UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Mateus Henrique Alves da Cruz

Implantação de sistemas usando CI/CD com Kubernetes e Istio

Uberlândia, Brasil

2025

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Mateus Henrique Alves da Cruz

Implantação de sistemas usando CI/CD com Kubernetes e Istio

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Ronaldo Castro de Oliveira

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2025

Mateus Henrique Alves da Cruz

Implantação de sistemas usando CI/CD com Kubernetes e Istio

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Trabalho aprovado. Uberlândia, Brasil, 10 de setembro de 2025:

Prof. Dr. Ronaldo Castro de Oliveira
Orientador

Prof. Dr. Daniel Duarte Abdala

Prof. Dr. Rodrigo Sanches Miani

Uberlândia, Brasil
2025

Dedico este trabalho aos meus pais e à minha família, pelo apoio incondicional e incentivo a perseguir meus sonhos, além de terem possibilitado a minha vida acadêmica em uma universidade federal de grande peso técnico como a UFU. À minha namorada, pelo carinho, paciência, compreensão e companheirismo durante todos os momentos desta jornada. Aos meus amigos que fizeram esta caminhada mais leve e prazerosa, compartilhando conhecimento e momentos inesquecíveis.

Agradecimentos

Agradeço à UFU e ao curso de Sistemas de Informação pela excelente formação acadêmica, que me possibilitou chegar tão longe na carreira técnica. Ao meu orientador que fez tanta questão do cumprimento da minha jornada acadêmica, além da atenção e orientação fundamental para o desenvolvimento deste trabalho. Aos funcionários da UFU que de alguma forma participaram da minha aventura e contribuíram para acender a minha paixão pela tecnologia.

A base de todo Estado é a educação da sua juventude

Resumo

Este trabalho apresenta a implementação de uma solução completa de Continuous Integration (CI) e Continuous Deployment (CD) utilizando tecnologias *cloud-native* modernas, desenvolvendo uma aplicação *full-stack* de gerenciamento de clientes com Go/Gin no *backend* e React/TypeScript no *frontend* sobre infraestrutura AWS robusta composta por Elastic Kubernetes Service (EKS), Istio *service mesh* e *pipelines* GitHub Actions automatizadas. A arquitetura utiliza Terraform para Infrastructure as Code (IaC), garantindo reprodutibilidade e versionamento da infraestrutura. O trabalho estabelece metodologia sistemática para arquiteturas *cloud-native* incluindo segregação de ambientes através de contas AWS separadas e automação completa de *deployment*. A validação prática foi realizada através de estudo de caso envolvendo implementação de nova funcionalidade que demonstrou a flexibilidade e a robustez da arquitetura, resultando em automação completa dos *pipelines* sem intervenção manual, *deployments* sem indisponibilidade e observabilidade abrangente com métricas, *logs* e *traces* integrados, representando um modelo replicável para modernização de práticas DevOps que demonstra como tecnologias emergentes resultam em sistemas resilientes, escaláveis e sustentáveis.

Palavras-chave: DevOps, Continuous Integration, Continuous Deployment, Kubernetes, Istio.

Lista de ilustrações

Lista de abreviaturas e siglas

AWS	*Amazon Web Services*
API	*Application Programming Interface* - Interface de Programação de Aplicações
ALB	*Application Load Balancer* - Balanceador de Carga de Aplicação
ARN	*Amazon Resource Name* - Nomes de recurso da Amazon
AZ	*Availability Zone* - Zona de disponibilidade
CI	*Continuous Integration* - Integração Contínua
CD	*Continuous Deployment* - Implantação Contínua
CapEx	*Capital Expenses* - Despesas de capital
CNCF	*Cloud Native Computing Foundation*
CDP	*Chrome DevTools Protocol*
CRUD	*Create, read, update and delete* - Criar, ler, atualizar e deletar
CIDR	*Classless Inter-Domain Routing* - Encaminhamento Entre Domínios Sem Classificação
DDD	*Domain Driven Design*
DNS	*Domain Name System* - Sistema de Nomes de Domínio
EKS	*Elastic Kubernetes Service*
EC2	*Elastic Compute Cloud*
ELB	*Elastic Load Balancer*
EBS	*Elastic Block Store*
ECR	*Elastic Container Registry*
gRPC	*gRPC Remote Procedure Calls* - Chamadas de Procedimento Remoto gRPC
HTTP	*Hypertext Transfer Protocol* - Protocolo de Transferência de Hipertexto
HCL	*HashiCorp Configuration Language*

IaC	*Infrastructure as Code* - Infraestrutura como código
IaaS	*Infrastructure as a Service* - Infraestrutura como serviço
IAM	*Identity and Access Management* - Gerenciamento de Identidade e Acesso
JSON	*JavaScript Object Notation* - Notação de Objeto JavaScript
JWT	*JSON Web Token*
LLM	*Large Language Model* - Modelos de Linguagem de Grande Escala
NIST	*National Institute of Standards and Technology* - Instituto Nacional de Padrões e Tecnologia dos Estados Unidos
NLB	*Network Load Balancer* - Balanceador de Carga de Rede
NPM	*Node Package Manager* - Gerenciador de pacotes do Node
OpEx	*Operational Expenses* - Despesas operacionais
OSI	*Open Systems Interconnection*
PaaS	*Platform as a Service* - Plataforma como serviço
REST	*Representational State Transfer*
RDS	*Relational Database Service*
RBAC	*Role-Based Access Control* - Controle de Acesso Baseado em Função
SaaS	*Software as a Service* - Software como serviço
SLA	*Service Level Agreement* - Acordo de nível de serviço
SQS	*Simple Queue Service*
SNS	*Simple Notification Service*
S3	*Simple Storage Service*
TLS	*Transport Layer Security* - Segurança de Camada de Transporte
URL	*Uniform Resource Locator* - Localizador Uniforme de Recursos
VMM	*Virtual Machine Monitor* - Monitor de Máquina Virtual
VPC	*Virtual Private Cloud*
YAML	*YAML Ain't Markup Language*

Sumário

1 Introdução

Com o avanço dos estudos nas áreas de Engenharia de *Software* e das metodologias de desenvolvimento de *software*, ficou cada vez mais clara a necessidade de se implantar novas versões de um determinado *software* de forma rápida e segura para atingir seu público final (PONTES; ARTHAUD, 2019). Desta forma, é possível coletar *feedbacks* dos usuários o mais cedo possível para realizar novas melhorias ou correções nessas versões, adquirindo assim vantagem competitiva no mercado atuante (COLOMO-PALACIOS et al., 2018).

De acordo com Thiago e Daniel (PONTES; ARTHAUD, 2019), essa popularização das metodologias ágeis também possibilitou que múltiplos times pequenos trabalhassem no desenvolvimento do mesmo *software*, popularizando a arquitetura de microsserviços, assim como a virtualização em contêiner. Tais abordagens trazem problemas a serem resolvidos, e para isso conceitos como orquestração de contêiner e *service mesh* também foram se popularizando e entrando em ascensão, completando todo o ecossistema de infraestrutura.

Continuous Integration (CI) é uma prática de desenvolvimento de *software* onde os desenvolvedores integram frequentemente suas mudanças de código em um repositório compartilhado, preferencialmente várias vezes ao dia, sendo cada integração verificada por uma construção automatizada que inclui a execução de testes para detectar erros de integração o mais rapidamente possível. A prática, cujo termo foi proposto inicialmente por Grady Booch em 1991 e posteriormente refinado por Kent Beck, visa reduzir riscos de entrega, diminuir o esforço de integração e promover um ambiente de desenvolvimento mais colaborativo e eficiente (FOWLER; FOEMMEL, 2006).

Continuous Deployment (CD) pode ser definido como um processo de desenvolvimento de *software* que busca sempre entregar rapidamente em ambientes produtivos novas mudanças incrementais de um *software*, em um processo automatizado que testa e implanta essas novas versões em ambientes produtivos (RAHMAN et al., 2015). Esse processo pode ser entendido como a parte final do processo completo de desenvolvimento de um *software*.

Para atingir tal rapidez e segurança no momento de implantar um *software* em ambiente produtivo, os estudos acerca do CD têm evoluído bastante e apresentado formas diferentes de implantar uma nova versão com mais segurança sem afetar diretamente toda a base de usuários simultaneamente (RAHMAN et al., 2015), evitando assim que sejam disponibilizadas novas funcionalidades ainda não testadas com o usuário final para toda a base de usuários, permitindo que sejam coletados dados acerca dessa funcionalidade antes de decidir disponibilizá-la para todos.

1.1 Justificativa

Diante da crescente necessidade de entregas rápidas e seguras de *software* em ambientes produtivos, aliada à popularização de arquiteturas de microsserviços e tecnologias de contêinerização, torna-se fundamental o estudo de estratégias de implantação que minimizem riscos e garantam alta disponibilidade dos sistemas. A combinação do Kubernetes como orquestrador de contêineres e do Istio como *service mesh* oferece um conjunto robusto de ferramentas para implementar estratégias de implantação progressiva, permitindo que organizações possam realizar *deployments* com maior confiança e controle. Este trabalho se justifica pela necessidade de compreender e documentar como essas tecnologias podem ser aplicadas de forma prática para resolver os desafios contemporâneos de *Continuous Deployment* em ambientes de produção.

1.2 Objetivos

O objetivo geral deste trabalho é explicar os conceitos importantes que são utilizados durante os processos de CI/CD, analisando um estudo de caso de uma esteira de implantação que aplica esses diversos conceitos na prática.

Como objetivos específicos, pretende-se:

- apresentar as funcionalidades do Kubernetes para orquestração de contêineres e gerenciamento de *deployments*;
- demonstrar como o Istio atua como *service mesh* para controle de tráfego e observabilidade;
- implementar uma esteira de CD funcional que integre essas tecnologias;
- e avaliar os benefícios e desafios da aplicação dessas estratégias em ambientes produtivos de microsserviços.

1.3 Método

Neste trabalho, materiais científicos e não científicos, mas de alta confiança, serão analisados acerca dos temas de desenvolvimento de *software* e CI/CD para serem utilizados como referencial teórico. Também será utilizado os conceitos descritos para implementar duas esteiras de implantação de um sistema *full-stack* para representar o estudo de caso. A metodologia de Estudo de Caso será empregada para demonstrar a aplicação prática dos conceitos de implantação progressiva. Segundo Yin (YIN, 2015), o estudo de caso é uma investigação empírica que investiga um fenômeno contemporâneo dentro de

seu contexto da vida real, especialmente quando os limites entre fenômeno e contexto não estão claramente definidos. Esta abordagem metodológica permite uma análise detalhada e aprofundada da implementação de estratégias de *deployment* progressivo utilizando Kubernetes e Istio, proporcionando *insights* valiosos sobre a viabilidade e eficácia dessas tecnologias em cenários reais de produção.

1.4 Organização do Trabalho

Este trabalho foi desenvolvido em 4 capítulos, sendo organizados e divididos nesta ordem:

- Capítulo 1: apresenta a introdução do trabalho e os objetivos e resultados esperados;
- Capítulo 2: fundamentação teórica e contextual acerca dos temas que serão trabalhados nesta monografia;
- Capítulo 3: principais ferramentas que serão usadas diretamente no estudo de caso deste trabalho;
- Capítulo 4: estudo e análise do estudo de caso apresentado neste trabalho;
- Capítulo 5: conclusões obtidas através das análises efetuadas no capítulo 4.

2 Fundamentação Teórica

Este capítulo contém as explicações de temas na área de computação necessárias para o entendimento dos serviços e ferramentas citados no Capítulo 3 e do estudo de caso analisado no Capítulo 4.

2.1 Integração Contínua

A Integração Contínua (Continuous Integration - CI) retrata uma prática de desenvolvimento de *software* que estimula os membros de uma equipe a integrarem seu trabalho de uma forma mais frequente. Cada integração é verificada por um processo automatizado de construção do artefato entregável e/ou executável (build) e testes, permitindo a detecção de erros de uma forma antecipada. De acordo com Fowler e Foemmel (FOWLER; FOEMMEL, 2006), a integração contínua se propõe a resolver problemas de integração que acontecem quando os desenvolvedores trabalham isoladamente de sua equipe por longos períodos, diminuindo significativamente o tempo dedicado à integração e permitindo que as equipes desenvolvam *softwares* coesos mais rapidamente.

A implementação eficaz da Integração Contínua exige uma mudança cultural que vai além da adoção de ferramentas típicas. Como ressaltam Shahin e outros (SHAHIN; BABAR; ZHU, 2017), a CI define um ciclo de *feedback* contínuo que aumenta a qualidade do código através de testes automatizados, análise estática de código e métricas de qualidade. Este ciclo de *feedback* possibilita que as equipes percebam e corrijam problemas logo após sua adição no código-base, resultando em uma base de código mais estável e de maior qualidade no longo prazo.

A Integração Contínua serve como fundamento para práticas mais avançadas como a Implantação Contínua (CD). Estas práticas, quando implementadas em conjunto, formam uma esteira de DevOps que automatiza totalmente o processo desde o *commit* do código até sua implantação em produção, quando será disponibilizado para o usuário final. Estudos experimentais conduzidos por Hilton e outros (HILTON et al., 2016) ressaltaram que organizações que adotam CI experimentam maior produtividade dos desenvolvedores, maior frequência de *releases* e melhoria significativa na qualidade do produto final, tornando a CI uma prática indispensável no desenvolvimento de *software* nos dias atuais.

2.2 Implantação Contínua

A Implantação Contínua (Continuous Deployment - CD) representa uma extensão natural da Integração Contínua, estabelecendo uma prática avançada de desenvolvimento de *software* onde cada alteração que passa pelos estágios automatizados de teste é liberada automaticamente para o ambiente de produção, removendo intervenções manuais no processo de disponibilização de *software*, permitindo entregas rápidas e frequentes de novas funcionalidades ao usuário final, conforme ressaltado por Chen (CHEN, 2015). A automação completa da esteira (*pipeline*) de implantação não só aumenta a eficiência operacional, mas também reduz notavelmente o risco associado às liberações de *software* através da padronização do processo e da eliminação de erros humanos.

A implementação bem-sucedida da Implantação Contínua requer uma infraestrutura robusta e uma cultura organizacional que valorize a automatização e a experimentação controlada. Segundo Humble e Farley (HUMBLE; FARLEY, 2010), organizações que aderem à implantação contínua desenvolvem capacidades avançadas de monitoramento e coleta de *feedback*, possibilitando a identificação de problemas em produção (que afetam usuários finais) de forma mais rápida e realizam *rollbacks* quando necessário. Esta capacidade de resposta rápida cria um ambiente onde as equipes se sentem seguras para experimentar e inovar, sabendo que potenciais problemas serão detectados e mitigados rapidamente, resultando em maior resiliência operacional.

A Implantação Contínua transforma principalmente a relação entre equipes de desenvolvimento e operações, representando um pilar essencial da cultura DevOps. De acordo com pesquisa conduzida por Leppänen e outros (LEPPÄNEN et al., 2015), organizações que implementam CD com sucesso experimentam benefícios significativos, incluindo ciclos de *feedback* mais curtos com clientes, maior satisfação dos desenvolvedores, redução do tempo de correção de *bugs* e aumento na frequência de disponibilização de novas funcionalidades ao usuário final. No entanto, os autores também ressaltam desafios importantes na adoção desta prática, como a necessidade de testes automatizados abrangentes, considerações de segurança e requisitos específicos do domínio que podem limitar a frequência de implantações automáticas em determinados contextos.

2.3 DevOps

DevOps simboliza uma abordagem cultural e organizacional que procura integrar os processos de desenvolvimento de *software* (Dev) e operações de TI (Ops), promovendo a colaboração contínua entre equipes tradicionalmente isoladas. Esta metodologia nasceu como resposta às limitações do modelo tradicional de entrega de *software*, onde equipes isoladas resultavam em ciclos de desenvolvimento lentos e propensos a falhas. Segundo Jabbari e outros (JABBARI et al., 2016), DevOps pode ser definido como um conjunto de

práticas projetadas para diminuir o tempo entre o *commit* de uma mudança em um sistema e a transferência dessa mudança para o ambiente produtivo, garantindo alta qualidade.

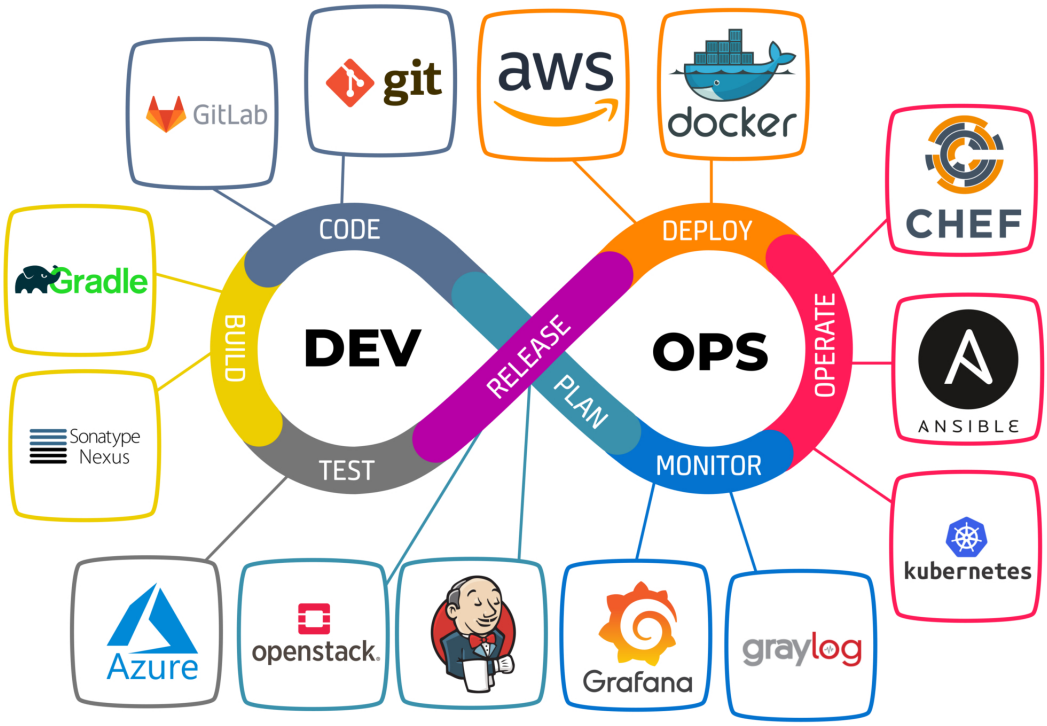

Figura 1 – Representação visual do modelo DevOps. Fonte: Extraído de (KULYK, 2019)

De acordo com o modelo visual de DevOps apresentado na Figura 1, o processo se inicia com o planejamento, onde são definidos os requisitos, funcionalidades e objetivos do projeto, seguido pela etapa de codificação, fase em que os desenvolvedores escrevem o código-fonte da aplicação. Na sequência, a etapa de *build* engloba a compilação e empacotamento do código, criando os artefatos executáveis, enquanto a etapa de teste executa validações automatizadas para garantir a qualidade do *software*. A fase de *release* prepara a versão para produção com aprovações e validações finais, precedendo a etapa de *deploy*, momento da efetiva implantação da aplicação no ambiente produtivo. O ciclo prossegue com a operação, etapa de execução e manutenção da aplicação em funcionamento, e termina no monitoramento, onde são coletadas métricas, *logs* e *insights* sobre performance e disponibilidade. Esta última fase alimenta continuamente o planejamento da próxima iteração, caracterizando a natureza cíclica e de melhoria contínua que define a metodologia DevOps, eliminando silos entre equipes e promovendo a entrega rápida e confiável de *software*. Ao longo do tempo ferramentas foram criadas para dar a suporte a cada etapa definida no ciclo de DevOps, como destacado na Figura 1.

A implementação bem-sucedida de DevOps tem como base princípios fundamentais que incluem automação, integração contínua, entrega contínua, monitoramento constante e *feedback* rápido. A automação de processos manuais e repetitivos não apenas reduz er-

ros humanos, mas também acelera o ciclo de desenvolvimento de *software*. Integração e entrega contínuas permitem que as alterações de código sejam testadas e implementadas rapidamente em ambientes de produção. Estas práticas são apoiadas por uma infraestrutura como código (IaC), que permite o provisionamento e gerenciamento automatizado da infraestrutura através de definições programáveis, como ressaltado por Ebert e outros (EBERT et al., 2016).

Os benefícios de DevOps para as organizações são substanciais e mensuráveis. Estudos apontam que empresas que adotam práticas de DevOps usufruem de melhorias significativas em métricas, incluindo redução no tempo de lançamento de novas funcionalidades, diminuição de falhas em produção e aumento na eficiência operacional. De acordo com a pesquisa conduzida por Forsgren e outros (FORSGREN; HUMBLE; KIM, 2018), organizações com alto desempenho em DevOps conseguem lançar alterações 46 vezes mais frequentemente e com 440 vezes menos tempo entre *commits* e implantações, quando comparadas com organizações de baixo desempenho. Além disso, essas empresas demonstram maior resiliência, com uma taxa de falha 7 vezes menor e uma capacidade 2.604 vezes mais rápida de recuperação de incidentes.

2.4 Microsserviços

Microsserviços simbolizam uma arquitetura voltada para o desenvolvimento de *software* que estrutura uma aplicação como uma coleção de serviços pequenos, autônomos e fracamente acoplados. Diferentemente das arquiteturas monolíticas tradicionais, onde toda a aplicação é construída como uma única unidade, os microsserviços permitem que componentes individuais sejam desenvolvidos, implantados e escalados de forma independente (NEWMAN, 2015). Essa autonomia e independência possibilita que diferentes times de desenvolvedores trabalhem de forma assíncrona e em paralelo, aderindo tecnologias e linguagens de programação específicas para cada microsserviço, dando assim mais autonomia para cada time escolher a melhor ferramenta que se encaixa para atender aos requisitos funcionais e não-funcionais de cada componente do sistema. Como exemplificado na Figura 2, vários microsserviços compõe uma única aplicação de um *e-commerce*, onde cada microsserviço se comunica da forma desejada e tem seu próprio isolamento da base de dados.

