprodutibilidade:** ambientes equivalentes podem ser recriados a partir do código, com previsibilidade, em dev, homologação e produção.

4.6 Estratégias de escalabilidade aplicadas

A motivação técnica para a escalabilidade horizontal foi discutida no referencial teórico. A Figura 5 sintetiza o contraste entre as estratégias vertical e horizontal, evidenciando por que a segunda foi adotada.

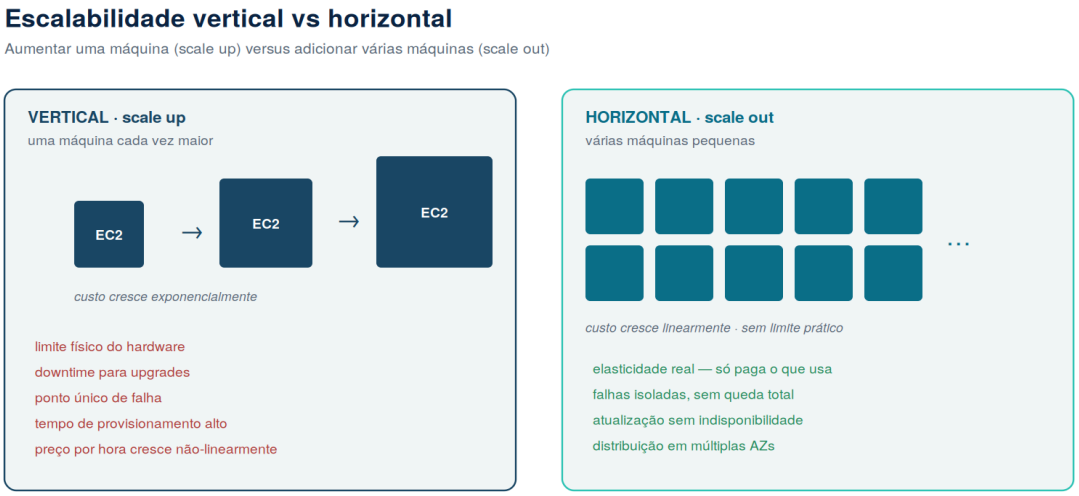

Figura 5 - Escalabilidade vertical (scale up) versus horizontal (scale out)
Fonte: Elaborado pelo autor.

Embora o experimento principal utilize a métrica de profundidade de fila do SQS como gatilho, é importante registrar que a mesma arquitetura suportaria políticas baseadas em CPU e em memória. Uma política baseada em CPU adicionaria instâncias quando a utilização média da frota ultrapassasse um limiar (por exemplo, 70% por dois períodos consecutivos), sendo apropriada para workloads CPU-bound. Uma política baseada em memória, que na AWS requer a publicação da métrica via CloudWatch Agent, por não ser nativa, seria adequada a aplicações que mantêm grandes estruturas em memória. A escolha pela métrica de fila, neste trabalho, deve-se à sua aderência direta ao modelo orientado a eventos: o tamanho da fila é o sinal mais fiel da carga de trabalho real pendente.

Foram configuradas duas políticas dinâmicas de escalonamento simples, ambas reagindo à métrica ApproximateNumberOfMessagesVisible:

- **Scale-out (HighSQS):** quando a fila possui 5 ou mais mensagens visíveis por 1 minuto, adiciona 2 instâncias EC2 ao ASG, com período de cooldown de 30 segundos antes de permitir nova ação.
- **Scale-in (LowSQS):** quando a fila permanece com 0 mensagens por mais de 3 minutos, remove 1 instância EC2 do ASG, otimizando custos em períodos ociosos.

A assimetria entre as políticas, adicionar de duas em duas e remover de uma em uma, é deliberada. Adicionar capacidade agressivamente protege contra a degradação sob carga (o custo de subprovisionar é insatisfação do usuário); remover conservadoramente evita oscilações (thrashing) e protege contra falsos vales de demanda.

4.7 Testes de escalabilidade

A validação consistiu em dois experimentos complementares, ambos acompanhados pelos dashboards do CloudWatch.

4.7.1 Teste de escalabilidade ascendente (scale-up)

Pela rota /send, foram publicadas múltiplas mensagens consecutivas na fila SQS. Quando a profundidade da fila ultrapassou 5 mensagens, o alarme HighSQS transitou para o estado de alarme e disparou a política de scale-out. Observou-se o provisionamento de 2 novas instâncias EC2 pelo ASG. O tempo médio decorrido entre o disparo do alarme e a entrada efetiva das novas instâncias em serviço, incluindo boot, execução do user-data e aprovação nos health checks do ALB, foi de aproximadamente 3 a 4 minutos. Durante todo o processo, 100% das requisições continuaram sendo roteadas a instâncias saudáveis, sem intervenção manual.

4.7.2 Teste de escalabilidade descendente (scale-down)

Pela rota /consume, as mensagens foram processadas e removidas da fila. Mantida a fila com 0 mensagens por mais de 3 minutos consecutivos, o alarme LowSQS disparou a política de scale-in, e o ASG removeu 1 instância. Verificou-se que apenas instâncias não-essenciais foram terminadas, preservando a capacidade mínima e a disponibilidade do serviço. A Figura 6 ilustra a complementaridade entre testes de carga e engenharia do caos como práticas de validação de resiliência.

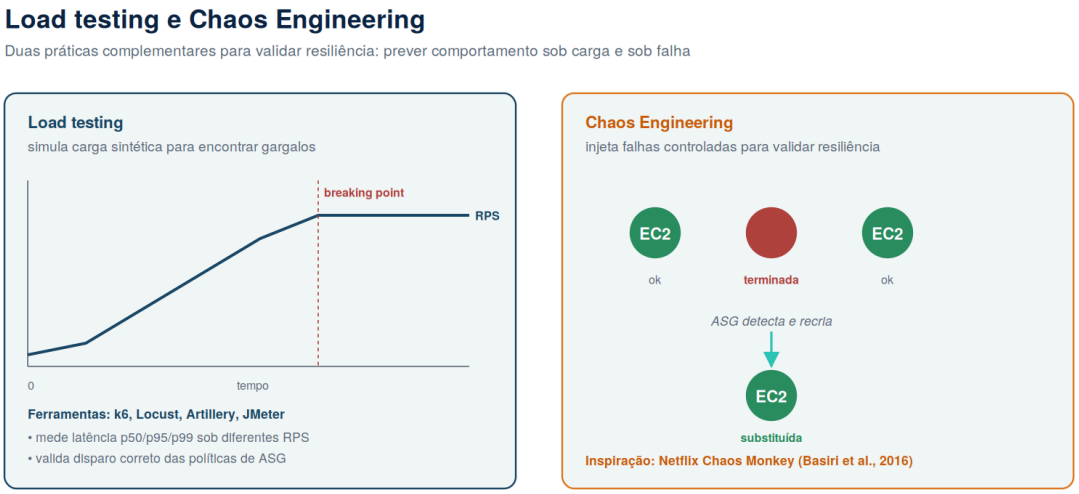

Figura 6 - Testes de carga e engenharia do caos como práticas complementares
Fonte: Elaborado pelo autor.

Cabe registrar que os testes conduzidos neste trabalho concentraram-se na validação das políticas de escalabilidade sob carga sintética. Como direção complementar, a engenharia do caos, descrita no referencial, permitiria validar empiricamente a tolerância a falhas: terminar deliberadamente uma instância em operação e verificar se o ASG a repõe sem impacto perceptível ao usuário, à maneira do Chaos Monkey da Netflix (BASIRI et al., 2016). Trata-se de extensão natural do experimento, indicada como trabalho futuro.

4.8 Análise de custos: máquina sempre ligada versus ASG elástico

Um dos argumentos econômicos mais fortes a favor da elasticidade é a redução de custo em relação ao superprovisionamento estático. Para tornar esse argumento concreto, modelaram-se dois cenários para um workload com picos previsíveis concentrados em horário comercial. A Figura 7 ilustra a comparação.