Um dos principais pilares da arquitetura de microsserviços é o conceito de *bounded context* (contexto delimitado), que é definido no *Domain-Driven Design* (DDD). De acordo com Evans (EVANS, 2004), cada microsserviço deve englobar uma funcionalidade específica do domínio de negócio, tendo sua respectiva base de dados de uma forma particular e suas interfaces bem definidas. Thoenes (THOENES, 2015) justifica que esta separação em contextos delimitados ajuda a coesão interna dos serviços enquanto reduz o

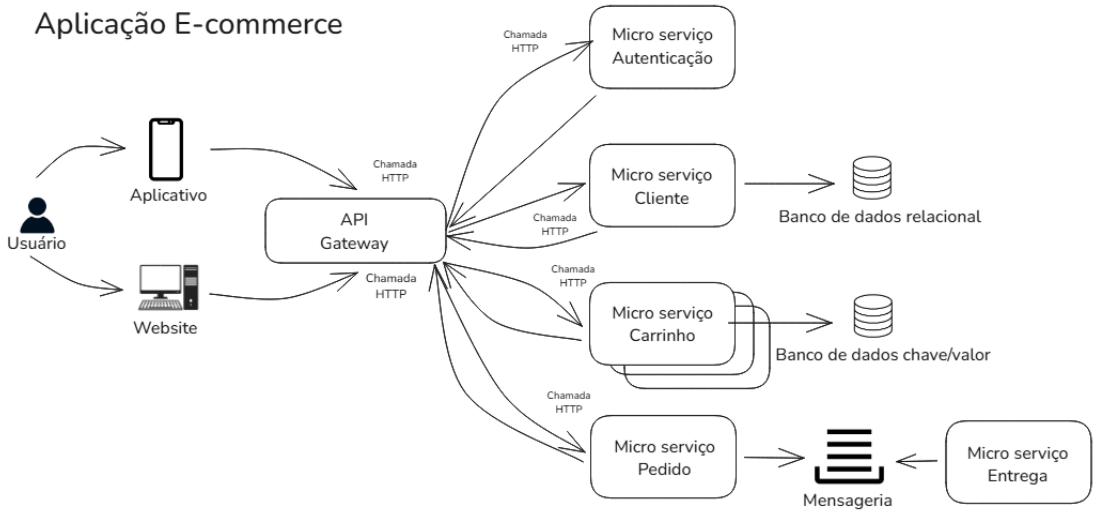

Figura 2 – Sistema de um *e-commerce* representado em uma arquitetura de microsserviços. Fonte: Do Autor.

acoplamento entre eles, resultando em sistemas mais resilientes e adaptáveis às mudanças nos requisitos de negócio. A comunicação entre os microsserviços é tipicamente realizada por meio de APIs REST, mensagens assíncronas como eventos ou protocolos como gRPC, estabelecendo limites claros entre os diferentes contextos da aplicação.

Implementar um sistema seguindo a arquitetura baseada em microsserviços traz vantagens significativas em termos de escalabilidade e resiliência. Segundo Dragoni (DRAGONI et al., 2017), microsserviços podem ser escalados de forma independente, permitindo que recursos computacionais sejam alocados de forma mais eficiente de acordo com o requisito computacional de cada componente, podendo assim também trazer benefícios em termos de custo com infraestrutura. Além disso, a natureza distribuída desta arquitetura favorece uma maior resiliência do sistema como um todo, pois falhas em um serviço não influenciam necessariamente o funcionamento dos demais. Esta característica é particularmente valiosa em ambientes de alta disponibilidade, onde a tolerância a falhas é um requisito crucial.

Estudos empíricos conduzidos por Balalaie e outros demonstram (BALALAIE; HEYDARNOORI; JAMSHIDI, 2016) que a adoção de microsserviços pode trazer ganhos significativos de produtividade e qualidade de *software*, especialmente em organizações com múltiplas equipes de desenvolvimento trabalhando em paralelo em diferentes aspectos de um mesmo sistema.

A arquitetura de microsserviços introduz alguns desafios relacionados à complexidade operacional e de gerenciamento. Fowler e Lewis (FOWLER; LEWIS, 2014) constatam que, enquanto arquiteturas monolíticas e centralizadas simplificam o processo de implantação, microsserviços necessitam de uma infraestrutura mais sofisticada para gerenciar múltiplos serviços distribuídos. O surgimento de tecnologias como contêineres

(Docker) e orquestradores (Kubernetes) tem sido fundamental para amenizar estes desafios, trazendo ferramentas para tornar o processo de empacotamento automatizado, a implantação e o gerenciamento do ciclo de vida dos microsserviços. Além disso, práticas de DevOps e CI/CD são frequentemente adotadas em conjunto com microsserviços para garantir processos de desenvolvimento ágeis e confiáveis, dando mais agilidade ao processo de entrega contínua.

2.5 Automação de testes

A automação de testes é uma disciplina fundamental da engenharia de *software* que envolve o uso de ferramentas especializadas, *scripts* e *frameworks* para executar casos de teste de forma automatizada, sem intervenção manual direta. Esta prática permite a verificação sistemática e repetível do comportamento de sistemas de *software*, garantindo que funcionalidades específicas operem conforme especificado nos requisitos. Segundo Sommerville (SOMMERVILLE, 2019), a automação de testes representa um componente essencial dos processos modernos de desenvolvimento de *software*, especialmente em metodologias ágeis e práticas de integração contínua, onde a necessidade de *feedback* rápido e confiável sobre a qualidade do código é crucial para manter a velocidade de entrega sem comprometer a estabilidade do produto.

Os benefícios da automação de testes são múltiplos e impactam diretamente a eficiência e a qualidade do desenvolvimento de *software*. A execução automatizada permite a realização de testes repetitivos de forma consistente, eliminando erros humanos e liberando recursos para atividades de maior valor agregado, como testes exploratórios e análise de cenários complexos. Além disso, a automação possibilita a execução de suítes de testes extensas em períodos reduzidos, facilitando a detecção precoce de defeitos e regressões no código. Como destacado por Fewster e Graham (FEWSTER; GRAHAM, 1999), a implementação efetiva de estratégias automatizadas pode resultar em reduções significativas no tempo de ciclo de desenvolvimento e nos custos associados à correção de defeitos encontrados tardiamente no processo.

Entretanto, a implementação bem-sucedida da automação de testes requer planejamento estratégico e considerações técnicas específicas. É fundamental identificar quais tipos de testes são mais adequados para automação, considerando fatores como frequência de execução, complexidade de implementação, estabilidade das funcionalidades e retorno sobre investimento. Testes de regressão, testes de carga e testes de APIs são exemplos de categorias que frequentemente se beneficiam da automação. A escolha de ferramentas apropriadas, a definição de arquiteturas de teste escaláveis e a manutenção contínua dos *scripts* automatizados são aspectos críticos que determinam o sucesso da iniciativa. Além disso, é importante reconhecer que a automação de testes não substitui completamente os

testes manuais, mas complementa e potencializa as atividades de garantia de qualidade, criando uma abordagem híbrida que maximiza a cobertura e eficácia dos esforços de teste.

2.6 Computação em nuvem

A computação em nuvem caracteriza um paradigma de tecnologia que transformou drasticamente a forma como empresas e pessoas utilizam recursos computacionais. De acordo com Mell e Grance (MELL; GRANCE, 2011), a computação em nuvem pode ser definida como um modelo que permite acesso global, conveniente e sob demanda a um conjunto compartilhado de recursos computacionais configuráveis (como redes, servidores, armazenamento, aplicações e serviços) que podem ser rapidamente provisionados e disponibilizados com mínimo esforço de gestão ou interação com o provedor de serviços. Esta definição, estabelecida pelo Instituto Nacional de Padrões e Tecnologia dos Estados Unidos (NIST), tornou-se amplamente aceita e determina cinco características cruciais: autoatendimento sob demanda, amplo acesso à rede, agrupamento de recursos, elasticidade rápida e serviço medido.

Serviços em computação em nuvem são tradicionalmente categorizados em três camadas principais: *Software* como Serviço (*Software as a Service* - SaaS), Plataforma como Serviço (*Platform as a Service* - PaaS) e Infraestrutura como Serviço (*Infrastructure as a Service* - IaaS). Zhang e Chang afirmam (ZHANG; CHENG; BOUTABA, 2010) que estes modelos refletem diferentes níveis de abstração oferecidos aos usuários da nuvem, onde cada modelo provê capacidades diferentes e transfere distintas responsabilidades de gerenciamento do cliente para o provedor de serviços. O SaaS disponibiliza aplicações completas hospedadas na nuvem, o PaaS oferece ambientes de desenvolvimento e implantação, enquanto o IaaS provê recursos computacionais virtualizados como servidores e armazenamento, dando maior flexibilidade e controle por parte do usuário.

A adoção da computação em nuvem traz benefícios significativos para as organizações, incluindo redução de custos operacionais, maior escalabilidade, agilidade de negócios aprimorada e acesso a tecnologias avançadas que estão constantemente sendo atualizadas com novas funcionalidades pelos provedores de serviços. Armbrust e outros (ARMBRUST et al., 2010) argumentam que a computação em nuvem elimina a necessidade de planejamento antecipado por parte dos usuários, permitindo que empresas comecem com recursos limitados e expandam sua infraestrutura de acordo com o crescimento da demanda. Esta característica de elasticidade representa uma mudança fundamental na economia da computação, transformando grandes despesas de capital (CapEx) em despesas operacionais (OpEx) variáveis e reduzindo as barreiras de entrada para novas empresas no mercado digital.

Apesar dos benefícios, a migração para ambientes de nuvem apresenta desafios

relacionados à segurança, privacidade, conformidade regulatória e também cria uma relação de dependência com os fornecedores do serviço. Buyya e outros observam (BUYYA et al., 2009) que questões como interoperabilidade entre diferentes provedores de nuvem, garantias de nível de serviço (*Service Level Agreement* - SLAs), proteção de dados e recuperação de desastres são preocupações críticas para organizações que consideram a adoção da nuvem. Além disso, os autores salientam a importância de desenvolver arquiteturas orientadas a serviços que possam se adaptar às necessidades específicas das organizações enquanto aproveitam a escalabilidade e flexibilidade características dos ambientes de nuvem.

A recente evolução da computação em nuvem tem sido caracterizada pelo surgimento de modelos híbridos e *multicloud*, que combinam infraestruturas privadas com serviços públicos de nuvem ou utilizam serviços de múltiplos provedores simultaneamente. Conforme destacado por Jamshidi e outros (JAMSHIDI; AHMAD; PAHL, 2013), essas abordagens permitem que as organizações otimizem seus investimentos em TI, evitem o acoplamento a um único fornecedor e atendam a requisitos específicos de desempenho, conformidade e localização de dados. Os autores também ressaltam o crescimento de tecnologias emergentes como computação de borda (*edge computing*), que estende o paradigma da nuvem para mais perto dos usuários finais, e tecnologias de conteinerização como Docker e Kubernetes, que facilitam a portabilidade e o gerenciamento de aplicações em ambientes distribuídos.

2.7 Virtualização

A virtualização representa uma tecnologia fundamental na computação moderna, possibilitando a abstração de recursos físicos de *hardware* em ambientes virtuais independentes. Esse processo permite que múltiplos sistemas operacionais e aplicações compartilhem os mesmos recursos físicos, otimizando a utilização do *hardware* e proporcionando maior flexibilidade operacional. Conforme destacado por Sahoo e outros (SAHOO; MOHAPATRA; LATH, 2010), a virtualização transformou radicalmente a infraestrutura de TI ao permitir consolidação de servidores, melhor gerenciamento e aproveitamento de recursos e significativa redução de custos operacionais, estabelecendo-se como um pilar central para a computação em nuvem e modernos centros de dados.

O conceito de virtualização não é recente, tendo suas origens na década de 1960 com o desenvolvimento dos *mainframes* da IBM, que possibilitavam a divisão de recursos computacionais entre diferentes usuários. No entanto, como ressaltam Rosenblum e Garfinkel (ROSENBLUM; GARFINKEL, 2005), foi a partir dos anos 2000 que a tecnologia ganhou ampla adesão com o surgimento de soluções como VMware e Xen, impulsionadas pela necessidade de melhor aproveitamento de recursos computacionais em servidores

x86. A partir de então, a virtualização evoluiu consideravelmente, abrangendo não apenas servidores, mas também redes, armazenamento e aplicações, estabelecendo-se como um componente essencial para implementação de infraestruturas ágeis e escaláveis.

As vantagens da virtualização são diversas e impactam diretamente a eficiência operacional das organizações. Entre os principais benefícios, evidenciam-se: a consolidação de servidores, que diminui o número de equipamentos físicos necessários; o isolamento de falhas, que aumenta a confiabilidade; a facilidade de *backup* e recuperação; além da possibilitação de criar ambientes de teste isolados. Estudos experimentais comprovam que implementações de virtualização podem resultar em economias de até 80% em custos com *hardware* e energia, além de possibilitar uma recuperação de desastres mais eficiente e melhor utilização de recursos computacionais (DANIELS, 2019).

Existem diferentes tipos de virtualização, cada um atendendo a necessidades específicas de infraestrutura de TI. A virtualização de servidores permite a execução de múltiplos sistemas operacionais em um único servidor físico; a virtualização de rede abstrai recursos de rede como *switches* e roteadores; a virtualização de armazenamento consolida múltiplos dispositivos físicos em um único *pool* lógico; enquanto a virtualização de *desktop* fornece ambientes de trabalho personalizados independentemente do *hardware* utilizado. Cada modalidade apresenta características próprias e pode ser implementada por meio de diferentes tecnologias, sendo as mais comuns baseadas em *hypervisors* ou contêineres (ZHANG et al., 2018).

2.7.1 Hypervisor

O *hypervisor*, também conhecido como Monitor de Máquina Virtual (VMM), é um componente de *software* fundamental para a virtualização tradicional, responsável por criar, gerenciar e orquestrar máquinas virtuais. Existem dois tipos principais: os *hypervisors* tipo 1 (*bare-metal*), que operam diretamente sobre o *hardware* sem necessidade de um sistema operacional hospedeiro, como VMware ESXi, Microsoft Hyper-V e Xen; e os *hypervisors* tipo 2 (*hosted*), que funcionam como aplicações sobre um sistema operacional, como VirtualBox e VMware Workstation. As diferenças entre os dois tipos de virtualização estão destacados na Figura 3

Segundo Popek e Goldberg (POPEK; GOLDBERG, 2017), um *hypervisor* eficiente deve garantir três propriedades fundamentais: equivalência, onde programas executados em ambientes virtualizados comportam-se identicamente à execução direta no *hardware*; controle de recursos, assegurando que o *hypervisor* mantenha controle total sobre os recursos virtualizados; e eficiência, minimizando a sobrecarga de processamento.

A evolução dos *hypervisors* integrou técnicas avançadas para otimizar desempenho e segurança, como a virtualização assistida por *hardware* (Intel VT-x e AMD-V), que

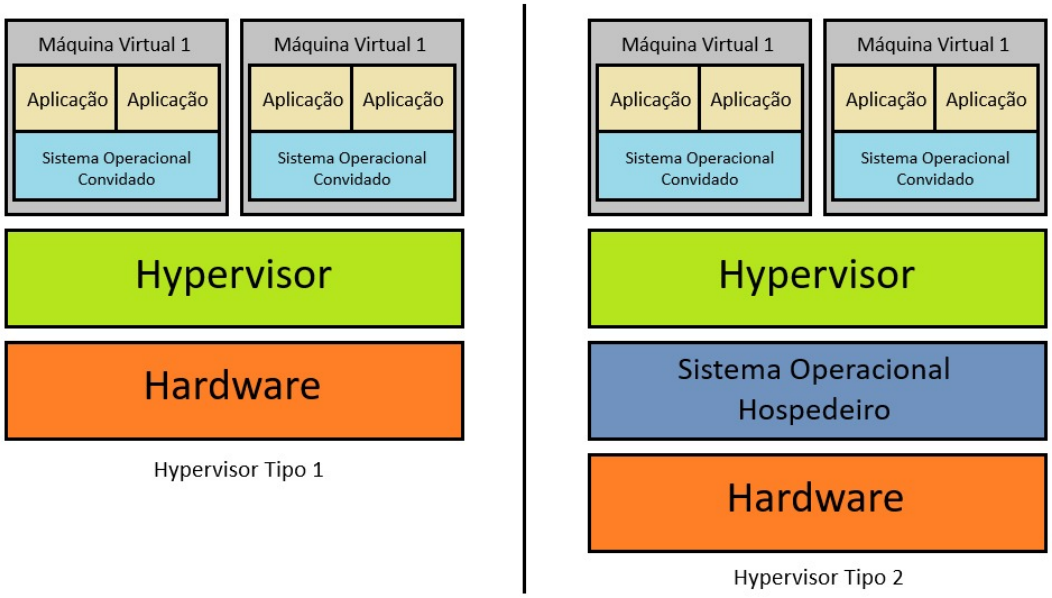

Figura 3 – Modelos de virtualização tipo 1 (*bare metal*) e tipo 2 (*hosted*). Fonte: Do Autor.

reduziu notavelmente a sobrecarga computacional ao transferir para o *hardware* funções previamente executadas por *software*. Outras inovações incluem a memória compartilhada entre máquinas virtuais, a migração ao vivo (*live migration*) que permite mover máquinas virtuais entre servidores físicos sem interrupção de serviço, e mecanismos de alocação dinâmica de recursos. Um estudo comparativo conduzido por Ahmad e outros (AHMAD; REHMAN; SAID, 2021) demonstrou que *hypervisors* modernos podem atingir eficiência próxima a 95% em comparação com sistemas não virtualizados em cargas de trabalho específicas, evidenciando os *hypervisors* como opções viáveis para virtualização de sistemas críticos com alto requisito de desempenho.

2.7.2 Contêiner

A tecnologia de contêineres simboliza uma abordagem de virtualização em nível de sistema operacional, diferenciando-se fundamentalmente dos *hypervisors* por compartilhar o *kernel* do sistema operacional hospedeiro entre os contêineres, como destacado pela Figura 4. Docker, introduzido em 2013, popularizou esta tecnologia ao oferecer um ecossistema completo para desenvolvimento, distribuição e execução de aplicações conteinerizadas. Diferentemente das máquinas virtuais tradicionais, os contêineres são extremamente leves, iniciando em segundos e consumindo significativamente menos recursos, pois eliminam a necessidade de virtualizar um sistema operacional completo para cada instância. Conforme analisado por Bernstein (BERNSTEIN, 2014), esta característica torna os contêineres ideais para arquiteturas de microsserviços e implementações de práticas DevOps, onde leveza, portabilidade e rápida inicialização são requisitos críticos.

O ecossistema de contêineres evoluiu rapidamente nos últimos anos, integrando

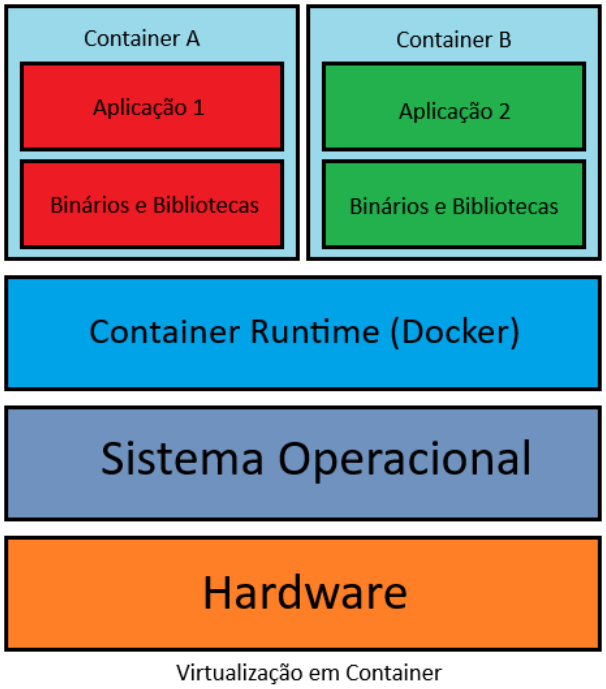

Figura 4 – Modelo de virtualização em contêiner. Fonte: Do Autor.

orquestradores como Kubernetes, que automatizam o *deployment*, escalabilidade e gerenciamento de aplicações conteinerizadas em ambientes distribuídos. Pahl e outros (PAHL et al., 2019) destacam que a adoção de contêineres cresceu exponencialmente devido à sua compatibilidade com práticas modernas de desenvolvimento de *software*, permitindo consistência entre ambientes de desenvolvimento, teste e produção por meio do conceito "*build once, run anywhere*". Além disso, a arquitetura de contêineres facilita implementações de computação em borda (*edge computing*) e computação em névoa (*fog computing*), onde recursos computacionais limitados demandam soluções de virtualização com baixa sobrecarga. Embora ofereçam isolamento menos robusto que máquinas virtuais tradicionais, avanços tecnológicos como *namespaces*, *cgroups* e *capabilities* no Linux têm fortalecido os aspectos de segurança, tornando contêineres cada vez mais adequados para aplicações empresariais críticas.

2.8 Infraestrutura como Código

Infraestrutura como Código (*Infrastructure as Code* - IaC) representa um paradigma moderno de gestão e provisão de infraestrutura computacional por meio de arquivos de configuração versionáveis, em vez de processos manuais tradicionais. Esta metodologia permite que equipes de operações e desenvolvedores tratem a infraestrutura tecnológica como *software*, aplicando práticas já consolidadas de desenvolvimento como controle de versão, testes automatizados, integração contínua e revisão por pares. Segundo Artac e

outros (ARTAC et al., 2017), a IaC não apenas simplifica a implantação de infraestrutura, mas também contribui positivamente para a qualidade de serviço e confiabilidade dos sistemas, ao reduzir a variabilidade e o potencial de erro humano durante o provisionamento.

A implementação de Infraestrutura como Código está intrinsecamente ligada aos princípios do movimento DevOps, promovendo maior colaboração entre as equipes de desenvolvimento e operações, permitindo a criação de definições declarativas ou procedurais que especificam o estado desejado da infraestrutura, possibilitando implantações repetíveis e previsíveis em diversos ambientes. Esta característica é particularmente valiosa em ambientes de nuvem, onde recursos podem ser provisionados e desprovisionados dinamicamente, adequando-se às necessidades de escalabilidade e elasticidade dos sistemas modernos.