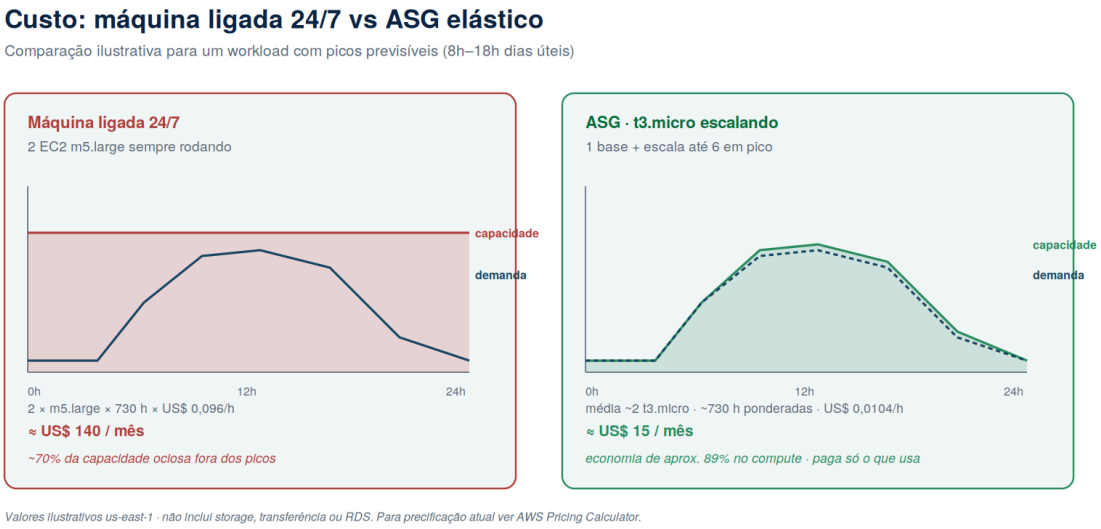

Figura 7 - Custo de máquina sempre ligada versus frota elástica via ASG
Fonte: Elaborado pelo autor.

No primeiro cenário, dimensiona-se a infraestrutura para o pico e mantém-se essa capacidade ligada 24 horas por dia. Tomando como exemplo duas instâncias de porte médio operando continuamente, a aproximadamente US$ 0,096 por hora cada, o custo mensal de compute aproxima-se de US$ 140, com a agravante de que boa parte dessa capacidade permanece ociosa fora dos horários de pico. No segundo cenário, com ASG sobre instâncias t3.micro (a cerca de US$ 0,0104 por hora) escalando entre uma instância base e o pico conforme a demanda, o custo médio mensal de compute aproxima-se de US$ 15. A Tabela 2 resume a comparação.

Tabela 2 - Comparação ilustrativa de custos de compute

Cenário	Configuração	Custo mensal aprox. (compute)
Máquina sempre ligada	2 × m5.large, 24/7	≈ US$ 140
ASG elástico	1 base + escala até 6 (t3.micro)	≈ US$ 15
Economia estimada	-	≈ 89%

Fonte: Elaborado pelo autor, com base em preços públicos da AWS (us-east-1).

Três ressalvas são necessárias. Primeiro, os valores são ilustrativos e referentes apenas ao compute: não incluem armazenamento (EBS), transferência de dados, o próprio RDS, o ALB ou o SQS, e os preços variam por região e ao longo do tempo; recomenda-se a consulta à calculadora oficial de preços da AWS para estimativas precisas. Segundo, as duas configurações não são estritamente equivalentes em capacidade de pico, mas a comparação ilustra o princípio: pagar pela capacidade média ajustada à demanda, em vez de pela capacidade máxima permanente. Terceiro, mecanismos adicionais de otimização, Savings

Plans, Reserved Instances para a capacidade de base e Spot Instances para a capacidade elástica, poderiam reduzir ainda mais o custo, ao preço de maior complexidade de gestão.

Um corolário importante diz respeito ao contraste entre uma única máquina grande e várias máquinas pequenas, mesmo a custo equivalente. A frota distribuída oferece tolerância a falhas (a perda de uma instância reduz a capacidade em uma fração, não a zero), atualização sem indisponibilidade (substituição rolante de instâncias) e granularidade fina de escalonamento. A máquina única, por outro lado, constitui ponto único de falha e exige reinício para qualquer mudança de capacidade. Assim, a escalabilidade horizontal entrega benefícios de resiliência que transcendem a mera economia.

4.9 AWS versus Azure: trade-offs de provedor

A arquitetura deste trabalho é fortemente acoplada a serviços gerenciados da AWS. É legítimo, portanto, indagar sobre as alternativas. A AWS e o Microsoft Azure são os dois maiores provedores de infraestrutura de nuvem; segundo a Synergy Research, no segundo trimestre de 2025 a AWS detinha cerca de 30% do mercado global de infraestrutura de nuvem, contra aproximadamente 20% do Azure (CARGOSON, 2025). A Tabela 3 sintetiza vantagens e desvantagens relativas relevantes para o tipo de arquitetura aqui estudada.

Tabela 3 - Comparação de alto nível entre AWS e Azure

Dimensão	AWS	Azure
Maturidade e portfólio	pioneira (2006), maior amplitude de serviços	robusta, forte em PaaS e identidade
Cobertura geográfica	maior número de regiões e AZs	ampla, forte em mercados híbridos
Integração corporativa	neutra quanto a ecossistema	forte com Windows, AD/Entra ID, Microsoft 365
Equivalente ao ASG	Auto Scaling Group	Virtual Machine Scale Sets (VMSS)
Modelo de cobrança	por hora ou por segundo	por minuto
Comunidade/documentação	maior volume de projetos e exemplos	crescente, forte suporte enterprise

Fonte: Elaborado pelo autor, com base em Cargoson (2025) e Veritis (2025).

Em síntese, a AWS oferece maior maturidade, amplitude de serviços e densidade de comunidade, fatores que favorecem o time-to-market e a disponibilidade de exemplos e profissionais. O Azure destaca-se em organizações já investidas no ecossistema Microsoft, com integração nativa de identidade e ferramentas corporativas, além de granularidade de cobrança por minuto. Para o escopo deste trabalho, a escolha pela AWS deveu-se à maturidade dos serviços empregados e à abundância de documentação e ferramental de IaC;

não obstante, a arquitetura proposta tem equivalentes diretos no Azure (VMSS no lugar do ASG, Azure Load Balancer/Application Gateway no lugar do ALB, Azure Database for PostgreSQL no lugar do RDS, Azure Queue Storage ou Service Bus no lugar do SQS, Azure Monitor no lugar do CloudWatch), o que reforça o caráter geral dos princípios discutidos.

5 CONCLUSÃO

Este trabalho partiu da questão de como as práticas DevOps e a automação de infraestrutura podem ser aplicadas para garantir escalabilidade, resiliência e eficiência operacional em ambientes de nuvem. Para respondê-la, projetou-se, implementou-se e avaliou-se uma arquitetura escalável na AWS, integralmente descrita como código, articulando Auto Scaling Groups, Application Load Balancer, Amazon RDS, Amazon SQS e Amazon CloudWatch sobre instâncias EC2 executando uma aplicação Node.js.

Quanto ao desafio da escalabilidade, a integração entre o ASG e o CloudWatch permitiu o ajuste automático da capacidade computacional em resposta à profundidade da fila do SQS, com provisionamento de novas instâncias em aproximadamente três a quatro minutos e contração da frota em períodos ociosos, sem qualquer intervenção manual. Quanto à alta disponibilidade, o ALB assegurou que apenas instâncias saudáveis recebessem tráfego, isolando falhas individuais; o RDS em modo Multi-AZ estendeu essa garantia ao plano de dados. Quanto ao acoplamento e ao tempo de resposta, o SQS viabilizou comunicação assíncrona e absorção de picos, conferindo resiliência à arquitetura. Quanto à observabilidade, o CloudWatch ofereceu visibilidade contínua e a base para ações corretivas automáticas.

O uso de Infraestrutura como Código com Terraform e Terragrunt, apoiado em estado remoto, versionado e em fluxo GitOps, trouxe padronização, versionamento, auditabilidade e reprodutibilidade à gestão da infraestrutura. A análise de custos evidenciou que a frota elástica pode reduzir o gasto de computação em ordem próxima a 89% frente ao superprovisionamento estático, ao mesmo tempo em que a distribuição em múltiplas instâncias e zonas entrega benefícios de resiliência que a máquina única não oferece.

Relativamente aos objetivos específicos, o estudo permitiu: (a) revisar conceitos e práticas da cultura DevOps a partir de literatura acadêmica e técnica de referência; (b) projetar e implementar a arquitetura escalável na AWS com Terraform e Terragrunt; (c) avaliar o comportamento da infraestrutura sob variações de carga por meio de testes de scale-up e scale-down monitorados via CloudWatch; e (d) analisar benefícios e limitações, incluindo custo e dependência de provedor, em contraste com alternativas autogerenciadas e com o Azure.

Reconhecem-se limitações. A infraestrutura foi avaliada em ambiente controlado de simulação, com cargas sintéticas que não reproduzem integralmente padrões reais de produção em larga escala. A análise concentrou-se em uma única aplicação web simples, e a generalização para sistemas com múltiplos microsserviços e dependências externas não foi

verificada empiricamente. Aspectos de segurança avançada, conformidade regulatória e custo operacional detalhado foram tratados de forma introdutória, constituindo oportunidades de aprofundamento.

Conclui-se, em resposta à questão de pesquisa, que a combinação de Infraestrutura como Código, automação reativa a eventos e monitoramento contínuo torna possível operar uma infraestrutura na nuvem de forma escalável, resiliente, auditável e economicamente eficiente, entregando os benefícios esperados da cultura DevOps mesmo no contexto de uma aplicação deliberadamente simples, e estabelecendo base sólida para evoluções futuras.