A adoção de IaC oferece benefícios quantificáveis em termos de velocidade de entrega, consistência de ambiente e redução de custos operacionais. Um estudo conduzido por Morris (MORRIS, 2020) demonstrou que organizações que utilizam práticas maduras de IaC obtiveram redução média de 70% no tempo de provisionamento de ambientes e diminuição de 30% em incidentes relacionados a inconsistências de configurações. Além disso, a documentação fornecida pelos *scripts* de IaC aumenta a visibilidade e o entendimento da infraestrutura entre todos os membros da equipe, facilitando a transferência de conhecimento e reduzindo a dependência de especialistas específicos sobre determinado provisionamento ou configuração, representando um avanço significativo na gestão do conhecimento organizacional no contexto da engenharia de *software*.

2.9 Orquestração de Contêiner

A orquestração de contêineres surgiu como uma tecnologia fundamental para gerenciar aplicações em ambientes de microsserviços, permitindo automatizar o *deployment*, o escalonamento e as operações de contêineres em *clusters* distribuídos. Esta abordagem resolve desafios críticos da conteinerização, como balanceamento de carga, descoberta de serviços e autorrecuperação, que se tornam complexos em ambientes de grande escala (BURNS et al., 2016a). A transformação digital nas organizações tem impulsionado a adoção dessa tecnologia, que permite maior agilidade no ciclo de desenvolvimento e entrega de *software*, além de otimizar recursos computacionais por meio do empacotamento eficiente de aplicações. Plataformas como Docker Swarm e Kubernetes tornaram-se pilares dessa tecnologia, viabilizando a automação de tarefas operacionais e a otimização de recursos computacionais.

Os sistemas de orquestração de contêineres são constituídos por diversos componentes fundamentais que trabalham em conjunto para garantir o funcionamento harmônico das aplicações distribuídas. Esses sistemas geralmente implementam funcionalidades

como balanceamento de carga, descoberta de serviços, gerenciamento de configuração, monitoramento de saúde e autorrecuperação (*auto-healing*). O componente central desses sistemas é o *scheduler*, responsável por determinar onde os contêineres serão executados com base em políticas predefinidas e na disponibilidade de recursos. Além disso, os sistemas modernos de orquestração oferecem recursos avançados como atualizações progressivas, estratégias de implantação *blue-green*, *canary deployments* e capacidade de responder automaticamente a falhas, garantindo a continuidade operacional das aplicações em cenários atípicos. Segundo Pahl e outros (PAHL et al., 2019), a orquestração eficiente deve considerar requisitos não funcionais como disponibilidade, desempenho e segurança, exigindo políticas sofisticadas de agendamento e alocação de recursos.

A adoção da orquestração de contêineres em ambientes empresariais apresenta benefícios relevantes, mas também introduz desafios técnicos e organizacionais. Entre as vantagens, destacam-se a utilização eficiente de recursos, redução de custos operacionais, agilidade no ciclo de desenvolvimento, portabilidade entre ambientes e isolamento de aplicações. Entretanto, desafios como complexidade técnica, necessidade de novas habilidades, considerações de segurança e dificuldades de integração com sistemas legados ainda representam barreiras significativas para muitas organizações. Apesar desses desafios, a orquestração de contêineres continua em evolução constante, incluindo avanços em áreas como segurança, observabilidade, gerenciamento de estado e federação de *clusters*, consolidando-se como um componente essencial da infraestrutura moderna de TI e facilitando a transição para arquiteturas de microsserviços e computação em nuvem.

2.10 Service Mesh

Service Mesh é uma camada de infraestrutura dedicada que gerencia a comunicação entre serviços em arquiteturas de microsserviços. Conforme definido por Calcote e Butcher (CALCOTE; BUTCHER, 2019), um *Service Mesh* consiste em um plano de controle e um plano de dados que trabalham em conjunto para gerenciar o tráfego de rede, impor políticas e coletar métricas sem exigir alterações no código da aplicação. Esta abordagem surgiu como resposta aos desafios crescentes de comunicação entre serviços em ambientes distribuídos complexos, onde questões como descoberta de serviços, balanceamento de carga, tolerância a falhas e segurança tornaram-se preocupações críticas para desenvolvedores e arquitetos de sistemas.

A arquitetura típica de um *Service Mesh* implementa o padrão de *proxy sidecar*, onde cada instância de serviço é acompanhada por um *proxy* que intercepta todas as comunicações de entrada e saída. Segundo o estudo conduzido por Li e outros (LI et al., 2021), essa arquitetura permite que funcionalidades como autenticação *mTLS* (TLS mútuo), autorização, monitoramento detalhado e controle de tráfego sejam aplicadas de

maneira consistente em toda a malha de serviços. O plano de controle centraliza a configuração e orquestração desses *proxies*, permitindo que administradores definam políticas globais que são automaticamente distribuídas e aplicadas em toda a infraestrutura, sem necessidade de intervenção nos serviços individuais.

A implementação de um *Service Mesh* oferece benefícios significativos para as organizações que operam sistemas distribuídos complexos, incluindo maior observabilidade, segurança aprimorada e resiliência operacional. No entanto, também apresenta desafios relacionados à complexidade adicional e ao potencial impacto no desempenho. Uma pesquisa abrangente realizada por Nascimento e outros (NASCIMENTO; SILVA; OLIVEIRA, 2023) analisou 32 implementações de *Service Mesh* em ambientes de produção e identificou que, embora a sobrecarga de latência média seja de aproximadamente 10ms por requisição, os ganhos em termos de observabilidade e controle superaram significativamente esse custo em cenários onde a complexidade da rede de serviços ultrapassava 20 microsserviços interconectados.

3 Metodologia

Este capítulo tem por objetivo apresentar e analisar as principais tecnologias relacionadas às áreas específicas da computação abordadas no Capítulo 2. Serão examinadas suas respectivas arquiteturas, detalhados seus funcionamentos operacionais e discutidas suas aplicações práticas no contexto estudado.

3.1 AWS

A Amazon Web Services (AWS) é uma plataforma de computação em nuvem que oferece mais de 200 serviços completos de *data centers* distribuídos globalmente. O funcionamento da AWS baseia-se no modelo de computação em nuvem no qual os clientes pagam apenas pelos recursos que utilizam, sem necessidade de investimentos iniciais em infraestrutura física. Este modelo permite escalabilidade instantânea, alta disponibilidade, baixa latência e redundância geográfica por meio de suas regiões e zonas de disponibilidade, como descrito na Figura 5. Cada região AWS é uma área geográfica separada que contém múltiplas zonas de disponibilidade isoladas, mas conectadas por redes de baixa latência, criando um sistema resiliente contra falhas em *data centers* individuais (AWS, 2023). Alguns poucos serviços como Route 53, IAM e CloudFront, por exemplo, são ofertados de forma global (sem estar atrelado a uma região) devido aos seus requisitos funcionais.

A arquitetura da AWS é construída sobre um modelo de responsabilidade compartilhada, no qual a AWS gerencia a segurança da nuvem (infraestrutura física, rede, virtualização) enquanto os clientes são responsáveis pela segurança na nuvem (configurações, dados, aplicações). Esta arquitetura é distribuída em camadas: a camada de *hardware* físico gerenciada exclusivamente pela AWS; a camada de virtualização que abstrai os recursos físicos; a camada de serviços que inclui as APIs e interfaces; e a camada de aplicações onde os clientes implantam suas aplicações. Segundo Bhardwaj e outros (BHARDWAJ; JAIN; JAIN, 2018), esta arquitetura multicamadas permite que a AWS mantenha alta disponibilidade enquanto proporciona flexibilidade total para os clientes.

Os principais serviços da AWS podem ser categorizados em computação, armazenamento, banco de dados, rede e segurança e alguns exemplos são:

- **EC2 (Elastic Compute Cloud)** - Fornece servidores virtuais redimensionáveis
- **EKS (Elastic Kubernetes Service)** - Fornece um serviço gerenciado de Kubernetes

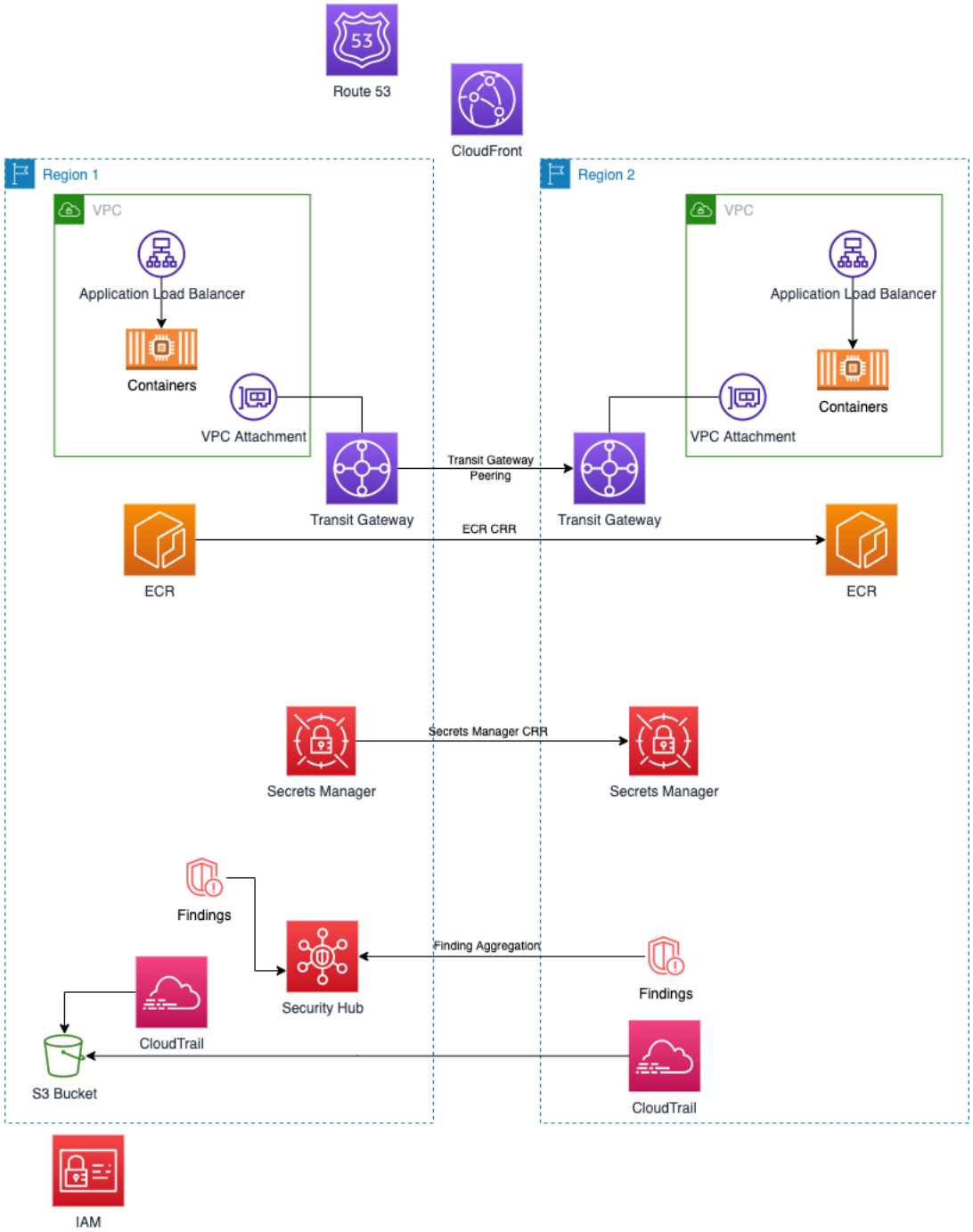

Figura 5 – Exemplo de arquitetura *multi-region* na AWS. Fonte: Extraído de (AWS, 2021).

- **Lambda** - Oferece computação sem servidor baseada em eventos
- **SQS (Simple Queue Service)** - Fornece um serviço de filas de mensageria para sistemas distribuídos
- **SNS (Simple Notification Service)** - Fornece um serviço de tópicos pub/sub de mensagens

- **ELB (Elastic Load Balancer)** - Distribui o tráfego de rede para aprimorar a escalabilidade da aplicação
- **S3 (Simple Storage Service)** - Proporciona armazenamento de objetos altamente escalável
- **EBS (Elastic Block Store)** - Fornece volumes de armazenamento em bloco
- **RDS (Relational Database Service)** - Gerencia bancos de dados relacionais
- **DynamoDB** - Oferece um banco de dados não relacional de alta performance
- **VPC (Virtual Private Cloud)** - Isola recursos em redes virtuais privadas
- **Route 53** - Oferece serviços de DNS
- **IAM (Identity and Access Management)** - Controla o acesso aos recursos

A integração desses serviços permite que as organizações implementem arquiteturas complexas e escaláveis. Como destacado por López e outros (LóPEZ; LOBATO; DUARTE, 2022) em seu estudo comparativo sobre plataformas de nuvem, a AWS destaca-se pela maturidade de seus serviços e pela profundidade de suas ofertas em cada categoria, permitindo desde simples aplicações *web* até complexos sistemas distribuídos de inteligência artificial e *machine learning*. A AWS também lidera em termos de recursos para desenvolvimento, como o AWS CloudFormation para infraestrutura como código e o AWS CodePipeline para CI/CD, permitindo que equipes adotem práticas modernas de DevOps. Essas capacidades tornaram a AWS a plataforma de escolha para empresas de todos os tamanhos, desde *startups* até corporações multinacionais que buscam transformação digital.

3.2 Docker

Docker é uma plataforma de conteinerização que modernizou o desenvolvimento e implantação de aplicações. Utilizando a tecnologia de contêineres, o Docker permite encapsular aplicações e suas dependências em unidades padronizadas e isoladas, garantindo consistência em diferentes ambientes. Este isolamento é alcançado através de recursos do *kernel* Linux como *namespaces*, que isolam recursos do sistema (processos, rede, sistema de arquivos) criando visões independentes para cada contêiner, e *cgroups*, que limitam e controlam o uso de recursos computacionais (CPU, memória, I/O), garantindo ambientes seguros e independentes sem a sobrecarga de máquinas virtuais tradicionais. A portabilidade oferecida pelo Docker simplifica consideravelmente o processo de desenvolvimento, permitindo que desenvolvedores criem aplicações em ambientes locais com a garantia de que funcionarão da mesma forma em ambiente produtivo (Docker Inc., 2023).

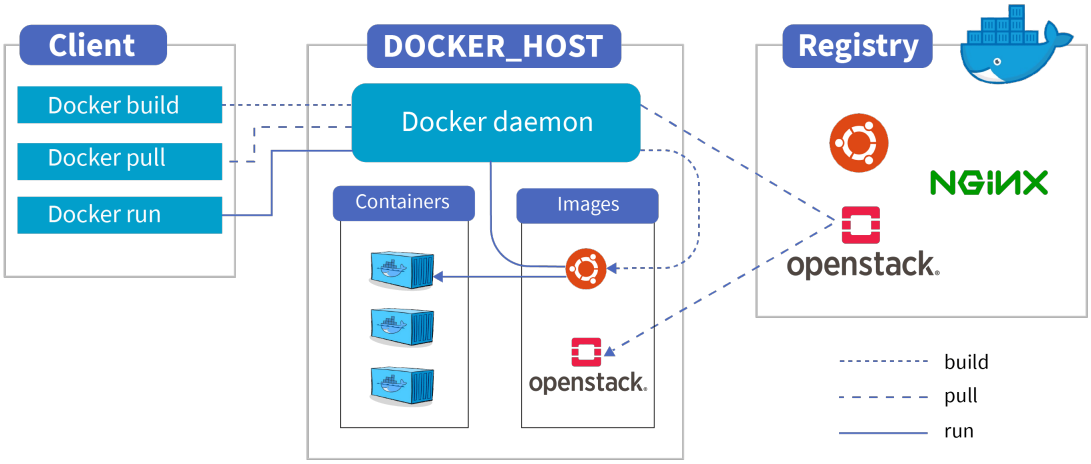

Figura 6 – Arquitetura do Docker. Fonte: Extraído de (WICKRAMASINGHE, 2021)

A arquitetura do Docker, como destacado na Figura 6, segue um modelo cliente-servidor, composto por três componentes principais: o Docker Client, o Docker Host e o Docker Registry. O Docker Client é a interface primária pela qual os usuários interagem com o Docker por meio de comandos *CLI*. O Docker Host é onde reside o Docker Daemon (dockerd), responsável por construir, executar e gerenciar os contêineres. O Docker Registry funciona como repositório para armazenar e distribuir imagens Docker, sendo o Docker Hub o *registry* público oficial (DOCKER, 2025). Esta arquitetura modular facilita a escalabilidade e a manutenção do sistema, permitindo que cada componente evolua independentemente (BERNSTEIN, 2014).

Os contêineres Docker são construídos a partir de imagens, que são *templates* contendo o sistema de arquivos e configurações necessárias para executar uma aplicação. Cada imagem é composta por camadas empilhadas seguindo o princípio de Union File System, o que permite o compartilhamento eficiente de recursos e reduz o consumo de espaço em disco. O Dockerfile é um arquivo de texto que contém instruções para construir uma imagem, definindo o ambiente de execução, dependências, variáveis de ambiente e comandos a serem executados. O dockerd é o componente central que gerencia contêineres, imagens, redes e volumes, implementando a API REST que permite a comunicação com o Docker Client.

O ecossistema Docker inclui ainda ferramentas complementares como Docker Compose para orquestração de múltiplos contêineres, Docker Swarm para *clustering* e orquestração em escala, e Docker Volumes para persistência de dados. O *networking* do Docker permite a comunicação entre contêineres por meio de redes virtuais isoladas, oferecendo diferentes *drivers* de rede para diversos casos de uso. Os Docker Volumes resolvem o problema da persistência de dados, permitindo o armazenamento de dados fora do ciclo de vida dos contêineres. Essa combinação de ferramentas e recursos torna o Docker

uma solução completa para o desenvolvimento, implantação e gerenciamento de aplicações modernas em ambientes distribuídos (MERKEL, 2014).

3.3 Kubernetes

Kubernetes é uma plataforma de código aberto projetada para automatizar a implantação, o dimensionamento e o gerenciamento de aplicações em contêineres. Desenvolvido originalmente pela Google e atualmente mantido pela Cloud Native Computing Foundation (CNCF), o Kubernetes orquestra *clusters* de *hosts* que executam contêineres, permitindo que as organizações executem cargas de trabalho de forma eficiente e resiliente. Seu funcionamento baseia-se no conceito de declaração de estado desejado, no qual os usuários especificam como desejam que suas aplicações sejam executadas, e o Kubernetes trabalha continuamente para garantir que o estado atual do sistema corresponda a esse estado desejado, realizando automaticamente ações como balanceamento de carga, escalonamento horizontal, recuperação de falhas e atualizações progressivas (*rolling updates*) (Kubernetes, 2023b). Esta abordagem declarativa permite abstrair a complexidade da infraestrutura, oferecendo uma plataforma consistente para execução de aplicações independentemente do ambiente físico. No Kubernetes, um Pod é a unidade de implantação mais básica, encapsulando um ou mais contêineres que compartilham os recursos de *hardware* do *cluster* como armazenamento e rede, com cada Pod recebendo um endereço IP único.

A arquitetura do Kubernetes, como destacado na Figura 7, segue um modelo distribuído cliente-servidor, composto por dois planos principais: o plano de controle (*control plane*) e o plano de dados (*data plane*). O plano de controle é responsável por manter e atualizar o estado desejado do *cluster*, tomando decisões globais sobre o *cluster* e detectando e respondendo a eventos. Já o plano de dados, constituído por nós trabalhadores (*worker nodes*), executa as cargas de trabalho das aplicações. Esta separação arquitetural permite alta disponibilidade, escalabilidade e segurança, uma vez que os componentes do plano de controle podem ser replicados em múltiplas máquinas para garantir tolerância a falhas. De acordo com Burns, um dos criadores do Kubernetes, esta arquitetura foi projetada para permitir separação clara de responsabilidades, facilitando tanto a manutenção quanto o escalonamento do sistema como um todo, além de possibilitar diferentes implementações para cada componente desde que respeitem as interfaces definidas (BURNS et al., 2016b).

O plano de controle do Kubernetes compreende vários componentes essenciais que trabalham juntos para manter o estado desejado do *cluster*. O API Server atua como *front-end* para o plano de controle, expondo a API do Kubernetes por meio da qual todos os outros componentes e usuários interagem com o *cluster*. O etcd é um armazenamento

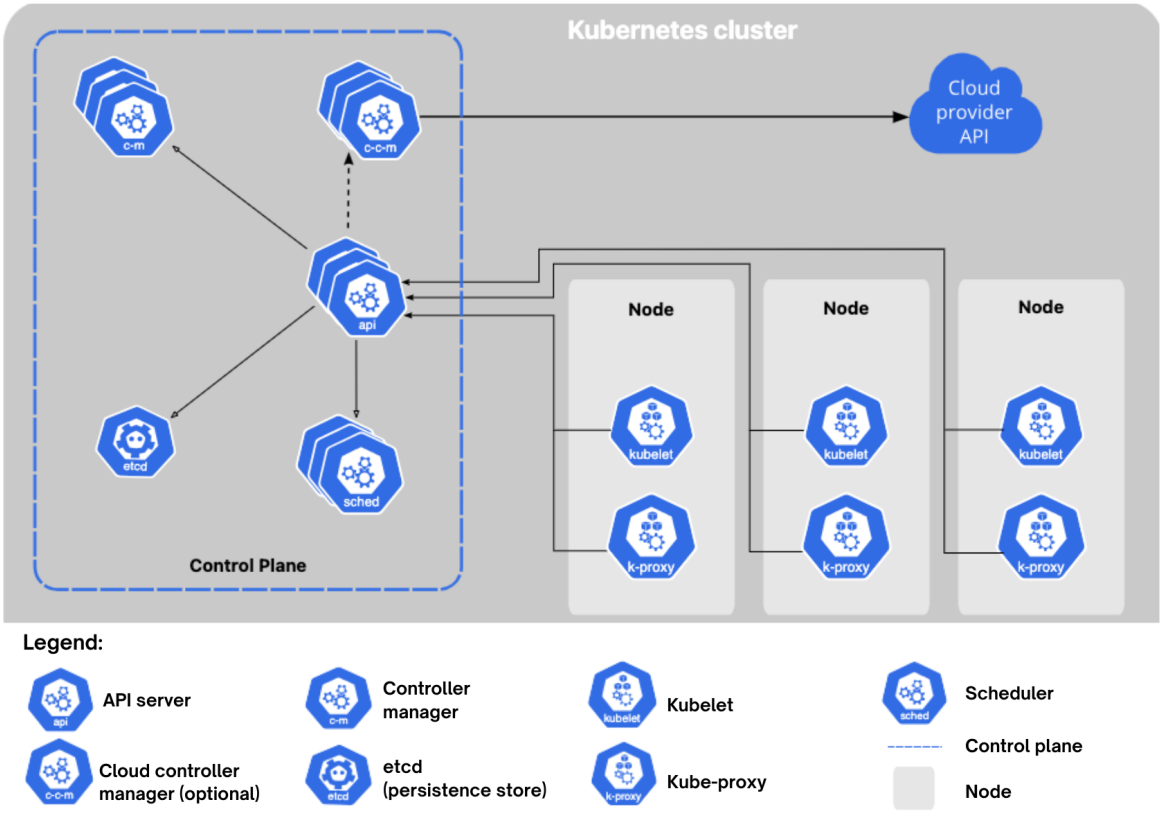

Figura 7 – Arquitetura do Kubernetes. Fonte: Extraído de (Kubernetes, 2023a).

distribuído de chave-valor que armazena todos os dados do *cluster* de forma consistente e altamente disponível, servindo como a "fonte da verdade"para todo o sistema. O *Scheduler* monitora *pods* recém-criados sem nós atribuídos e seleciona um nó para executá-los com base em requisitos de recursos, políticas, afinidade, anti-afinidade e outras restrições. O Controller Manager executa processos controladores em segundo plano, como o controlador de nós (monitorando o estado dos nós), o controlador de replicação (garantindo o número correto de *pods* para cada objeto ReplicaSet) e controladores de serviços e *endpoints* (criando recursos para balanceamento de carga e descoberta de serviços). O Cloud Controller Manager permite que o código específico de provedores de nuvem evolua independentemente do código principal do Kubernetes, incorporando lógicas específicas para diferentes ambientes de nuvem (Kubernetes, 2023b).