6 TRABALHOS FUTUROS

A arquitetura proposta constitui uma base sobre a qual diversas extensões são possíveis. Cinco direções destacam-se como especialmente promissoras.

A primeira é a redução da dependência de serviços proprietários por meio de equivalentes abertos ou autogerenciados, em prol de maior soberania tecnológica e portabilidade. Concretamente, poder-se-ia mapear o ASG sobre EC2 para Kubernetes com Horizontal Pod Autoscaler, o RDS PostgreSQL para PostgreSQL autogerenciado, o SQS para RabbitMQ ou Apache Kafka, o ALB para NGINX, HAProxy ou Traefik, e o CloudWatch para Prometheus com Grafana. Uma avaliação comparativa de FinOps, contemplando o custo total de propriedade, incluídas as horas de engenharia para operação, manutenção e segurança, permitiria contrastar rigorosamente a solução gerenciada com a autogerenciada.

A segunda é a incorporação sistemática de engenharia do caos. A partir das ferramentas e princípios discutidos no referencial (BASIRI et al., 2016), experimentos controlados de injeção de falha, terminação de instâncias, indução de latência de rede, simulação de indisponibilidade do banco, permitiram validar empiricamente a resiliência arquitetural, e não apenas presumi-la.

A terceira é a realização de testes de carga formais e reprodutíveis com ferramentas como k6 ou Locust, mapeando a curva de capacidade do sistema, identificando o ponto de quebra e calibrando com rigor os limiares e cooldowns das políticas de escalabilidade sob diferentes perfis de tráfego.

A quarta é a consolidação de um pipeline GitOps de ponta a ponta, com promoção automatizada entre ambientes, validação estática de IaC (lint, varredura de segurança, detecção de drift) e portões de qualidade, aproximando o experimento de um fluxo de produção maduro.

A quinta, e talvez a mais instigante, é a extensão do modelo de escalabilidade reativa para workloads de inteligência artificial, nos quais a métrica que dispara o escalonamento deixa de ser o tamanho da fila e passa a ser, por exemplo, o custo de inferência por requisição, a vazão de tokens por segundo ou a latência de modelos de linguagem. Nesse cenário, o roteamento poderia tornar-se semântico, encaminhando cada requisição ao modelo mais adequado conforme a intenção, e a observabilidade incorporaria métricas de qualidade de resposta e custo financeiro em tempo real, abrindo um campo fértil de investigação na interseção entre práticas de SRE e plataformas de IA.

REFERÊNCIAS

ARTAC, M.; BOROVSAK, T.; DI NITTO, E.; GUERRIERO, M.; TAMBURRI, D. A. DevOps: introducing infrastructure-as-code. In: IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). [S. l.]: IEEE, 2017. p. 497-498. DOI: 10.1109/ICSE-C.2017.162.

AMAZON WEB SERVICES (AWS). *Amazon EC2 Auto Scaling: documentation.* [S. l.], 2024. Disponível em: https://docs.aws.amazon.com/autoscaling/. Acesso em: 27 maio 2026.

BASIRI, A.; BEHNAM, N.; DE ROOIJ, R.; HOCHSTEIN, L.; KOSEWSKI, L.; REYNOLDS, J.; ROSENTHAL, C. Chaos engineering. *IEEE Software*, [S. l.], v. 33, n. 3, p. 35-41, 2016. DOI: 10.1109/MS.2016.60.

BECK, K. et al. *OpenGitOps principles.* [S. l.]: Cloud Native Computing Foundation, 2021. Disponível em: https://opengitops.dev/. Acesso em: 27 maio 2026.

BEYER, B.; JONES, C.; PETOFF, J.; MURPHY, N. R. *Site Reliability Engineering: how Google runs production systems.* Sebastopol: O'Reilly Media, 2016.

CARGOSON. *AWS vs Azure vs Google: cloud market share (2025).* [S. l.], 2025. Disponível em: https://www.cargoson.com/. Acesso em: 27 maio 2026.

CHAVES, F. B. A. O modelo DevOps para execução de integração contínua e entrega contínua. 2020. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação), Universidade Federal do Maranhão, São Luís, 2020.

EXAME. iFood fora do ar? Aplicativo passa por instabilidades neste sábado, 21. *Exame*, [S. l.], 21 jun. 2025. Disponível em: https://exame.com/tecnologia/. Acesso em: 27 maio 2026.

FORSGREN, N.; HUMBLE, J.; KIM, G. *Accelerate: the science of lean software and DevOps: building and scaling high performing technology organizations.* Portland: IT Revolution Press, 2018.

GOMES, T. G. Análise do processo de implantação da cultura DevOps em uma organização. 2020. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação), Universidade Federal da Paraíba, João Pessoa, 2020.

HUMBLE, J.; FARLEY, D. *Continuous Delivery: reliable software releases through build, test, and deployment automation.* Boston: Addison-Wesley, 2010.

INSTITUTO FEDERAL DO SERTÃO PERNAMBUCANO. Análise das plataformas para implementação da cultura DevOps: uma visão geral dos principais prestadores de serviços em nuvem. 2022. Trabalho de Conclusão de Curso, Instituto Federal do Sertão Pernambucano, [S. l.], 2022.

KIM, G.; DEBOIS, P.; WILLIS, J.; HUMBLE, J. *The DevOps Handbook: how to create world-class agility, reliability, and security in technology organizations.* Portland: IT Revolution Press, 2016.

KUMARA, I.; GARRIGA, M.; ROMEU, A. U. et al. The do's and don'ts of infrastructure code: a systematic gray literature review. *Information and Software Technology*, [S. l.], v. 137, 106593, 2021. DOI: 10.1016/j.infsof.2021.106593.

PAHL, C.; SEZEN, Ö. C.; HOFER, F. Artificial intelligence for infrastructure-as-code: a systematic literature review. Electronics, [S. l.], v. 15, n. 4, 755, 2026. DOI: 10.3390/electronics15040755.

MORRIS, K. *Infrastructure as Code: dynamic systems for the cloud age.* 2. ed. Sebastopol: O'Reilly Media, 2020.

PARNIN, C.; HELMS, E.; ATLEE, C.; BOUGHTON, H.; GHATTAS, M.; GLOVER, A.; HOLMAN, J.; MICCO, J.; MURPHY, B.; SAVOR, T.; STUMM, M.; WHITAKER, S.; WILLIAMS, L. The top 10 adages in continuous deployment. *IEEE Software*, [S. l.], v. 34, n. 3, p. 86-95, 2017. DOI: 10.1109/MS.2017.86.

QUATTROCCHI, G.; TAMBURRI, D. A. Infrastructure as code. *IEEE Software*, [S. l.], v. 40, n. 1, p. 37-40, 2023. DOI: 10.1109/MS.2022.3209558.

RAHMAN, A.; FARHANA, E.; PARNIN, C.; WILLIAMS, L. Gang of eight: a defect taxonomy for infrastructure as code scripts. In: 42nd International Conference on Software Engineering (ICSE). [S. l.]: ACM/IEEE, 2020. p. 752-764. DOI: 10.1145/3377811.3380409.

RAHMAN, A.; MAHDAVI-HEZAVEH, R.; WILLIAMS, L. A systematic mapping study of infrastructure as code research. *Information and Software Technology*, [S. l.], v. 108, p. 65-77, 2019. DOI: 10.1016/j.infsof.2018.12.004.

RICHARDSON, A. *GitOps: operations by pull request.* [S. l.]: Weaveworks, 2017. Disponível em: https://www.weave.works/blog/gitops-operations-by-pull-request. Acesso em: 27 maio 2026.

TECHTUDO. iFood fora do ar e com problemas? App tem bug e não abre, dizem usuários. *TechTudo*, [S. l.], 21 jun. 2025. Disponível em: https://www.techtudo.com.br/. Acesso em: 27 maio 2026.

UNIVERSIDADE DO VALE DO TAQUARI (UNIVATES). Implementação de infraestrutura como código para provisionamento automatizado de ambientes. 2022. Trabalho de Conclusão de Curso, Univates, Lajeado, 2022.

UNIVERSIDADE FEDERAL DE UBERLÂNDIA. Implementação de infraestrutura como código e integração CI/CD em ambiente de nuvem. 2025. Trabalho de Conclusão de Curso, Universidade Federal de Uberlândia, Uberlândia, 2025.

VERITIS. *AWS vs Azure cloud: a glance at comparison.* [S. l.], 2025. Disponível em: https://www.veritis.com/blog/aws-vs-azure-cloud-a-glance-at-comparison/. Acesso em: 27 maio 2026.

YIN, R. K. *Case study research and applications: design and methods.* 6. ed. Thousand Oaks: SAGE Publications, 2018.