No plano de dados, cada nó trabalhador executa componentes necessários para hospedar as cargas de trabalho da aplicação. O *kubelet* é um agente que executa em cada nó, responsável por garantir que os contêineres descritos nos PodSpecs estejam em execução e saudáveis, comunicando-se com o API Server para receber instruções e reportar o estado do nó e seus contêineres. O *kube-proxy* mantém regras de rede em cada nó, permitindo a comunicação de rede com os *pods* a partir de sessões de rede dentro ou fora do *cluster*, implementando parte do conceito de *Service* do Kubernetes. O Container

Runtime é o *software* responsável por executar os contêineres, como Docker ou containerd, fornecendo ambiente isolado para as aplicações. Conforme destacado por Medel e outros (MEDEL et al., 2018), esta arquitetura modular do plano de dados permite que o Kubernetes suporte diferentes ambientes de execução de contêineres e tecnologias de rede, proporcionando flexibilidade para adaptação a diferentes requisitos operacionais e de infraestrutura, enquanto mantém um modelo de gerenciamento unificado.

3.4 Istio

O Istio é uma ferramenta de malha de serviço (*service mesh*) de código aberto que oferece uma solução para gerenciar serviços distribuídos em ambientes de microsserviços. Seu funcionamento baseia-se na injeção de *proxies sidecar* (Envoy) em cada *pod* de serviço, permitindo interceptar toda comunicação entre serviços sem modificação do código das aplicações. O Istio gerencia o tráfego de rede, implementa políticas e coleta métricas de telemetria, possibilitando a comunicação segura e confiável entre serviços através de recursos como balanceamento de carga, descoberta de serviços, monitoramento e controle de falhas. Como destacado por Zhang e outros (ZHANG et al., 2021), o Istio fornece capacidades críticas para implementar microsserviços de maneira consistente, incluindo segurança, observabilidade e gerenciamento de tráfego.

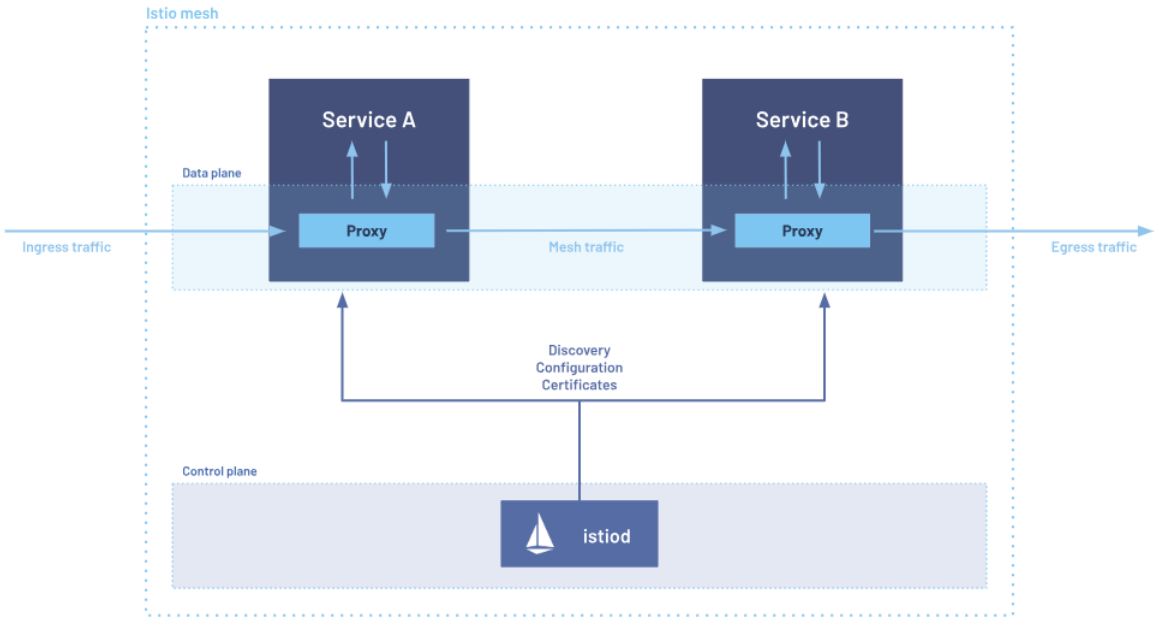

Figura 8 – Arquitetura do Istio. Fonte: Extraído de (Istio Documentation, 2024b).

A arquitetura do Istio, como destacado na Figura 8, é dividida em dois planos principais: o plano de dados e o plano de controle. O plano de dados consiste nos *proxies* Envoy implantados como *sidecars*, que interceptam todas as requisições de rede entre

microsserviços e aplicam as políticas definidas. O plano de controle, por sua vez, é responsável por configurar os *proxies* e gerenciar as políticas de serviço. Esta separação de responsabilidades permite que o Istio forneça alta disponibilidade enquanto simplifica a gestão operacional. De acordo com a documentação oficial, o Istio simplifica a configuração e o gerenciamento das funções de rede complexas através de abstrações que são independentes da implementação da plataforma subjacente (Istio Documentation, 2024a).

Os componentes principais do Istio incluem o Pilot, Citadel e Galley (que compõem o Istiod), além do *proxy* Envoy. O Pilot atua como o componente central do plano de controle, sendo responsável pela configuração dos *proxies* Envoy e pela propagação de informações de descoberta de serviços. O Citadel gerencia a segurança no Istio, lidando com a autenticação, autorização e criptografia através da geração e distribuição de certificados TLS. O Galley valida a configuração fornecida pelo usuário e a distribui para os outros componentes do plano de controle. O Envoy, como *proxy* de serviço, implementa recursos como balanceamento de carga, *circuit breaking*, *health checks* e comunicação entre serviços, além de coleta de métricas para telemetria.

A implementação do Istio oferece benefícios significativos para o gerenciamento de microsserviços, incluindo maior segurança, observabilidade aprimorada e estratégias avançadas de gerenciamento de tráfego. O controle de tráfego permite recursos como *canary releases* (implantações nas quais a porcentagem de tráfego que tem acesso à nova versão aumenta gradualmente de acordo com a configuração desejada) e testes A/B, enquanto os mecanismos de resiliência protegem a aplicação contra falhas em cascata. A telemetria detalhada proporciona *insights* sobre o comportamento dos serviços e a segurança é reforçada por meio de autenticação *mTLS* (TLS mútuo) entre serviços. Segundo Sharma e Sarangdevot (SHARMA; SARANGDEVOT, 2022), a implementação do Istio em ambientes de produção melhora significativamente a confiabilidade, segurança e visibilidade da infraestrutura de microsserviços, resultando em operações mais estáveis e maior produtividade das equipes de desenvolvimento.

3.5 Helm

O Helm é um gerenciador de pacotes para Kubernetes que simplifica a implantação e o gerenciamento de aplicações. Permite aos usuários definir, instalar e atualizar aplicações em contêineres para Kubernetes de forma simples e reproduzível por meio de *charts*, que são pacotes de recursos Kubernetes pré-configurados. O Helm utiliza *templates* escritos em Go que são combinados com valores definidos pelo usuário para gerar os manifestos Kubernetes necessários, permitindo reutilização e padronização de configurações em diferentes ambientes.

A arquitetura do Helm evoluiu significativamente entre as versões. Na versão 3, o

Helm adota uma arquitetura simplificada cliente-única, removendo o componente servidor (Tiller) presente nas versões anteriores. O cliente Helm interage diretamente com a API do Kubernetes usando as credenciais e permissões do usuário, melhorando a segurança e eliminando vulnerabilidades anteriores. Esta mudança arquitetural também simplificou o modelo de segurança, alinhando-o melhor com o modelo de controle de acesso baseado em funções (*role based access*) do Kubernetes.

Os principais componentes do Helm, como destacado na Figura 9, incluem: o Cliente Helm, uma aplicação CLI que fornece a interface para operações como instalação, atualização e desinstalação de *charts*; os *Charts*, que são pacotes de recursos Kubernetes pré-configurados contendo todos os arquivos necessários para uma aplicação; os Repositórios, que armazenam e compartilham *charts*; e os *Releases*, que são instâncias de *charts* em execução no *cluster* Kubernetes. O sistema de versionamento e histórico de *releases* do Helm permite rastrear mudanças e facilmente reverter para versões anteriores quando necessário, tornando-o ferramenta essencial para o gerenciamento do ciclo de vida de aplicações em ambientes Kubernetes (BUTCHER; FARINA; DUFFEY, 2020).

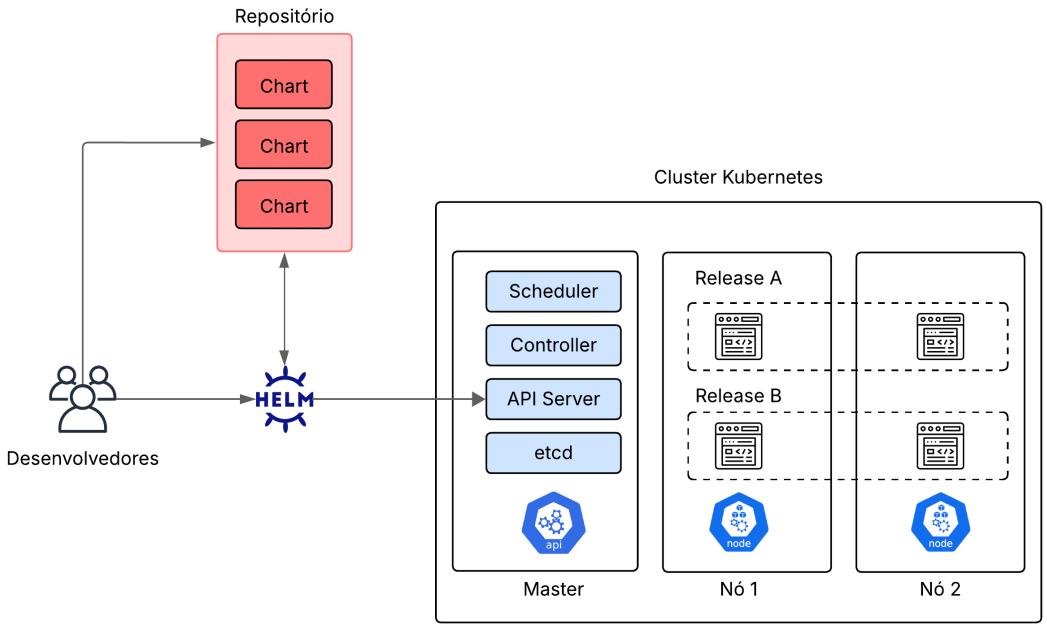

Figura 9 – Arquitetura do Helm. Fonte: Do Autor.

3.6 Terraform

O Terraform é uma ferramenta de infraestrutura como código (IaC) desenvolvida pela HashiCorp que permite aos usuários definir e provisionar infraestrutura de *data center* usando uma linguagem de configuração declarativa chamada HashiCorp Configuration Language (HCL) ou, opcionalmente, JSON. O Terraform funciona por meio de um fluxo

de trabalho que começa com a escrita do código de configuração que define os recursos desejados, seguido pela execução do comando *terraform plan* que cria um plano de execução detalhando o que será alterado para atingir o estado desejado, e finaliza com *terraform apply* que implementa as mudanças planejadas. Este modelo declarativo permite que os usuários especifiquem o estado final desejado da infraestrutura, deixando a cargo do Terraform determinar como alcançá-lo, reduzindo significativamente a complexidade operacional e minimizando erros humanos no gerenciamento de infraestrutura (HashiCorp, 2023).

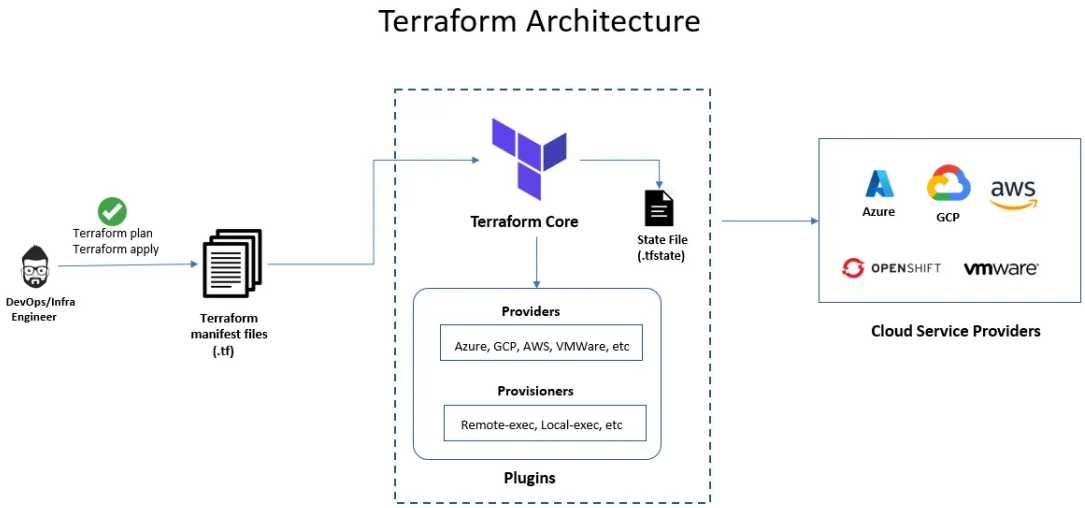

Figura 10 – Arquitetura do Terraform. Fonte: Extraído de (RAJ, 2023).

A arquitetura do Terraform, como destacado na Figura 10, é composta por dois componentes principais: o *Core* e os *Providers*. O *Core* é responsável por interpretar as configurações, criar o grafo de recursos (*Resource Graph*), gerenciar o estado da infraestrutura e coordenar as operações de planejamento e aplicação. O grafo de recursos é uma estrutura de dados fundamental que representa as dependências entre os diferentes recursos definidos no código, permitindo que o Terraform determine a ordem correta de criação, modificação ou exclusão desses recursos. Por outro lado, os *Providers* são *plugins* que implementam a integração com vários provedores de serviços (AWS, Azure, Google Cloud, etc.) ou ferramentas internas, abstraindo as APIs específicas de cada plataforma e fornecendo uma interface consistente para o *Core*. Esta arquitetura modular permite que o Terraform suporte uma ampla variedade de provedores de infraestrutura enquanto mantém um fluxo de trabalho unificado para o usuário (BRIKMAN, 2019).

Os principais componentes do Terraform desempenham papéis específicos no ecossistema da ferramenta. O Terraform CLI é a interface de linha de comando que permite aos usuários interagir com o Terraform, executando comandos como "*init*", "*plan*", "*apply*"e "*destroy*". O *State Manager* é responsável por manter e atualizar o arquivo de estado (*terraform.state*), que rastreia os metadados de todos os recursos gerenciados e suas con-

figurações atuais, essencial para que o Terraform saiba o que já existe e o que precisa ser alterado. O *Configuration Parser* analisa e valida os arquivos de configuração HCL ou JSON, convertendo-os em representação interna que o *Core* possa processar. Já o *Provisioner* executa *scripts* ou comandos nos recursos após sua criação, permitindo configurações adicionais que não podem ser expressas por meio da API do provedor. Finalmente, os *Providers*, como mencionado anteriormente, são responsáveis pela comunicação direta com as APIs das plataformas de infraestrutura, criando, lendo, atualizando e excluindo recursos conforme instruído pelo *Core* (ARUNDEL; DOMINGUS, 2022).

3.7 GitHub Actions

O GitHub Actions é uma plataforma de CI/CD integrada ao GitHub que permite automatizar fluxos de trabalho de desenvolvimento de *software*. O funcionamento do GitHub Actions baseia-se em eventos que ocorrem no repositório, como *push*, *pull request* ou programação cronológica, que acionam *workflows* definidos em arquivos YAML localizados no diretório *.github/workflows* dentro do repositório. Quando um evento é disparado, o GitHub Actions executa o *workflow* correspondente, que consiste em um ou mais *jobs* compostos por *steps* individuais, podendo ser executados em paralelo ou sequencialmente de acordo com as dependências definidas (GitHub, 2023).

A arquitetura do GitHub Actions é composta por múltiplas camadas interconectadas que trabalham em conjunto para processar os *workflows*. Na camada superior está o *Event Listener*, responsável por detectar eventos no repositório e iniciar o processamento dos *workflows*. Em seguida, o *Workflow Orchestrator* gerencia a execução dos *jobs* e suas dependências, enquanto o *Runner Manager* aloca os *runners* (ambientes de execução) adequados para cada *job*. Os *runners* podem ser hospedados pelo GitHub (facilitando a vida do usuário) ou pelo próprio usuário (proporcionando maior autonomia e controle sobre os dados processados na esteira de entrega contínua), e fornecem o ambiente isolado onde as ações são executadas. Esta arquitetura distribuída permite escalabilidade e flexibilidade para diferentes necessidades de automação (COOK; SOLIS, 2022).

Os principais componentes do GitHub Actions incluem *workflows*, eventos, *jobs*, *steps*, *actions* e *runners*, cada um com funções específicas. Os *workflows* são os arquivos de configuração que definem os processos de automação. Os eventos são os gatilhos que iniciam a execução dos *workflows*. Os *jobs* são unidades de trabalho executadas em um único *runner* e compostas por múltiplos *steps* sequenciais. Os *steps* são tarefas individuais que podem executar comandos ou *actions*. As *actions* são unidades reutilizáveis de código que podem ser compartilhadas entre diferentes *workflows* e repositórios, permitindo modularização e reutilização de funcionalidades. Por fim, os *runners* são os servidores que executam os *jobs* definidos nos *workflows*, responsáveis por fornecer o ambiente de

execução isolado para cada *job* (GitHub, 2023).

3.8 Playwright

O Playwright é um *framework* de automação de testes ponta-a-ponta (*end-to-end*) desenvolvido pela Microsoft e lançado oficialmente em janeiro de 2020, projetado especificamente para atender às demandas de aplicações *web* modernas (Microsoft, 2025). O *framework* oferece suporte nativo aos principais motores de renderização, incluindo Chromium, Firefox e WebKit, por meio de uma API unificada que permite a execução de testes multiplataforma em sistemas Windows, Linux e macOS. A arquitetura do Playwright utiliza comunicação WebSocket para estabelecer conexões bidirecionais eficientes entre o cliente e o servidor, proporcionando baixa latência e comunicação em tempo real. Diferentemente de outras ferramentas de automação que dependem de múltiplas requisições HTTP, o Playwright mantém uma conexão WebSocket persistente que permanece ativa até a conclusão de todos os testes, reduzindo pontos de falha durante a execução e proporcionando solução estável e rápida. Esta abordagem permite que o *framework* supere limitações comuns de ferramentas de automação baseadas em processos internos, oferecendo recursos avançados como espera automática, assertivas específicas para *web* e capacidades de rastreamento e depuração integradas.

A arquitetura fundamental do Playwright é baseada em três componentes principais que trabalham em conjunto para proporcionar isolamento e eficiência na execução de testes. Segundo a documentação oficial, o conceito de "Contexto de Navegador"(Microsoft, 2025) constitui o núcleo da estratégia de isolamento do *framework*, funcionando como ambientes limpos e isolados equivalentes a perfis de navegador incógnito, sendo rápidos e econômicos de criar, mesmo quando executados em uma única instância de navegador. Cada teste executa em seu próprio Contexto de Navegador, garantindo que possua *local storage*, *session storage*, *cookies* e configurações completamente isolados, eliminando interferências entre testes e facilitando a depuração. Esta arquitetura possibilita a simulação de cenários complexos envolvendo múltiplas abas, múltiplas origens e múltiplos usuários dentro de um único teste, além de permitir a reutilização de estados de autenticação entre testes mantendo o isolamento total. O servidor Playwright, alimentado por Node.js, atua como intermediário traduzindo comandos do cliente em instruções compreensíveis pelo navegador, utilizando protocolos nativos como CDP (*Chrome DevTools Protocol*) para Chromium e protocolos específicos para Firefox e WebKit, garantindo controle direto e confiável sobre os navegadores durante a execução dos testes automatizados.

3.9 Trabalhos relacionados

Esta dissertação (BRAGA, 2015) apresenta um panorama abrangente sobre o uso de práticas DevOps nas indústrias de *software*, abordando um movimento relativamente novo que busca estreitar a colaboração entre equipes de desenvolvimento e operação. O trabalho teve como objetivo principal realizar um mapeamento sistemático da literatura e um *survey* para identificar as principais áreas de concentração dos estudos sobre DevOps, os principais autores da área, e especialmente as práticas e técnicas mais utilizadas pelas organizações que adotam essa abordagem. A pesquisa se justifica pela necessidade de estruturar e catalogar conhecimentos sobre DevOps, visto que o movimento nasceu na indústria e ainda carece de padronização acadêmica e práticas prescritivas bem definidas.

Os resultados do mapeamento sistemático e do *survey* com 28 organizações revelaram que as principais áreas de concentração em DevOps incluem integração contínua, entrega contínua, testes contínuos e automação da infraestrutura. Como principais práticas identificadas destacam-se: implantações através de máquinas virtuais, visibilidade do *pipeline* de implantação, processos robustos de *rollback*, além de técnicas como *canary release*, *toggled features* e *blue-green deployments*. A pesquisa contribui para o campo ao fornecer um guia estruturado de melhores práticas DevOps, demonstrando que organizações que adotam essas práticas conseguem entregas mais rápidas e eficientes, com maior colaboração entre equipes e melhor alinhamento entre desenvolvimento, operação e necessidades de negócio.

Este trabalho (SILVA et al., 2023) tem como proposta explorar a criação e automação de infraestrutura em nuvem para *deployment* de aplicações *web* utilizando Kubernetes e a metodologia GitOps. O objetivo central é demonstrar como a infraestrutura como código (IaC) aliada às práticas DevOps pode aproximar engenheiros de *software* das tecnologias modernas de gerenciamento de ambientes computacionais. Entre os objetivos específicos estão a implementação de um *cluster* Kubernetes em nuvem pública via Terraform e Ansible, a integração de *pipelines* CI/CD e a introdução de ferramentas como Rancher e Fleet para gestão automatizada.

Os resultados obtidos no estudo de caso confirmam que a aplicação da abordagem GitOps traz benefícios significativos, como aumento na eficiência operacional, maior rastreabilidade e confiabilidade da infraestrutura. A metodologia permitiu implantar e gerenciar ambientes de forma mais simples e segura, reduzindo falhas humanas e acelerando os ciclos de entrega de *software*. Como conclusão, o trabalho serve como guia prático para profissionais que desejam adotar GitOps na gestão de *clusters* Kubernetes, destacando boas práticas, ferramentas utilizadas e os desafios enfrentados durante a implementação.

Este estudo (AMGOTHU, 2024) explora as metodologias por trás das implantações *Canary* e *Blue-Green*, sua integração com *pipelines* de CI/CD, e os benefícios, configuração

experimental, desafios e avanços futuros dessas estratégias. Os principais objetivos foram otimizar processos de CI/CD através da implementação de estratégias de implantação progressiva, reduzir riscos associados a lançamentos de novas versões de código, e melhorar a estabilidade geral do sistema através do uso de ferramentas automatizadas como Jenkins.

O trabalho demonstrou que as implantações *Canary* permitem lançamentos incrementais de novas funcionalidades para um pequeno subconjunto de usuários, fornecendo monitoramento de desempenho em tempo real e mitigação de riscos antes de um lançamento em escala completa para o ambiente de produção, enquanto as implantações *Blue-Green* envolvem manter dois ambientes separados, azul e verde, e alternar entre eles para garantir que novas mudanças não perturbem o ambiente de produção. O estudo concluiu que, ao utilizar o Jenkins para gerenciar essas abordagens de implantação, as organizações podem aprimorar seus processos de implantação, reduzir o risco de interrupções e melhorar a estabilidade geral do sistema.

Este artigo (CHOUDHURY; NANDYALA, 2024) propõe uma exploração dos efeitos do ajuste de desempenho em questões como confiabilidade e tempo de atividade com relação às três arquiteturas atuais de implantação de microsserviços: monolítica, microsserviços e *serverless*. O estudo abrange tanto arquiteturas monolíticas quanto microsserviços, juntamente com uma investigação sobre os padrões de *design* e princípios utilizados dentro de microsserviços.

O objetivo principal é apresentar uma lista de padrões usados na arquitetura de microsserviços, a comparação entre os princípios expostos pelos especialistas na decomposição de arquiteturas de microsserviços, Martin Fowler e Sam Newman, e o precursor do Princípio de Ocultação de Informação, David Parnas, que discute modularização como mecanismo para melhorar flexibilidade e compreensão de um sistema. As conclusões expõem as vantagens e desvantagens das arquiteturas monolíticas e de microsserviços obtidas da revisão da literatura em forma resumida, que pode ajudar como referência para pesquisadores da academia e indústria. Adicionalmente, revelam as tendências das arquiteturas de microsserviços hoje, demonstrando que a adoção de microsserviços e conteineirização emergiu como uma abordagem transformadora para construir soluções de *software* escaláveis e eficientes.

Este trabalho (COSTA et al., 2023) tem como proposta apresentar e analisar métodos e estratégias para sistemas que utilizam microsserviços em uma plataforma distribuída, examinando os pontos positivos, benefícios e desvantagens dessa arquitetura. O objetivo é fornecer uma base conceitual e técnica para que o leitor compreenda a arquitetura de microsserviços e consiga implementar e implantar uma aplicação padrão utilizando microsserviços em um ambiente Kubernetes com Docker. Para isso, foi desenvolvida uma aplicação de exemplo que foi reestruturada de uma arquitetura monolítica para microsserviços, integrando um aplicativo móvel e uma aplicação *web* que consomem

três microsserviços principais.

As conclusões do estudo demonstram que a implementação de microsserviços utilizando Docker e Kubernetes proporcionou benefícios significativos como escalabilidade, disponibilidade, resiliência e eficiência operacional para a aplicação. A utilização do Docker possibilitou empacotar os microsserviços em contêineres, facilitando o processo de implantação e garantindo consistência do ambiente de execução, enquanto o Kubernetes foi fundamental para a gestão do *cluster*, permitindo escalabilidade automática baseada no consumo de CPU e recuperação automática de falhas. Embora tenham sido identificados desafios relacionados à divisão da aplicação em serviços independentes e ao gerenciamento do *cluster*, os benefícios obtidos justificam a adoção dessa arquitetura, contribuindo para maior agilidade no desenvolvimento e operação da aplicação.

4 Desenvolvimento

Neste estudo de caso, implementou-se um sistema completo de cadastro de clientes que possui as quatro funcionalidades básicas: criar, atualizar, deletar e listar um cliente com os dados nome e *email*. O sistema foi dividido em duas estruturas distintas de *backend* e *frontend* a fim de melhor organizar o código implementado. O *backend* utiliza a linguagem de programação Go (GO, 2025) e o *framework* Gin (GIN, 2025), e o *frontend* utiliza a linguagem de programação TypeScript (TYPESCRIPT, 2025) com a biblioteca React (REACT, 2025). Além do sistema, foi implementado um projeto de testes automatizados que validam o sistema como um todo, utilizando a ferramenta Playwright. Toda a gestão de infraestrutura foi realizada através de um projeto de infraestrutura como código utilizando a ferramenta Terraform, garantindo que toda a parte de criação e exclusão de recursos de infraestrutura esteja automatizada.

Também neste estudo de caso, implementaram-se dois *pipelines* de CI/CD estruturados em sete etapas distintas, cada uma desempenhando um papel fundamental no processo automatizado de desenvolvimento e implantação de *software*. Os *pipelines* foram projetados para garantir a qualidade do código através de validações progressivas, iniciando com a detecção automática de *commits* em *branches* específicas e culminando na implantação segura em ambiente de produção. Esta abordagem sequencial permite identificar e corrigir problemas em estágios iniciais do desenvolvimento, reduzindo significativamente os riscos associados às implantações e melhorando a confiabilidade do sistema como um todo.

A separação entre *frontend* e *backend* oferece benefícios significativos para o desenvolvimento de sistemas web modernos, conforme destacado por (GONG et al., 2020). Esta arquitetura permite que o *frontend* se concentre exclusivamente na exibição de páginas, lógica de interface, roteamento e recursos estáticos, enquanto o *backend* se dedica ao processamento de dados e lógica de negócios, fornecendo interfaces padronizadas para comunicação. Diferentemente da arquitetura tradicional, onde componentes *frontend* e *backend* são altamente acoplados e podem colapsar simultaneamente em caso de falhas, a separação proporciona maior tolerância a falhas - mesmo quando o sistema apresenta problemas, mensagens de erro podem ser retornadas aos usuários, mantendo a funcionalidade básica da interface. Além disso, esta abordagem aumenta significativamente a capacidade de carga do sistema, melhora a experiência do usuário através de páginas com visibilidade em tempo real, facilita a manutenção independente dos componentes e permite maior flexibilidade para atender diferentes dispositivos (PC, *tablets*, *smartphones*) sem comprometer a funcionalidade do sistema como um todo.

No contexto de CI/CD, essa arquitetura desacoplada viabiliza *pipelines* de *deployment* independentes para *frontend* e *backend*, permitindo atualizações mais frequentes e menos arriscadas. As equipes podem desenvolver, testar e implantar suas respectivas camadas de forma autônoma, reduzindo dependências entre times e acelerando o ciclo de desenvolvimento. Além disso, a possibilidade de versionamento independente facilita *rollbacks* específicos e estratégias de *deploy* como *blue-green* ou *canary releases*, aumentando a confiabilidade e agilidade dos processos de entrega de *software*.

Todos os pré requisitos para o capítulo 4 podem ser conferidos no Apêndice A e todas as implementações podem ser acessadas no repositório oficial (CRUZ, 2025) deste estudo de caso.

4.1 Arquitetura Proposta na AWS

A arquitetura implementada na AWS para o sistema de gerenciamento de clientes segue um padrão moderno de computação em nuvem, utilizando serviços gerenciados e práticas de alta disponibilidade. A solução, destacada na Figura 11, está distribuída em duas zonas de disponibilidade (us-east-1a e us-east-1b) dentro da região us-east-1, garantindo redundância e resiliência contra falhas de infraestrutura. A arquitetura emprega o Elastic Kubernetes Service (EKS) como plataforma de orquestração de contêineres, integrado com o *service mesh* Istio para gerenciamento avançado de tráfego e observabilidade através do EKS *add-on* gratuito "Solo.io Istio Distribution".

A base da infraestrutura é constituída por uma Virtual Private Cloud (VPC) com endereçamento IP 10.0.0.0/16, proporcionando um ambiente de rede isolado e seguro dentro da AWS. A VPC está organizada em quatro *subnets* estrategicamente distribuídas: duas *subnets* públicas (10.0.101.0/24 e 10.0.102.0/24) e duas *subnets* privadas (10.0.1.0/24 e 10.0.2.0/24). As *subnets* públicas hospedam componentes que necessitam de acesso direto à internet, como o Internet Gateway e NAT Gateway, enquanto as *subnets* privadas abrigam os recursos críticos da aplicação, incluindo os nós do *cluster* EKS. Esta segmentação garante que os componentes da aplicação permaneçam protegidos contra acesso direto da internet, seguindo as melhores práticas de segurança em nuvem.

O ponto de entrada principal para tráfego externo é o *Network Load Balancer* (NLB) da AWS, que opera na camada 4 do modelo OSI e oferece alta performance e baixa latência. Este balanceador é do tipo *internet-facing*, permitindo que usuários externos acessem a aplicação. O NLB é automaticamente provisionado e gerenciado pelo AWS Load Balancer Controller, um componente que executa dentro do *cluster* EKS e se integra nativamente com os recursos Kubernetes. O tráfego proveniente da internet é direcionado pelo NLB para o Istio Ingress Gateway, que atua como o *proxy* de entrada para o *service mesh*, proporcionando recursos avançados de roteamento L7, terminação

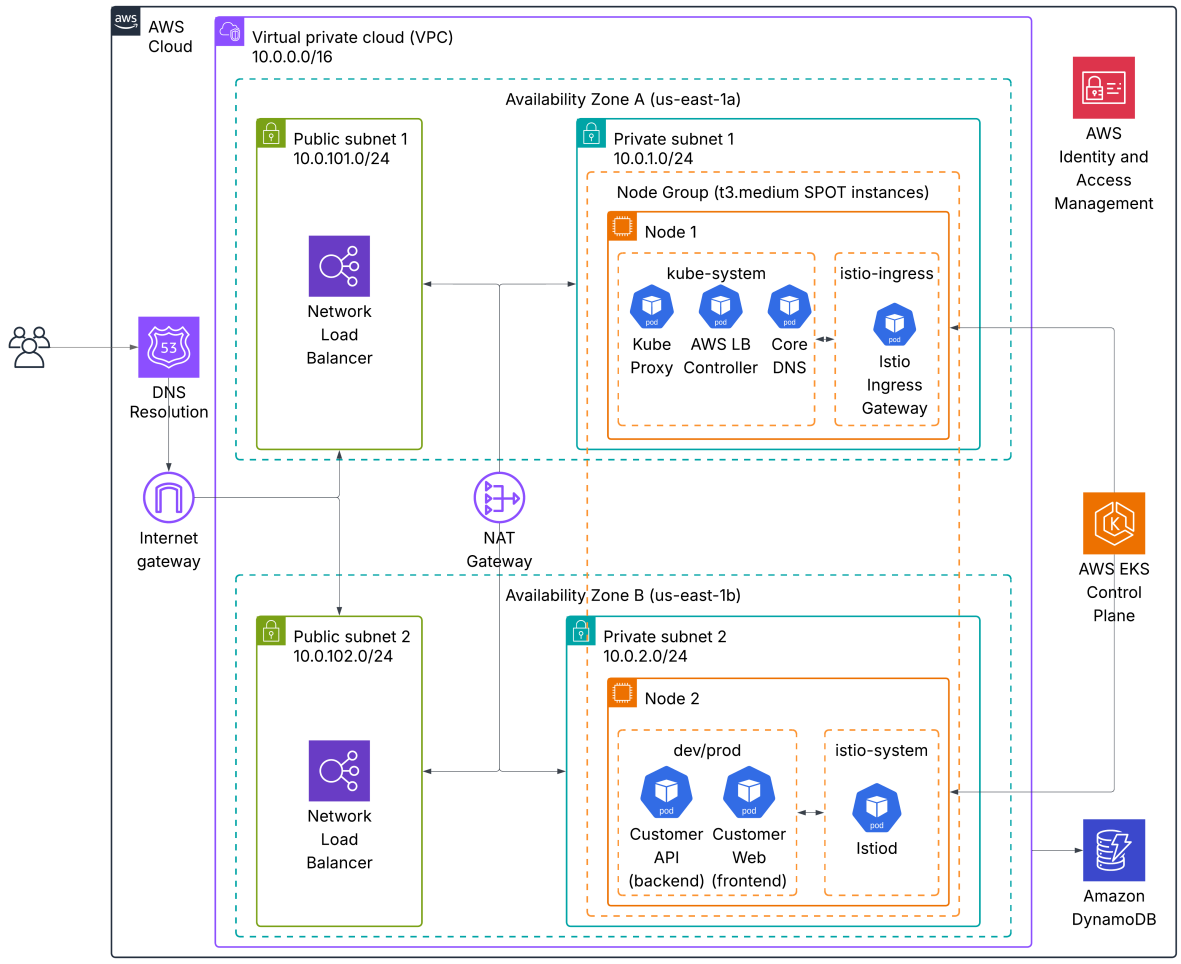

Figura 11 – Arquitetura do estudo de caso na AWS. Fonte: Do Autor.

SSL/TLS e políticas de segurança.

O fluxo de dados segue um caminho bem definido desde a requisição externa até o processamento pela aplicação. Inicialmente, as requisições HTTP/HTTPS dos usuários chegam ao *Network Load Balancer* através da internet. O NLB então encaminha o tráfego para o Istio Ingress Gateway, que está executando em *pods* dentro do *namespace* istio-ingress no *cluster* EKS. O Istio Gateway utiliza VirtualServices para implementar regras de roteamento inteligentes: requisições para o *path* /api/* são direcionadas para os *pods* da API de clientes, enquanto todas as outras requisições (/*) são encaminhadas para os *pods* do *frontend*. Esta separação permite o desenvolvimento e *deploy* independente dos componentes do *frontend* e do *backend*.

O *cluster* Amazon EKS serve como a plataforma central de orquestração, gerenciando automaticamente o plano de controle do Kubernetes e garantindo alta disponibilidade dos componentes mestres. Os nós de trabalho são organizados em grupos de nós gerenciados (EKS Managed Node Groups), utilizando instâncias EC2 do tipo t3.medium configuradas como instâncias SPOT para otimização de custos. O *cluster* está configurado

para escalar automaticamente entre 1 e 4 nós conforme a demanda, distribuídos entre as duas zonas de disponibilidade para garantir resiliência. Os nós hospedam os *pods* das aplicações, bem como os componentes do *service mesh* Istio, incluindo os *sidecars* Envoy que interceptam e gerenciam toda a comunicação entre serviços.

O Istio *service mesh* no modo Ambient adiciona uma camada sofisticada de gestão de tráfego, segurança e observabilidade à arquitetura sem a necessidade de *sidecars*. O componente principal continua sendo o Istiod, que atua como o plano de controle do *service mesh* e reside no *namespace* istio-system, distribuindo configurações, certificados TLS e políticas de segurança. No modo Ambient, a interceptação do tráfego é realizada por dois componentes principais: o ztunnel (*zero-trust tunnel*) que opera como DaemonSet em cada nó para fornecer conectividade segura L4 e criptografia mTLS transparente, e os *waypoint proxies* que são implantados sob demanda para funcionalidades L7 avançadas como *circuit breakers*, políticas de *retry* e balanceamento de carga sofisticado. Esta arquitetura elimina a sobrecarga de recursos dos *sidecars* enquanto mantém observabilidade granular do tráfego e implementação de políticas de segurança *zero-trust* entre os microsserviços, oferecendo uma abordagem mais eficiente e menos intrusiva para gerenciamento de *service mesh*.

O *namespace* kube-system abriga componentes essenciais da infraestrutura Kubernetes, incluindo o AWS Load Balancer Controller, que é fundamental para a integração entre os recursos Kubernetes e os serviços da AWS de balanceamento de carga. Este *controller* monitora recursos do tipo Service e Ingress no *cluster*, provisionando automaticamente *Application Load Balancers* (ALB) ou *Network Load Balancers* (NLB) conforme necessário. A comunicação entre os componentes é facilitada pelo CoreDNS, que resolve nomes de serviços internos, e pelo kube-proxy, que gerencia as regras de rede para distribuição de tráfego entre *pods*.

O Amazon DynamoDB serve como a solução de banco de dados NoSQL gerenciado, operando no modelo *pay-per-request* para otimização de custos baseada no uso real. A tabela "Customers"utiliza "id"como chave de partição (*hash key*), garantindo distribuição eficiente dos dados através das partições do DynamoDB. A escolha do DynamoDB alinha-se com os requisitos de escalabilidade da aplicação, oferecendo latência de milissegundos, *backup* automático e replicação multi-região quando necessário. A comunicação entre os *pods* da aplicação e o DynamoDB é realizada através das APIs REST da AWS, utilizando credenciais IAM gerenciadas através do IAM Roles for Service Accounts (IRSA).

A arquitetura implementa múltiplas camadas de segurança seguindo o princípio de defesa em profundidade. O AWS Identity and Access Management (IAM) centraliza o controle de acesso, com *roles* específicas para diferentes componentes: EKS *Cluster* Service Role para operações do plano de controle, EKS Node Group Role para instâncias EC2 *worker*, e *roles* IRSA para *pods* que necessitam acessar serviços AWS. Os grupos

de segurança (Security Groups) atuam como *firewalls* virtuais, controlando o tráfego de rede a nível de instância e permitindo apenas as portas necessárias para operação do Istio e Kubernetes. A segmentação em *subnets* privadas garante que os recursos críticos não sejam diretamente acessíveis da internet.

A distribuição dos recursos entre duas zonas de disponibilidade garante continuidade operacional mesmo na falha completa de uma zona. O EKS automaticamente distribui os *pods* entre os nós disponíveis, e o Istio implementa verificações de saúde e *circuit breakers* para detectar e isolar componentes com falha. O *Network Load Balancer* monitora continuamente a saúde dos alvos e remove automaticamente nós não responsivos do *pool* de balanceamento. Esta arquitetura multi-zona, combinada com a natureza sem estado dos *pods* de aplicação e a alta disponibilidade nativa do DynamoDB, proporciona um sistema resiliente.

O acesso externo é facilitado através de um Internet Gateway anexado às *subnets* públicas, permitindo comunicação bidirecional com a internet para recursos que necessitam dessa conectividade. Para recursos em *subnets* privadas que requerem acesso de saída à internet (como atualizações de sistema ou *download* de imagens de contêiner), um NAT Gateway localizado na *subnet* pública da AZ-2 fornece conectividade unidirecional segura. A resolução de DNS externa é gerenciada através do serviço Route 53 da AWS, enquanto internamente o CoreDNS lida com a resolução de nomes dos serviços Kubernetes.

A arquitetura suporta dois ambientes distintos com contas AWS separadas: ambiente de desenvolvimento (Account ID: 760347630853) e ambiente de produção (Account ID: 131204617414). Esta separação garante isolamento completo entre os ambientes, permitindo testes seguros de novas funcionalidades no ambiente de desenvolvimento sem risco de impacto na produção. Ambos os ambientes seguem o mesmo padrão arquitetural, com possíveis variações nos tamanhos de instância, configurações de *auto-scaling* e políticas de *backup* conforme os requisitos específicos de cada ambiente.

4.2 Implementação do Ambiente Base

A implementação do ambiente base consistiu na criação de uma infraestrutura completa na AWS, utilizando as melhores práticas de DevOps e arquiteturas *cloud-native*. O ambiente foi projetado para suportar aplicações containerizadas com alta disponibilidade, observabilidade e escalabilidade automática, estabelecendo uma base sólida para o *deployment* e operação do sistema de gerenciamento de clientes.

4.2.1 Estrutura do projeto Terraform

A organização da infraestrutura seguiu uma abordagem modular através do Terraform, promovendo separação clara de responsabilidades e facilitando a manutenção e

evolução dos recursos. A estrutura adotada contempla arquivos especializados para diferentes aspectos da infraestrutura: o arquivo principal `main.tf` concentra as configurações de providers e do estado remoto do Terraform, enquanto `variables.tf` e `outputs.tf` gerenciam respectivamente as variáveis de entrada e a exposição de valores para outros módulos. Todos os arquivos relacionados ao Terraform estão localizados na pasta `infrastructure` do repositório.

A camada de rede é definida em `vpc.tf`, estabelecendo a fundação de conectividade através da configuração de VPC e *subnets*. A orquestração de contêineres é tratada em `eks.tf`, abrangendo tanto o *cluster* Kubernetes quanto os grupos de nós associados, assim como as configurações do grupo de nós, destacado na Figura 12. A persistência de dados é configurada em `dynamodb.tf`, implementando a solução NoSQL escolhida para o projeto. O controle de acesso é centralizado em `iam.tf`, definindo *roles* e políticas IAM necessárias para operação segura dos recursos. Por fim, a integração com *service mesh* é realizada através de `istio-gateway.tf`, configurando os componentes de *ingress* e roteamento inteligente de tráfego. A configuração do Istio será abordada em uma seção futura.

```
module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "~> 20.11"

  cluster_name                   = "${local.name}-cluster"
  cluster_version                = var.kubernetes_version
  cluster_endpoint_public_access = true

  # Give the Terraform identity admin access to the cluster
  # which will allow resources to be deployed into the cluster
  enable_cluster_creator_admin_permissions = true

  cluster_addons = {
    coredns    = {}
    kube-proxy = {}
    vpc-cni    = {}
  }

  vpc_id     = module.vpc.vpc_id
  subnet_ids = module.vpc.private_subnets

  eks_managed_node_groups = {
    main = {
      name = "${local.name}-ng"

      instance_types = ["t3.medium"]

      min_size     = 1
      max_size     = 4
      desired_size = 2
      capacity_type = "SPOT"
    }
  }

  tags = local.tags
}
```

Figura 12 – Configuração do módulo EKS no arquivo eks.tf. Fonte: Do Autor

O gerenciamento de estado do Terraform foi implementado utilizando *backend* remoto no Amazon S3, garantindo consistência e permitindo colaboração entre múltiplos desenvolvedores. Esta configuração utiliza arquivos `.hcl` específicos por ambiente, pro-

porcionando flexibilidade na definição de diferentes *buckets* e configurações de *backend*. O mecanismo de bloqueio de estado foi implementado através do DynamoDB, prevenindo conflitos durante execuções concorrentes e garantindo integridade do estado da infraestrutura.

A estratégia multi-ambiente foi implementada através de arquivos de variáveis especializados, permitindo customização específica para cada contexto de *deployment*. Os arquivos `terraform.dev.tfvars` e `terraform.prod.tfvars` definem configurações adaptadas às necessidades de desenvolvimento e produção respectivamente, enquanto `backend-dev.hcl` e `backend-prod.hcl` estabelecem as configurações de *backend* correspondentes. Esta abordagem facilita a gestão de diferenças entre ambientes, como tamanhos de instância, políticas de *backup* e configurações de rede, mantendo consistência arquitetural.

4.2.2 Configuração do *Cluster* Kubernetes

O *cluster* Kubernetes foi implementado utilizando o Elastic Kubernetes Service (EKS), que oferece um plano de controle gerenciado pela AWS. A configuração foi realizada através do Terraform, garantindo reprodutibilidade e versionamento da infraestrutura.

4.2.2.1 Configuração do EKS via Terraform

O *cluster* EKS foi configurado no arquivo `infrastructure/eks.tf` utilizando o módulo oficial `terraform-aws-modules/eks/aws` versão 20.11. As principais configurações incluem:

- **Versão do Kubernetes**: 1.32 (configurável via variável)
- **Nome do *Cluster***: Formato `tcc-ufu-{ambiente}-cluster`
- **Acesso público**: Habilitado para facilitar desenvolvimento
- ***Add-ons* básicos**: CoreDNS, kube-proxy e vpc-cni
- **Permissões administrativas**: Configuradas para o criador do *cluster*

4.2.2.2 Configuração de Grupos de Nós

O grupo de nós foi configurado com as seguintes especificações:

- **Tipo de instância**: t3.medium (configurável)
- ***Scaling***: Mínimo 1, máximo 4, desejado 2

- **Tipo de capacidade**: SPOT para redução de custos
- ***subnets***: Implementadas em *subnets* privadas para segurança

4.2.2.3 Configuração de Rede e Grupos de Segurança

A configuração de rede foi estabelecida com regras específicas para integração com Istio, incluindo a abertura das portas 15017 e 15012, responsáveis respectivamente pela comunicação com o *webhook* do Istio *sidecar injector* e pela comunicação entre a API do *cluster* e os nós. A VPC foi configurada com o bloco CIDR 10.0.0.0/16, proporcionando um amplo espaço de endereçamento IP para acomodar o crescimento futuro da infraestrutura. Dentro desta VPC, foram criadas duas *subnets* públicas (10.0.101.0/24 e 10.0.102.0/24) para recursos que necessitam acesso direto à internet, e duas *subnets* privadas (10.0.1.0/24 e 10.0.2.0/24) para os nós de trabalho do *cluster*, garantindo isolamento e segurança adequados para os componentes da aplicação.

4.2.2.4 AWS Load Balancer Controller

O AWS Load Balancer Controller foi implementado através do gerenciador de pacotes Helm na versão 1.8.1, sendo instalado no *namespace* kube-system para gerenciamento automático de *load balancers*. O *controller* utiliza uma *service account* configurada com IRSA (IAM Roles for Service Accounts), permitindo autenticação segura e transparente com os serviços AWS sem necessidade de armazenamento de credenciais nos *pods*. A integração foi estabelecida para operar especificamente com a VPC criada e a região AWS configurada, garantindo que todos os *load balancers* sejam provisionados corretamente dentro da infraestrutura definida.

4.2.2.5 Configuração do kubectl para acesso ao *cluster*

O acesso ao *cluster* EKS foi configurado através do kubectl, cliente de linha de comando oficial para interação com APIs do Kubernetes. A configuração foi completamente automatizada através do Terraform, implementando um sistema de autenticação AWS baseado no comando `aws eks get-token` para obtenção de *tokens* temporários, eliminando a necessidade de gerenciamento manual de credenciais. O processo inclui a decodificação automática do certificado da autoridade certificadora do *cluster* e a configuração automática do *endpoint* HTTPS do *cluster* EKS. A implementação utiliza o *plugin* exec com a versão de API client.authentication.k8s.io/v1beta1 para garantir compatibilidade e permitir renovação automática de *tokens*, assegurando acesso contínuo e seguro ao *cluster*.

A configuração foi implementada nos *providers* Terraform (kubernetes, helm, kubectl) garantindo que todas as operações de *deploy* utilizem as credenciais corretas. O

processo de autenticação é transparente e utiliza as credenciais AWS configuradas no ambiente de execução. Com a configuração do kubectl concluída, é possivel operar o *cluster* Kubernetes remotamente de diversas formas, como por exemplo listar todos os *pods* que estão no *namespace* kube-system, como destacado na Figura 13.

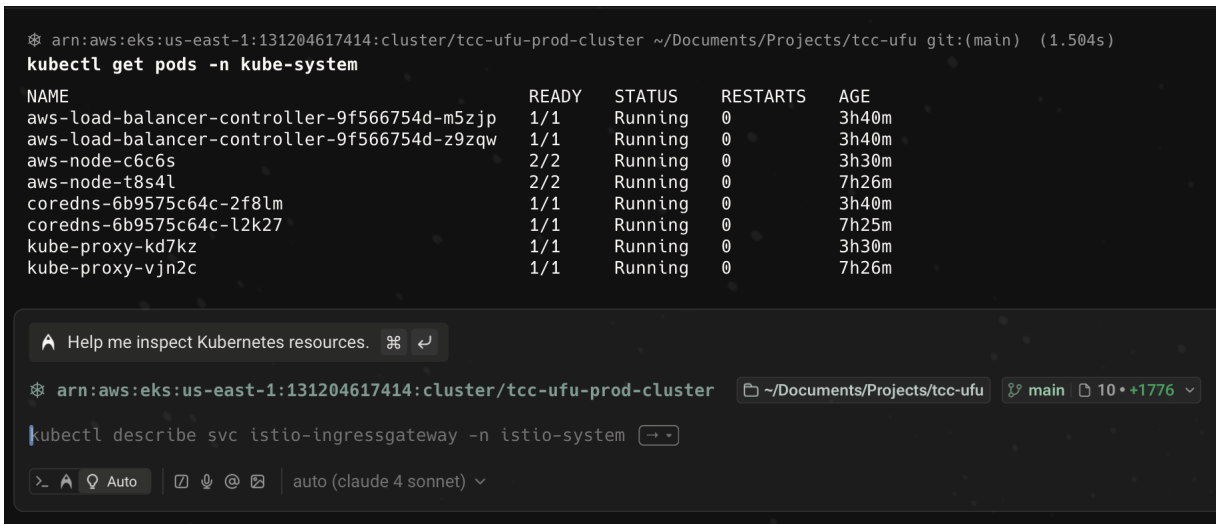

Figura 13 – *Pods* listados no *namespace* kube-system no *cluster* EKS de produção. Fonte: Do Autor

4.2.3 Instalação e Configuração do Istio

A instalação do Istio utilizou o *add-on* oficial da AWS através do Solo.io Istio Distribution, disponível no AWS Marketplace (MARKETPLACE, 2025). Esta abordagem garante suporte oficial e integração otimizada com o EKS e traz mais simplicidade ao processo de instalação do Istio.

4.2.3.1 Instalação via EKS *Add-on*

O Istio foi instalado como um *add-on* do EKS utilizando o recurso `aws_eks_addon` definido no arquivo `infrastructure/eks.tf`. A instalação utilizou a distribuição solo-io_istio-distro na versão v1.26.0-eksbuild.1, configurada com política de conflitos OVERWRITE para facilitar atualizações futuras. As configurações incluem o meshConfig com accessLogFile direcionado para /dev/stdout, permitindo coleta centralizada de *logs* através dos sistemas de registro do Kubernetes. O componente *pilot* foi configurado com *auto-scaling* desabilitado e limitado a uma única réplica, adequado para o ambiente de desenvolvimento e reduzindo o consumo de recursos computacionais.

4.2.3.2 Configuração do *Service Mesh*

O *service mesh* foi configurado seguindo as melhores práticas para ambiente de desenvolvimento, utilizando cluster.local como *mesh* ID e domínio de confiança, estabele-

cendo uma identidade consistente para todos os componentes do *mesh*. Os *logs* de acesso foram habilitados para proporcionar observabilidade completa do tráfego entre serviços, facilitando *debugging* e monitoramento. O ID do *cluster* foi mantido como Kubernetes (valor padrão), garantindo compatibilidade com as convenções padrão do Istio e simplificando futuras integrações com ferramentas de observabilidade.

4.2.3.3 Istio Ingress Gateway

Foi implementado um gateway customizado no arquivo `istio-gateway.tf` com as seguintes características:

- ***Namespace***: istio-ingress (criado especificamente)
- ***Deployment***: Baseado em istio/proxyv2:1.26.0
- **Tipo de Serviço**: LoadBalancer com *Network Load Balancer* (NLB)
- **Portas**: 80 (HTTP), 443 (HTTPS), 15021 (status)
- **Recursos**: 100m CPU / 128Mi RAM (request), 2000m CPU / 1024Mi RAM (limit)
- **Verificações de Saúde**: *Liveness* e *readiness probes* configuradas

4.2.3.4 *Service Account* e *Role-Based Access Control* (RBAC)

No contexto de Kubernetes e Istio, as *service accounts* representam identidades para processos que executam em *pods*, funcionando como contas de usuário para aplicações e serviços dentro do *cluster*. Essas contas são automaticamente montadas nos *pods* e fornecem *tokens* JWT que permitem autenticação com a API do Kubernetes. O RBAC complementa as *service accounts* ao definir um sistema granular de autorização baseado em papéis, onde recursos e operações são organizados em *Roles* e *ClusterRoles*, que são posteriormente vinculados às *service accounts* através de *RoleBindings* e *ClusterRoleBindings*. No ambiente Istio, essa arquitetura de segurança é estendida através do uso de *service accounts* como base para identidades de serviço, permitindo políticas de segurança mTLS automáticas entre serviços e controle de acesso refinado através de *Authorization Policies*, criando assim uma malha de serviços de confiança zero onde cada comunicação é autenticada e autorizada com base nas identidades e permissões definidas. O *gateway* utiliza uma *service account* dedicada (`istio-ingressgateway`) com as permissões necessárias para operação no *namespace* istio-ingress.

4.2.3.5 Validação da Instalação

A validação da instalação do Istio foi realizada através de um processo abrangente de verificações para garantir o funcionamento adequado de todos os componentes.

O processo iniciou com a verificação dos *pods* através do comando `kubectl get pods -n istio-system`, confirmando que o plano de controle (istiod) estava em execução e operacional. A validação do *status* do *add-on* foi realizada via console AWS EKS, confirmando que o `solo-io_istio-distro` estava instalado com *status* "Active". O *deployment* do *gateway* customizado no *namespace* `istio-ingress` foi verificado, assegurando *status* "Ready"e funcionamento adequado. A criação automática do *Network Load Balancer* (NLB) na AWS foi confirmada, estabelecendo o ponto de entrada externo para o *cluster*. As verificações de saúde foram validadas através dos *endpoints* de saúde nos *pods*, especificamente `/healthz/ready` na porta 15021, garantindo que os componentes estavam prontos para receber tráfego. Testes de conectividade entre *pods* utilizando *proxy sidecar* foram executados para validar a funcionalidade do *service mesh*. A análise dos *logs* do istiod confirmou a ausência de erros durante a inicialização, e a verificação das regras de grupo de segurança para as portas 15017 e 15012 assegurou a conectividade adequada dos componentes de rede.

A validação também incluiu testes funcionais básicos como injeção de *sidecar* em *pods* de teste e verificação da coleta de métricas de telemetria. Todos os componentes foram verificados através de verificações de saúde automatizadas implementadas nos *deployments*.

4.2.4 Configuração Base da Aplicação

A aplicação foi estruturada seguindo práticas de containerização e orquestração, com *deploy* realizado através de Helm *charts* utilizados pelos *pipelines* do GitHub Actions. Um *commit* na *branch* main acionou o *pipeline* pela primeira vez, realizando assim os *deploys* iniciais em desenvolvimento e posteriormente em produção. A Figura 14 demonstra o sistema de cadastro de clientes implantado no ambiente produtivo ao final da execução da pipeline.

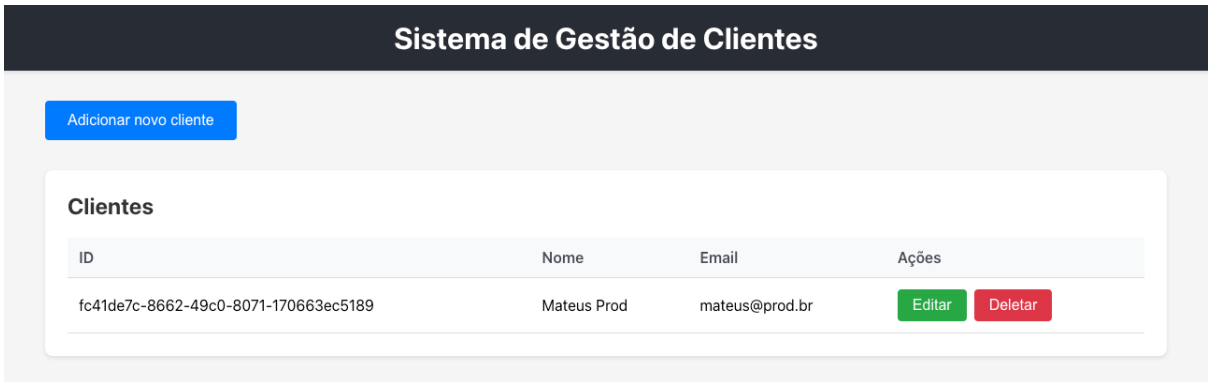

Figura 14 – Sistema de gerenciamento de clientes implantado em produção. Fonte: Do Autor.

4.2.4.1 Implantação Inicial via Helm *Charts*

A aplicação possui Helm *charts* organizados no diretório `infrastructure/charts`, estruturados em três componentes distintos que refletem a arquitetura da solução. O *chart* **customer-api** contém as definições para *deploy* da API REST desenvolvida em Go, enquanto o *chart* **customer-frontend** gerencia o *deploy* da interface web React. Adicionalmente, foi criado o *chart* **customer-management** como um *chart* genérico que permite o *deploy* conjunto de ambos os componentes, facilitando a gestão integrada da aplicação.

4.2.4.2 Configuração de *Namespaces*

A estruturação de *namespaces* foi organizada de forma a proporcionar isolamento adequado entre ambientes e componentes da infraestrutura. Os *namespaces* **development** e **production** foram criados para segregar completamente os ambientes de desenvolvimento e produção, garantindo que recursos, configurações e políticas de segurança sejam aplicados de forma independente. O *namespace* **istio-system** abriga o plano de controle do Istio, mantendo os componentes centrais do *service mesh* isolados da aplicação. O *namespace* **istio-ingress** foi dedicado exclusivamente ao *ingress gateway*, permitindo configurações específicas de rede e segurança para o componente responsável pelo tráfego externo.

4.2.4.3 Configuração da API REST

O *chart* `customer-api` foi configurado com:

- ***Deployment***: 1 réplica inicial com *auto-scaling* desabilitado
- **Serviço**: ClusterIP na porta 80, porta de destino 8080
- **ConfigMap**: Variáveis de ambiente para AWS_REGION, TABLE_NAME, PORT
- ***Service Account***: Configurada com IRSA para acesso ao DynamoDB
- **Verificações de Saúde**: *Liveness* e *readiness probes* em /health
- **Recursos**: 50m CPU / 64Mi RAM (request), 100m CPU / 128Mi RAM (limit)

4.2.4.4 Configuração da Interface Web

O *chart* `customer-frontend` foi configurado com:

- ***Deployment***: 1 réplica inicial com *auto-scaling* desabilitado
- **Serviço**: ClusterIP na porta 80, porta de destino 80

- **ConfigMap**: Configuração de API_URL relativa (/api) para comunicação com *backend*
- ***Service Account***: Sem anotações IRSA
- **Verificações de Saúde**: *Liveness* e *readiness probes* em /health
- **Recursos**: 50m CPU / 64Mi RAM (request), 100m CPU / 128Mi RAM (limit)

4.2.4.5 Integração com Istio

A integração com Istio foi estabelecida através de uma configuração abrangente de recursos que garantem exposição externa segura e roteamento inteligente de tráfego. O recurso *Gateway* foi configurado para proporcionar exposição externa da aplicação, estabelecendo o ponto de entrada controlado para tráfego externo. O *VirtualService* foi implementado para gerenciar o roteamento de tráfego HTTP e HTTPS, definindo regras de direcionamento baseado no *path* de uma requisição. A *DestinationRule* foi configurada com políticas de balanceamento de carga utilizando o algoritmo LEAST_CONN, otimizando a distribuição de carga entre as instâncias disponíveis. O *service mesh* proporciona comunicação transparente entre serviços, incluindo criptografia mútua automática (mTLS), coleta de métricas de tráfego e capacidades avançadas de observabilidade, sem necessidade de modificações no código da aplicação.

4.3 Automação vs Intervenção Manual

Ambos os *pipelines* implementados para este estudo de caso caracterizam-se por serem completamente automatizados, sem necessidade de intervenção manual em nenhuma etapa do processo de *deployment*. Desde a execução dos testes unitários e de integração até o *deployment* final em produção, todas as etapas são executadas automaticamente mediante gatilhos específicos como *push* para as *branches* main ou develop. Esta automação total é alcançada através de condicionais bem definidas que garantem que apenas código que passou por todos os testes de qualidade seja promovido para produção, eliminando a necessidade de aprovações manuais ou intervenções humanas durante o fluxo de *deployment*. O sistema utiliza saídas de *jobs* anteriores e verificações de resultado para determinar automaticamente se o processo deve prosseguir para a próxima etapa.

Uma alternativa ao modelo completamente automatizado seria a implementação de uma etapa de aprovação manual antes da promoção da imagem do ambiente de desenvolvimento para produção e subsequente *deployment*. Esta etapa poderia ser implementada utilizando o conceito de *environments* do GitHub Actions com revisores obrigatórios, onde um ou mais desenvolvedores ou engenheiros DevOps precisariam revisar e aprovar

manualmente a promoção para produção. Esta aprovação manual seria posicionada estrategicamente entre os *jobs* "e2e-tests"e "promote-image", criando um portal de qualidade humano onde aspectos como impacto nos usuários, momento do *deployment* e validações adicionais poderiam ser considerados antes de prosseguir com a publicação em produção.

Os *pipelines* completamente automatizados oferecem vantagens significativas em termos de velocidade de entrega e consistência operacional. A automação total elimina gargalos humanos, permitindo implantações frequentes e rápidas que são essenciais para práticas de entrega contínua e DevOps modernas. Além disso, reduz significativamente o risco de erros humanos durante o processo de *deployment*, uma vez que todas as etapas seguem procedimentos padronizados e testados. A automação também permite *deployments* fora do horário comercial e garante que o mesmo processo seja executado consistentemente, independentemente de quem iniciou o *deployment*. Esta abordagem promove uma cultura de desenvolvimento onde os desenvolvedores têm confiança para fazer mudanças frequentes, sabendo que o *pipeline* automatizado irá validar e implantar suas alterações de forma segura e confiável.

Por outro lado, *pipelines* com intervenção manual proporcionam maior controle sobre o momento e o contexto dos *deployments* em produção. A aprovação manual permite que partes interessadas avaliem fatores externos como carga de usuários, períodos críticos de negócio, ou dependências de outros sistemas antes de autorizar um *deployment*. Esta abordagem é particularmente valiosa em organizações com processos de conformidade rigorosos ou em sistemas críticos onde o impacto de falhas pode ser significativo. A intervenção manual também oferece uma oportunidade adicional para revisão humana de mudanças complexas ou de alto risco, permitindo que desenvolvedores avaliem o contexto completo das alterações antes da promoção para produção.

As desvantagens dos *pipelines* automatizados incluem a potencial falta de controle sobre o momento dos *deployments* e a possibilidade de *deployments* ocorrerem em momentos inoportunos, como durante picos de tráfego ou manutenções planejadas de sistemas dependentes. Embora o *pipeline* automatizado execute todos os testes programados, ele não pode avaliar contextos de negócio complexos ou fatores externos que podem afetar o sucesso do *deployment*. Adicionalmente, a automação total pode criar uma falsa sensação de segurança, onde equipes podem se tornar excessivamente dependentes dos testes automatizados sem considerar cenários extremos ou validações manuais que poderiam identificar problemas não cobertos pelos testes.

As desvantagens dos *pipelines* com intervenção manual são primariamente relacionadas à velocidade e eficiência operacional. A necessidade de aprovação manual cria gargalos que podem atrasar significativamente o tempo de mercado de funcionalidades e correções críticas. Esta abordagem também introduz dependência de disponibilidade humana, onde *deployments* podem ser atrasados devido a ausências, fusos horários diferentes,

ou simples esquecimento de aprovação. Além disso, a intervenção manual pode introduzir inconsistências no processo, onde diferentes aprovadores podem aplicar critérios distintos ou tomar decisões baseadas em informações incompletas. A longo prazo, a dependência de aprovações manuais pode desencorajar *deployments* frequentes, levando a acúmulo de mudanças e aumentando o risco e complexidade de cada *deployment* individual.

4.4 Arquitetura dos *Pipelines*

Os *pipelines* implementados seguem as melhores práticas da engenharia de *software* moderna, incorporando automação completa, testabilidade e segregação de ambientes. O fluxo contempla desde os testes unitários até a validação em produção, estabelecendo barreiras de qualidade que garantem estabilidade e confiabilidade do *software* entregue.

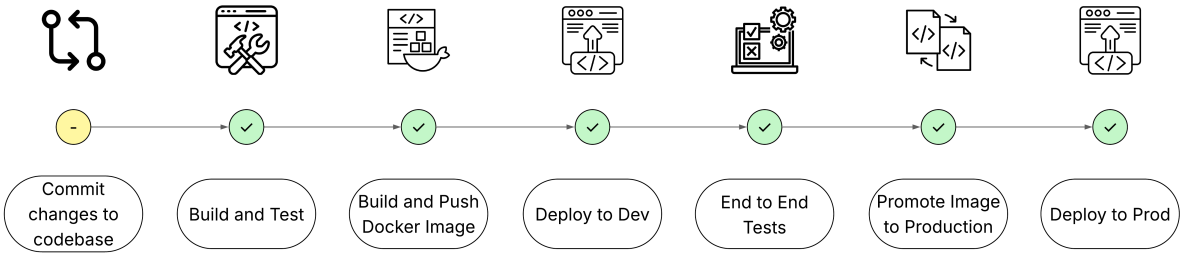

Figura 15 – Diagrama da arquitetura dos *pipelines* CI/CD. Fonte: Do Autor.

A Figura 15 ilustra os sete estágios que compõem tanto o *pipeline* de *backend* quanto o *pipeline* do *frontend*. Embora ambos sigam a mesma estrutura geral, cada um adapta-se às especificidades tecnológicas de sua aplicação.

4.4.1 *Commit changes*

A partir de um *commit* na *branch* main (usualmente ao fazer o *merge* de um *pull request*), o *pipeline* é acionado. Neste estudo de caso, separamos a execução dos *pipelines* de acordo com o caminho dos arquivos que estão sendo modificados no *pull request* ou no *commit* em questão.

4.4.2 *Build and Test*

O primeiro estágio do *pipeline* executa verificações de qualidade e testes unitários, com implementações distintas para cada tecnologia:

***Backend* (Go):** Configura o Go 1.21, instala dependências via make deps, formata o código (make fmt), executa *linting* (make lint) e roda os testes unitários com relatório de cobertura (make test-coverage). Utiliza *cache* de módulos Go baseado em go.sum.

***Frontend* (Node.js):** Configura o Node.js 18, instala dependências via make install, executa *linting* (make lint), roda testes unitários (make test) e executa o *build* (make build). Utiliza *cache* NPM baseado em package-lock.json e armazena artefatos de *build* estáticos.

Ambas as versões geram identificadores únicos baseados em data e *hash* do *commit*, além de determinar se o *deployment* deve prosseguir baseado na *branch* e tipo de evento.

4.4.3 *Build and Push Docker Image*

Este estágio executa condicionalmente após o sucesso do anterior, construindo e publicando imagens Docker:

Diferença principal: O *frontend* inclui argumentos de *build* específicos (–build-arg REACT_APP_API_URL="/api") necessários para aplicações React conhecerem os *endpoints* de API em tempo de compilação, enquanto o *backend* obtém configurações via variáveis de ambiente.

Ambos criam *tags* versionadas e "latest", publicando em repositórios ECR específicos (customer-api para *backend*, customer-frontend para *frontend*).

4.4.4 *Deploy to Dev Environment*

Realiza o *deployment* em ambiente de desenvolvimento usando Kubernetes e Helm:

***Backend*:** Configura *service account* com o endereço ARN de uma *role* do AWS IAM, tabela de clientes no DynamoDB no ambiente de desenvolvimento, e configurações específicas do Istio *Gateway* para API REST.

***Frontend*:** *Deploy* simplificado focado apenas na configuração de imagem e recursos, sem necessidade de configurações de banco ou *service accounts* específicas.

Ambos criam o *namespace* "development"com as *labels* relacionadas ao Istio em modo Ambient e no final obtêm a URL do *Gateway* para acesso externo.

4.4.5 *End-to-End Tests*

Executa os testes de integração completos usando Node.js 18 e Playwright:

***Backend*:** Verifica o estado de saúde do *backend* fazendo uma requisição para o *endpoint* "/customers"e executa make test-api-ci, focando em funcionalidades da API REST.

***Frontend*:** Verifica o estado de saúde do *frontend* fazendo uma requisição para o *endpoint* raiz "/"e executa make test-frontend-ci, testando interface de usuário e interações.

Ambos suportam o *fallback* para o *port-forward* de seus *pods* implantados no Kubernetes para o caso em que a URL do *Gateway* não esteja disponível.

4.4.6 *Promote Image to Production ECR*

Promove as imagens validadas para o ECR de produção, executando esta etapa apenas para a *branch* main após sucesso dos testes E2E. O processo é idêntico para ambas as aplicações: configura credenciais para ambas as contas AWS, baixa imagem do ECR de desenvolvimento, adiciona a *tag* novamente e envia para ECR de produção.

4.4.7 *Deploy to Production*

Executa o *deployment* em produção com configurações otimizadas:

Recursos diferenciados: O *frontend* utiliza menos recursos (50m CPU, 64Mi memória - *request*; 200m CPU, 256Mi memória - *limit*) se comparado ao *backend* (100m CPU, 128Mi memória - *request*; 500m CPU, 512Mi memória - *limit*), refletindo diferentes demandas computacionais para as duas aplicações. Todas essas configurações de recursos são feitas através dos *templates* do Helm, permitindo ter configurações diferentes entre as duas aplicações e também entre os ambientes de desenvolvimento e o produtivo.

Testes de Verificação: Os testes de verificação de produção também diferem nos *endpoints* testados: "/"para o *frontend* e "/customers"para o *backend*, mantendo consistência com os *endpoints* de verificação de saúde utilizados nos testes automatizados.

4.4.8 *Notify Deployment Status*

Consolida os resultados e gera um resumo detalhado sobre o *status* final do *deployment*. Analisa os resultados e determina o *status* de cada etapa e cria um resumo com a versão implantada, o *hash* do *commit* que acionou o *pipeline* e as URLs de cada ambiente quando estas estão disponíveis.

4.4.9 Características Unificadoras

A arquitetura demonstra maturidade através da padronização de estruturas entre *pipelines*, adaptando apenas aspectos tecnológicos específicos. Esta consistência facilita a manutenção, reduz a curva de aprendizado e garante uma aplicação uniforme das melhores práticas de DevOps em toda a *stack* da aplicação. A segregação de ambientes e promoção de imagens validadas asseguram que apenas o código que foi devidamente testado e aprovado chegue à produção.

4.5 Implementação dos *Pipelines*

A implementação dos *pipelines* CI/CD foi realizada utilizando GitHub Actions, aproveitando sua integração nativa com repositórios Git e capacidades de automação robustas. Os *workflows* são definidos em arquivos que utilizam a linguagem de serialização de dados YAML localizados no diretório `.github/workflows/`, sendo `backend-ci-cd.yml` e `frontend-ci-cd.yml` responsáveis pelos respectivos *pipelines*.

4.5.1 Configuração de *Triggers* e Variáveis de Ambiente

Ambos os *pipelines* utilizam gatilhos baseados em eventos de *push* e *pull request*, mas com seletores de caminho específicos para otimizar execução:

***Backend*:** O *workflow* é acionado por mudanças nos caminhos `backend/**` e `.github/workflows/backend-ci-cd.yml`, garantindo que apenas alterações relevantes ao *backend* disparem o *pipeline*.

***Frontend*:** Similarmente, responde a mudanças em `frontend/**` e também no arquivo `.github/workflows/frontend-ci-cd.yml`.

As variáveis de ambiente demonstram as diferenças tecnológicas fundamentais:

```
# Backend
env:
  GO_VERSION: '1.21'
  ECR_REPOSITORY: customer-api
  BACKEND_HELM_CHART_PATH: infrastructure/charts/customer-api

# Frontend
env:
  NODE_VERSION: '18'
  ECR_REPOSITORY: customer-frontend
  FRONTEND_HELM_CHART_PATH: infrastructure/charts/customer-frontend
```

4.5.2 Implementação do *Job Build e Test*

A implementação deste *job* revela as maiores diferenças entre os dois *pipelines*, refletindo as particularidades de cada *stack*.

Configuração de Ambiente - *Backend*:

```
- name: Set up Go
  uses: actions/setup-go@v5
```

```
  with:
    go-version: ${{ env.GO_VERSION }}
    cache-dependency-path: backend/go.sum

- name: Cache Go modules
  uses: actions/cache@v4
  with:
    path: |
      ~/.cache/go-build
      ~/go/pkg/mod
    key: ${{ runner.os }}-go-${{ hashFiles('backend/go.sum') }}
```

Configuração de Ambiente - *Frontend*:

```
- name: Setup Node.js
  uses: actions/setup-node@v4
  with:
    node-version: ${{ env.NODE_VERSION }}
    cache: 'npm'
    cache-dependency-path: frontend/package-lock.json
```

A sequência de comandos também difere significativamente:

***Backend*:** Executa `make deps`, `make fmt`, `make lint` e `make test-coverage`, focando em formatação automática de código Go e relatórios detalhados de cobertura.

***Frontend*:** Executa `make install`, `make lint`, `make test` e `make build`, onde a etapa de *build* é crucial para gerar os artefatos estáticos necessários para aplicações React.

4.5.3 *Build* e *Push* de Imagens Docker

A construção das imagens Docker apresenta uma diferença arquitetural importante:

***Backend* - Construção Simples:**

```
- name: Build, tag, and push image to Amazon ECR
  run: |
    docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
    docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:latest .
```

***Frontend* - Com Argumentos de *Build*:**

```
- name: Build, tag, and push image to Amazon ECR
  run: |
    docker build \
      --build-arg REACT_APP_API_URL="/api" \
      -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
    docker build \
      --build-arg REACT_APP_API_URL="/api" \
      -t $ECR_REGISTRY/$ECR_REPOSITORY:latest .
```

Esta diferença reflete uma característica fundamental das aplicações React, que necessitam conhecer URLs de API em tempo de compilação, contrastando com aplicações *backend* que obtêm configurações dinamicamente via variáveis de ambiente.

4.5.4 Configurações de *Deploy*

Os *jobs* de *deployment* revelam diferenças na complexidade de configuração. Como destacado na Figura 16, o *backend* tem mais configurações necessárias que o *frontend*, recebendo mais variáveis de ambiente como a região da AWS, o *endpoint* do DynamoDB e o nome da tabela.

```
# Helm upgrade/install to production
helm upgrade --install customer-api-prod ${{ env.BACKEND_HELM_CHART_PATH }} \
  --namespace production \
  --set image.repository=$(echo $IMAGE_URI | cut -d':' -f1) \
  --set image.tag=$VERSION \
  --set serviceAccount.annotations."eks\.amazonaws\.com/role-arn"="arn:aws:iam::131204617414:role/tcc-ufu-prod-pod-role" \
  --set replicaCount=1 \
  --set config.awsRegion=${{ env.AWS_REGION }} \
  --set config.tableName="Customers-prod" \
  --set config.dynamodbEndpoint="https://dynamodb.${{ env.AWS_REGION }}.amazonaws.com" \
  --set virtualService.gateway="customer-api-prod-gateway" \
  --set virtualService.backend.host="customer-api-prod.production.svc.cluster.local" \
  --set virtualService.frontend.host="customer-frontend-prod.production.svc.cluster.local" \
  --set resources.requests.cpu="100m" \
  --set resources.requests.memory="128Mi" \
  --set resources.limits.cpu="500m" \
  --set resources.limits.memory="512Mi" \
  --wait \
  --timeout=600s
```

Figura 16 – Trecho de código do Helm utilizado no *pipeline* do *backend* para executar o *deploy* em produção. Fonte: Do Autor.

No *pipeline* do *frontend*, podemos perceber uma configuração muito mais simplificada que não exige nenhuma variável de ambiente, como destacado na Figura 17.

Podemos notar também algumas diferenças essenciais no uso de recursos computacionais entre o *backend* e o *frontend*. O *backend* recebe mais recursos de CPU e memória RAM pela sua natureza tecnológica, e o *frontend*, pela sua simplicidade, recebe menos recursos a nível de *deployment*.

```
# Helm upgrade/install to production
helm upgrade --install customer-frontend-prod ${{ env.FRONTEND_HELM_CHART_PATH }} \
  --namespace production \
  --set image.repository=$(echo $IMAGE_URI | cut -d':' -f1) \
  --set image.tag=$VERSION \
  --set replicaCount=1 \
  --set resources.requests.cpu="50m" \
  --set resources.requests.memory="64Mi" \
  --set resources.limits.cpu="200m" \
  --set resources.limits.memory="256Mi" \
  --wait \
  --timeout=600s
```

Figura 17 – Trecho de código do Helm utilizado no *pipeline* do *frontend* para executar o *deploy* em produção. Fonte: Do Autor.

4.5.5 Implementação de Testes *End-to-End*

Esta etapa do *pipeline* do GitHub Actions executa testes automatizados para validar o funcionamento completo da aplicação em ambiente real utilizando a ferramenta Playwright. A diferenciação nos testes E2E ocorre principalmente nos *endpoints* de verificação de saúde e comandos executados:

Backend:

```
# Verificação de saúde
if curl -f "$DEV_URL/customers" > /dev/null 2>&1; then
  echo "Dev environment is ready!"
  break
fi
# Execução dos testes
make test-api-ci || TEST_RESULT=$?
```

Frontend:

```
# Verificação de saúde
if curl -f "$DEV_URL/" > /dev/null 2>&1; then
  echo "Dev environment is ready!"
  break
fi
# Execução dos testes
make test-frontend-ci || TEST_RESULT=$?
```

4.5.6 Padrões de Reutilização e Manutenibilidade

A implementação demonstra excelente reutilização de padrões entre os *pipelines*. *Jobs* como "Promote Image to Production ECR"e "Notify Deployment Status"são praticamente idênticos, diferindo apenas nas variáveis de ambiente e nomes de recursos. Esta padronização facilita:

- **Manutenção:** Correções e melhorias podem ser aplicadas de forma consistente
- ***Debugging*:** Problemas similares têm soluções conhecidas
- **Integração:** Desenvolvedores familiarizados com um *pipeline* facilmente compreendem o outro
- **Evolução:** Novas funcionalidades podem ser implementadas seguindo padrões estabelecidos

A implementação também incorpora práticas avançadas como execução condicional, gerenciamento de artefatos, execução de *jobs* em paralelo e tratamento abrangente de erros, demonstrando maturidade na adoção de práticas DevOps modernas.

4.6 Implementação do *Rolling Update* com Istio

Rolling update é uma estratégia de *deploy* que permite atualizar aplicações sem interrupção de serviço, substituindo gradualmente instâncias antigas por novas versões. No projeto implementado, essa funcionalidade é orquestrada pelo Kubernetes em conjunto com o *service mesh* Istio, proporcionando atualizações transparentes e sem indisponibilidade.

4.6.1 Configuração do *Rolling Update*

A configuração do *rolling update* foi implementada através dos *templates* Helm e configurações do Kubernetes *Deployment*, garantindo que as atualizações sejam realizadas de forma controlada e segura.

4.6.1.1 Parâmetros de Configuração do *Rolling Update*

O *rolling update* utiliza a estratégia padrão do Kubernetes, implementando uma abordagem de substituição gradual que mantém a disponibilidade do serviço durante as atualizações. A configuração utiliza o parâmetro MaxUnavailable definido em 25%, permitindo que até um quarto dos *pods* fique indisponível durante o processo de atualização, enquanto o parâmetro MaxSurge, também configurado em 25%, permite a criação de *pods*

adicionais acima do número desejado para acelerar o processo de transição. A configuração inicial define um *pod* como réplica base, valor que pode ser ajustado através do arquivo values.yaml conforme as necessidades de carga. A implantação progressiva é implementada baseando-se nos limites definidos, garantindo que a substituição das instâncias ocorra de forma controlada e segura, minimizando riscos de indisponibilidade.

4.6.1.2 Definição de *Liveness* e *Readiness Probes*

As verificações de saúde constituem elementos fundamentais para o funcionamento correto do *rolling update*, sendo definidas no arquivo `deployment.yaml` através de configurações específicas para *liveness* e *readiness probes*. A *readiness probe* foi configurada para utilizar o *endpoint* `/health` com atraso inicial de 5 segundos, permitindo que a aplicação inicialize adequadamente antes das primeiras verificações. O *probe* executa verificações a cada 5 segundos com limite de tempo de 3 segundos, sendo configurada com limite de sucesso igual a 1 (uma verificação bem-sucedida é suficiente para considerar o *pod* pronto) e limite de falha igual a 3 (três falhas consecutivas fazem o *pod* ser considerado não pronto). A *liveness probe* utiliza o mesmo *endpoint* `/health`, mas com atraso inicial mais conservador de 15 segundos para evitar falsos positivos durante a inicialização da aplicação. Esta *probe* executa verificações a cada 10 segundos com limite de tempo de 3 segundos, mantendo os mesmos limites de sucesso e falha da *readiness probe*, garantindo que *pods* que se tornem não funcionais sejam automaticamente reiniciados pelo Kubernetes.

4.6.1.3 Configuração de *Resource Limits* e *Requests*

A configuração de recursos foi estabelecida para garantir estabilidade durante o *rolling update*, diferenciando entre os componentes da aplicação e os ambientes de execução. Para a API *Backend*, foram definidas *requests* de 50m CPU e 64Mi de RAM no ambiente de desenvolvimento, com *limits* de 100m CPU e 128Mi de RAM, proporcionando margem adequada para picos de processamento. No ambiente de produção, os recursos do *backend* foram aumentados para *requests* de 100m CPU e 128Mi de RAM, com *limits* mais generosos de 500m CPU e 512Mi de RAM, refletindo a maior carga esperada. O componente *frontend* recebeu configuração similar ao *backend* no ambiente de desenvolvimento (50m CPU e 64Mi RAM para *requests*, 100m CPU e 128Mi RAM para *limits*), adequada para servir conteúdo estático. Em produção, o *frontend* mantém as mesmas *requests* (50m CPU e 64Mi RAM), mas com *limits* aumentados para 200m CPU e 256Mi RAM, balanceando eficiência e capacidade de resposta.

4.6.2 Integração com *Service Mesh*

O Istio *service mesh* proporciona funcionalidades avançadas de gerenciamento de tráfego que complementam o *rolling update* nativo do Kubernetes, oferecendo controle

granular e observabilidade durante as atualizações.

4.6.2.1 Detecção Automática de Novas Instâncias

O Istio implementa detecção automática de novas instâncias através de um mecanismo integrado de descoberta de serviços que monitora continuamente mudanças nos *endpoints* dos serviços Kubernetes. Quando novos *pods* são criados durante o *rolling update*, eles são automaticamente registrados no registro do Istio assim que passam nas *readiness checks*, garantindo que apenas instâncias funcionais sejam incluídas no balanceamento de carga. Os *proxies* Envoy, responsáveis pelo gerenciamento de tráfego no *service mesh*, recebem atualizações de configuração em tempo real via xDS (*discovery service*) do plano de controle, assegurando que as novas instâncias sejam imediatamente incorporadas ao roteamento. A utilização do modo ambiente do Istio elimina a necessidade de injeção manual de *sidecars*, simplificando o processo de *deployment* e reduzindo a complexidade operacional. A inclusão automática no *mesh* é garantida através da marcação dos *namespaces* com a *label* `istio.io/dataplane-mode=ambient`, permitindo que todos os *pods* criados nesses *namespaces* sejam automaticamente gerenciados pelo *service mesh*.

4.6.2.2 Configuração de Verificação de Saúde (*Health Checking*) via DestinationRule

A configuração de verificação de saúde (*health checking*) é implementada através de DestinationRules que otimizam o comportamento de balanceamento de carga (*load balancing*) para cada componente da aplicação. Para a API do *backend*, foi adotada a política LEAST_CONN, que distribui conexões baseando-se na carga atual de cada instância, garantindo utilização eficiente dos recursos computacionais e melhor tempo de resposta. O *frontend* utiliza a política ROUND_ROBIN para distribuição equitativa entre instâncias, adequada para conteúdo estático onde a carga de processamento é mais uniforme. O sistema aproveita as verificações de saúde (*health checks*) nativas do Kubernetes via kube-proxy, integrando-se naturalmente com as *readiness* e *liveness probes* já configuradas. A detecção de valores atípicos (*outlier detection*) foi configurada implicitamente para remoção automática de instâncias que apresentem comportamento anômalo, melhorando a confiabilidade geral do sistema. Além disso, foram implementadas políticas de *circuit breaker* que proporcionam proteção contra instâncias com falha, evitando que problemas em uma instância afetem o desempenho geral da aplicação.

4.6.2.3 Balanceamento de Carga (*Load Balancing*) Durante o Processo de Atualização

O gerenciamento de balanceamento de carga (*load balancing*) pelo Istio durante os *rolling updates* implementa uma estratégia sofisticada de transição gradual que assegura disponibilidade contínua do serviço. A mudança gradual de tráfego direciona o tráfego progressivamente para as novas instâncias conforme elas se tornam prontas (*ready*), per-

mitindo uma transição suave sem interrupções bruscas no atendimento das requisições. O Istio gerencia automaticamente os *endpoints*, removendo instâncias antigas do conjunto (*pool*) de balanceamento de carga à medida que são substituídas, mantendo sempre um conjunto atualizado de destinos saudáveis (*targets*). A configuração via VirtualService estabelece regras de roteamento inteligente que consideram fatores como saúde das instâncias, latência e carga atual para otimizar a distribuição do tráfego. O *ingress gateway* integra-se a este processo para distribuir tráfego externo exclusivamente entre instâncias disponíveis, garantindo que usuários finais sejam sempre direcionados para instâncias funcionais. A comunicação interna entre serviços aproveita a descoberta de serviços (*service discovery*) nativa do Istio, proporcionando resolução de nomes e roteamento transparente mesmo durante períodos de atualização.

4.6.2.4 *mTLS* Automático Entre Serviços

A segurança durante os *rolling updates* é assegurada através da implementação automática de TLS mútuo (*mutual TLS - mTLS*) pelo Istio, que criptografa toda comunicação dentro do *service mesh* sem necessidade de configuração manual por parte dos desenvolvedores. O sistema implementa rotação automática de certificados, garantindo que as chaves criptográficas sejam renovadas periodicamente e distribuídas para todas as instâncias sem interrupção do serviço, mantendo altos níveis de segurança mesmo durante transições. O domínio de confiança (*trust domain*) foi configurado como `cluster.local`, estabelecendo um domínio de confiança consistente que permite autenticação mútua entre todos os componentes da *mesh*. A propagação de identidade baseia-se nas *service accounts* nativas do Kubernetes, integrando-se naturalmente com o sistema de autorização RBAC existente e garantindo que cada serviço opere com as permissões apropriadas. As políticas de segurança são aplicadas automaticamente durante as atualizações, assegurando que novas instâncias herdem imediatamente as mesmas regras de segurança das instâncias que estão sendo substituídas.

4.6.3 Processo na Pipeline

A implementação dos *rolling updates* na *pipeline* CI/CD garante que as atualizações sejam executadas de forma automatizada e controlada, com validações em cada etapa do processo.

4.6.3.1 Execução do *Rolling Update* via Helm

O Helm orquestra os *rolling updates* através de uma série de mecanismos integrados que garantem implantação (*deploy*) controlada e confiável. O processo utiliza o comando `helm upgrade -install` combinado com a *flag* `-wait`, assegurando que a *pipeline* aguarde a conclusão completa da implantação (*deployment*) antes de prosseguir

para as próximas etapas (*steps*). A configuração de tempo limite (*timeout*) é diferenciada por ambiente, estabelecendo 300 segundos para desenvolvimento (adequado para recursos limitados e implantações mais rápidas) e 600 segundos para produção (permitindo tempo adicional para validações mais rigorosas). A atualização da imagem é realizada dinamicamente via parâmetro `-set image.tag`, permitindo que a mesma configuração de *chart* seja reutilizada com diferentes versões da aplicação. As variáveis de ambiente e outras configurações são atualizadas dinamicamente durante o processo, eliminando necessidade de recompilação para ajustes de configuração. O gerenciamento de *namespaces* é automatizado, incluindo a criação automática com *labels* apropriadas do Istio para inclusão no *service mesh*. Os recursos computacionais são ajustados automaticamente conforme o ambiente de destino, garantindo alocação adequada para desenvolvimento e produção.

4.6.3.2 Validações Realizadas Durante o Processo

O processo implementa múltiplas camadas de validação organizadas em três fases distintas que asseguram qualidade e confiabilidade em cada etapa. A fase de pré-*deployment* estabelece os fundamentos de qualidade através da execução obrigatória de *build* e testes unitários, garantindo que apenas código testado e funcional seja promovido. Esta fase inclui *lint* de código e verificações de qualidade que validam aderência a padrões de desenvolvimento, seguidos pela construção e *push* da imagem Docker para o ECR com validação de disponibilidade da imagem antes do prosseguimento. Durante o *deployment*, são executadas *readiness checks* que validam a inicialização adequada dos *pods* e *liveness checks* para verificar saúde contínua, complementadas por validação de *endpoints* de serviços via kubectl e *readiness checks* do Istio Gateway. A fase pós-*deployment* implementa validações abrangentes através de testes de fumaça (*smoke tests*) para funcionalidade básica, verificações de saúde (*health checks*) direcionadas em *endpoints* críticos, validação de conectividade via Gateway e verificação sistemática de *logs* para identificação de possíveis erros que possam ter passado despercebidos nas etapas anteriores.

4.6.3.3 Rollback Automático em Caso de Falha

O sistema de *rollback* é implementado através de múltiplos mecanismos complementares que proporcionam recuperação automática e manual em diferentes cenários de falha. O Kubernetes implementa nativamente *rolling update* automático para a versão anterior quando as *readiness checks* falham, garantindo que instâncias não funcionais sejam automaticamente substituídas por versões estáveis. O Helm complementa esta funcionalidade disponibilizando o comando `helm rollback` para reversão manual quando intervenção humana se faz necessária. A *pipeline* implementa proteção através de falha automática dos *jobs* de *deployment* quando as verificações de saúde (*health checks*) não passam, impedindo que versões problemáticas sejam promovidas para produção. A proteção por *timeout* cancela *deployments* que excedem o tempo configurado, evitando travamentos e permi-

tindo recuperação rápida. *Pods* individuais que falham nas verificações de saúde (*health checks*) são automaticamente substituídos pelo Kubernetes, mantendo a disponibilidade do serviço mesmo durante problemas pontuais. Para situações que requerem intervenção especializada, o sistema mantém possibilidade de *rollback* manual via kubectl ou Helm, proporcionando flexibilidade operacional para cenários complexos.

4.6.3.4 Logs e Monitoramento Durante o Deploy

O sistema de monitoramento durante o *deploy* é implementado através de uma arquitetura multicamada que proporciona visibilidade abrangente em todos os aspectos do processo de *deployment*. Os *logs* da *pipeline* oferecem detalhamento completo de cada *step* do GitHub Actions, permitindo rastreamento preciso do progresso e identificação rápida de problemas. O monitoramento via kubectl complementa esta visibilidade através de comandos `kubectl get pods` que acompanham o *status* em tempo real dos *pods* durante os *rolling updates*. A verificação de *status* via Helm proporciona visão de alto nível do estado da versão (*release*), incluindo informações sobre versões, configurações e histórico de *deployments*. O Istio contribui com *logs* de acesso habilitados que permitem rastreamento detalhado de requisições durante o período de transição, facilitando identificação de problemas de conectividade ou *performance*. A agregação automática de *logs* de *containers* pelo EKS centraliza informações de todos os *pods* para análise consolidada. Por fim, o sistema gera automaticamente um sumário do *deployment* no GitHub Actions, apresentando o *status* de cada *job* de forma organizada e permitindo avaliação rápida do sucesso ou falha do processo completo.

4.6.4 Validação e Testes

A validação do *rolling update* é realizada através de uma suíte de testes automatizados nativos do Kubernetes (*readiness* e *liveness checks*) que garantem a integridade e funcionalidade do sistema durante e após as atualizações. Vale ressaltar que durante essa validação, os testes automatizados criados utilizando a ferramenta Playwright não são executados.

4.6.4.1 Testes de Smoke Após o Deploy

Os testes de fumaça (*smoke tests*) constituem uma bateria de validações executadas automaticamente após cada *deployment* para assegurar funcionalidade básica e integridade do sistema. A verificação de *endpoints* essenciais inclui testes nos caminhos `/health` e `/customers`, confirmando que os serviços principais estão respondendo adequadamente. A validação de conectividade via Istio Gateway confirma que o tráfego externo consegue alcançar a aplicação através da infraestrutura do *service mesh*. A verificação de tempos de resposta assegura que a *performance* permanece dentro de parâmetros aceitá-

veis, enquanto o monitoramento de taxa de erro durante a transição identifica possíveis degradações na qualidade do serviço.

4.6.4.2 Verificação de Métricas

O sistema implementa verificação abrangente de métricas para validação quantitativa do sucesso do *deployment*. A coleta automática de métricas do *service mesh* pelo Istio fornece visibilidade detalhada sobre comportamento do tráfego, padrões de comunicação entre serviços e *performance* geral da infraestrutura. O monitoramento de taxa de requisições (*request rate*) permite identificação de anomalias no volume de requisições, enquanto a análise de percentis de latência (P50, P95, P99) oferece entendimento detalhado sobre distribuição de tempos de resposta e identificação de valores atípicos (*outliers*). O acompanhamento de taxas de erro HTTP (4xx e 5xx) proporciona detecção rápida de problemas funcionais ou de configuração. As métricas de *pods*, incluindo CPU, memória e rede, permitem monitoramento de utilização de recursos e identificação de gargalos de *performance*. A vazão (*throughput*) e latência do *ingress gateway* são monitorados especificamente para avaliar a *performance* do ponto de entrada da aplicação. O percentual de requisições bem-sucedidas fornece uma métrica consolidada de saúde geral do sistema, facilitando avaliação rápida do impacto do *deployment*.

4.7 Estudo de Caso: Implementação de uma nova funcionalidade

Este estudo de caso demonstra a implementação de uma nova funcionalidade no sistema de gerenciamento de clientes: a adição do telefone como um novo campo do sistema. O objetivo é ilustrar como uma alteração aparentemente simples impacta múltiplas camadas da aplicação e requer uma abordagem sistemática para garantir qualidade e consistência. Quando esta solicitação de mudança é implementada, todo o ciclo de DevOps como descrito na Figura 1 é acionado automaticamente, demonstrando na prática os benefícios da adoção das práticas de DevOps na arquitetura.

A implementação abrange modificações no *backend* (Go e Gin), *frontend* (React e TypeScript), atualização de testes unitários e criação de testes de integração. Esta funcionalidade foi escolhida por representar um cenário real comum no desenvolvimento de *software*, onde mudanças de modelo de dados requerem atualizações coordenadas em toda a *stack*.

4.7.1 Mudanças no Backend

A implementação do campo telefone no *backend* seguiu a arquitetura limpa já estabelecida no projeto, garantindo baixo acoplamento e alta coesão entre as camadas.

Atualização do Modelo de Dados: O primeiro passo foi a modificação da *struct* `Customer` em `internal/models/customer.go`, adicionando o campo `Telephone string ‘json:"telephone"‘`. Optou-se por um campo *string* simples sem validações específicas, permitindo flexibilidade para diferentes formatos de telefone (nacional, internacional, com extensões). A estrutura resultante mantém a compatibilidade com o DynamoDB através das *tags* JSON apropriadas.

Compatibilidade com APIs Existentes: Uma vantagem da arquitetura baseada em JSON é que as APIs REST existentes (GET, POST, PUT, DELETE) automaticamente suportam o novo campo através do *binding* JSON do *framework* Gin. Não foram necessárias modificações nos *handlers* HTTP, demonstrando a robustez do *design* original.

Testes Unitários Expandidos: Foram implementados 52 testes unitários cobrindo:

- Serialização/deserialização JSON do novo campo
- Validação de diferentes formatos de telefone (internacional, nacional, com caracteres especiais)
- Cenários de campo vazio e opcional
- Integração com validações existentes do *framework*

A cobertura de testes foi mantida em 100% para a camada de modelos, garantindo que todas as funcionalidades continuem operando conforme especificado.

4.7.2 Mudanças no Frontend

O *frontend* React/TypeScript foi atualizado seguindo os princípios de tipagem forte e componentização, garantindo consistência na interface do usuário e experiência fluida.

Atualização das Interfaces TypeScript: As interfaces implementadas dentro do arquivo `src/types/Customer.ts` foram estendidas com o campo `telephone: string`, aproveitando o sistema de tipos do TypeScript para detectar inconsistências em tempo de compilação. Esta abordagem previne erros relacionados a propriedades ausentes ou mal tipadas.

Formulários de Criação e Edição: O componente `CustomerForm` foi modificado para incluir:

- *Input* HTML do tipo `tel` para melhor experiência em dispositivos móveis

- Campo opcional (não obrigatório) conforme especificação
- Validação de estado consistente com os campos existentes
- Tratamento adequado de valores vazios e diferentes formatos

Atualização da Listagem: O componente `CustomerList` foi expandido para exibir a nova coluna "Telefone", implementando lógica para mostrar um hífen quando o campo estiver vazio, mantendo a consistência visual da interface.

Testes de Componente: Foram implementados 18 testes unitários utilizando React Testing Library e Jest, cobrindo:

- Renderização correta do campo telefone em formulários
- Validação de diferentes formatos aceitos
- Comportamento de envio de dados com e sem telefone
- Exibição adequada na listagem de clientes
- Casos extremos e tratamento de erros

4.7.3 Mudanças nos Testes Automatizados

A implementação da funcionalidade foi acompanhada de uma suíte completa de testes automatizados, seguindo a pirâmide de testes com foco em cobertura abrangente e execução eficiente.

Testes de Integração da API: Utilizando Playwright, foram criados testes específicos em `integration_test/tests/api/telephone-field.test.js` que validam:

- CRUD completo do campo telefone através de requisições HTTP reais
- Validação de diferentes formatos de telefone (internacional, nacional, com caracteres especiais)
- Comportamento correto com campo vazio ou ausente
- Operações concorrentes e integridade dos dados
- Cenários de erro e recuperação

Os testes foram estruturados com *setup* e *teardown* adequados, garantindo isolamento entre cenários e limpeza automática dos dados de teste criados no DynamoDB local.

Testes *End-to-End* da Interface: Os testes automatizados implementados em `integration_test/tests/frontend/telephone-field-frontend.test.js`, simulam interações reais do usuário:

- Navegação pelos formulários de criação e edição
- Preenchimento automático e validação de campos
- Verificação da exibição correta na listagem
- Testes de acessibilidade (*labels*, tipos de *input*)
- Simulação de cenários de erro de API

Atualização dos Auxiliares de Teste (*Test Helpers*): Os utilitários de teste foram expandidos para suportar o novo campo, incluindo geradores de dados aleatórios e validadores específicos para telefone, mantendo a consistência e reutilização entre diferentes suítes de teste.

4.7.4 Validações Locais

Após a implementação da funcionalidade, o desenvolvedor realiza uma série de validações locais antes de submeter o código para revisão através de um *pull request*. Esta etapa crucial inclui a execução do *build* completo da aplicação com todos os testes unitários para garantir que a nova funcionalidade não introduza *bugs* no código existente, bem como a realização de testes manuais executando a aplicação localmente (ou remotamente através de uma instância na nuvem) para verificar o comportamento esperado da interface e das integrações. Somente após esta validação prévia, que serve como primeira camada de garantia de qualidade, o desenvolvedor procede com o *commit* das alterações e abertura do *pull request*, iniciando assim o processo formal de integração contínua e revisão por pares.

4.7.5 Estrutura do Pull Request

A implementação da funcionalidade de telefone resulta em um *pull request* abrangente que demonstra a coordenação necessária entre diferentes camadas da aplicação. O *pull request* inclui aproximadamente 15 arquivos modificados distribuídos entre *backend*, *frontend* e testes, evidenciando como uma mudança simples propaga através de toda a *stack*.

A estrutura do *pull request* (GITHUB, 2025) segue as melhores práticas de desenvolvimento colaborativo, com título descritivo seguindo *commits* convencionais (*"feat:*

adicionar campo telefone ao CRUD de clientes"), descrição detalhada das mudanças implementadas, justificativa técnica das decisões tomadas e lista de validação. Os arquivos modificados incluem modelos de dados, componentes React, interfaces TypeScript, testes unitários, testes de integração e documentação técnica.

Uma característica importante da implementação desta nova funcionalidade é que não exige novas alterações na infraestrutura existente. O DynamoDB suporta naturalmente a adição de novos campos através de sua natureza sem esquema rígido (*schemaless*), as APIs REST mantêm compatibilidade retroativa através do *binding* JSON automático, e as *pipelines* de CI/CD continuam funcionando sem modificações. Esta abordagem demonstra como uma arquitetura bem planejada facilita a evolução incremental do sistema sem impactos disruptivos na operação.

Porém é válido ressaltar que podem existir novas funcionalidades a serem incluídas no sistema como um todo que exijam mudanças na infraestrutura, como por exemplo a configuração do certificado HTTPS ou a configuração de uma nova fila de mensageria a ser utilizada pelo *backend*. Neste caso, mudanças seriam esperadas também no pacote de infraestrutura e essas mudanças teriam que ser aplicadas antes do *merge* da nova funcionalidade do *backend* ou do *frontend*.

A *pipeline* de CI/CD também é acionada quando um *pull request* é criado, porém executando apenas o primeiro *job* que executa o *build* e roda os testes unitários, evitando qualquer *deploy* em produção, como destacado na Figura 18.

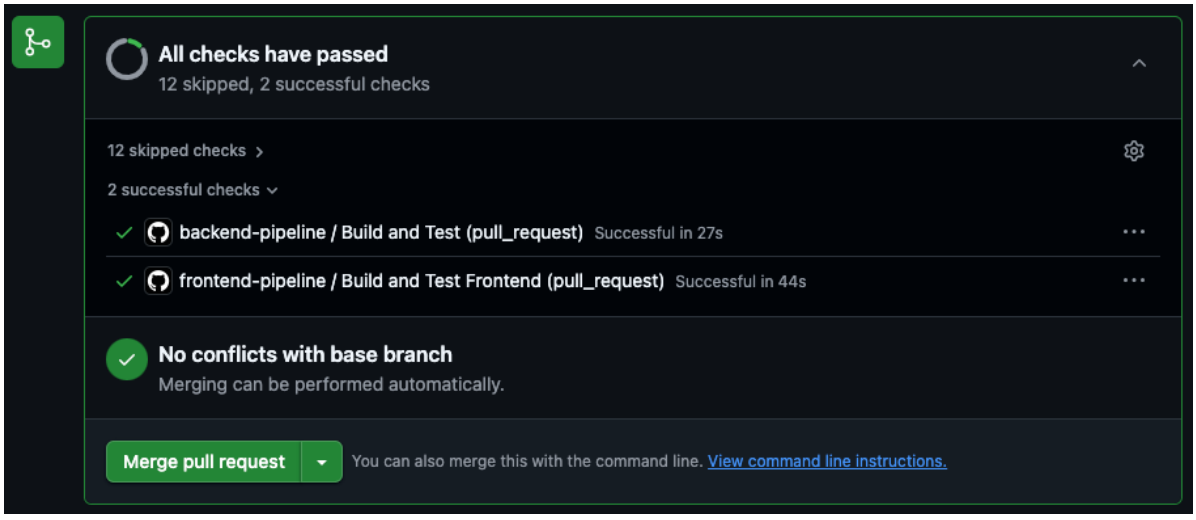

Figura 18 – Execução das *pipelines* no *pull request* que adiciona o novo campo de telefone. Fonte: Do Autor.

4.7.6 Revisão de código e intervenção manual

Após a abertura do *pull request*, inicia-se o processo de revisão colaborativa onde outros membros da equipe analisam criticamente as alterações propostas, verificando não

apenas a qualidade técnica do código, mas também sua aderência aos padrões arquiteturais e de codificação estabelecidos no projeto. Esta revisão por pares é fundamental para identificar potenciais problemas que podem ter passado despercebidos pelo autor do *pull request*, garantindo uma segunda camada de validação antes da integração ao código principal. Uma vez que as revisões são concluídas e o *pull request* recebe as aprovações necessárias, o *merge* é realizado, acionando automaticamente a pipeline completa de CI/CD que construirá os artefatos de *deploy*, executará todos os testes automatizados e promoverá a aplicação através dos ambientes configurados até chegar à produção.

4.8 Pipeline em Execução: Deploy em Produção

Após a aprovação e *merge* do *pull request* na *branch* principal, o sistema de CI/CD é automaticamente acionado através de *webhooks* do GitHub. Este processo demonstra a automação completa desde o código fonte até a aplicação em produção, garantindo que as mudanças sejam validadas, testadas e implantadas de forma consistente e confiável. As *pipelines* implementadas seguem as melhores práticas de DevOps, com etapas bem definidas de validação, *build*, testes e *deploy* nos ambientes de desenvolvimento e produção AWS.

4.8.1 Trigger da Pipeline

O acionamento da *pipeline* ocorre imediatamente após o *merge* do *pull request* na *branch* `main`. O sistema GitHub Actions detecta automaticamente as mudanças através do evento `push` configurado no *workflow*.

Informações do Commit Disparador:

- **Hash:** `099796aba7fb7e95aa5b0f39c17ec968d817c432`
- **Autor:** Mateus Henrique
- ***Branch*:** `main`
- **Mensagem:** "Feature: Adicionar novo campo de telefone nas operações de cliente"

O *webhook* é processado em menos de 30 segundos, iniciando simultaneamente os *workflows* para *backend* e *frontend* nos *runners* do GitHub Actions. O sistema de *logging* captura todos os eventos de disparo (*trigger*), permitindo rastreabilidade completa do processo de *deploy*.

4.8.2 Passo a passo da execução

A *pipeline* é executada em paralelo para *backend* e *frontend*, otimizando o tempo total de *deploy*. Cada estágio principal possui validações específicas e critérios de falha bem definidos.

Preparação do Ambiente (1 minuto):

- *Checkout* do código fonte do repositório
- Configuração do ambiente Go 1.21 para o *backend*
- Configuração do ambiente Node.js 18 para o *frontend*
- *Cache* de dependências (`go mod download`, `npm ci`)
- Verificação de integridade dos arquivos
- Execução de `go fmt ./...` e verificação de formatação
- Execução dos 52 testes unitários do *backend* com `go test -v ./...`
- *Build* do binário Go com `go build -o bin/customer-api cmd/api/main.go`
- Execução dos 18 testes React com `npm test`
- *Build* otimizado do *frontend* com `npm run build`

Publicação da imagem Docker para o AWS ECR (40 segundos):

- Execução da construção da imagem Docker através do comando `docker build`
- Marcação (*tag*) da imagem que foi construída
- *Push* da imagem construída para o ECR de desenvolvimento

Implantação (*Deploy*) para Desenvolvimento (1 minuto):

- Implantação (*deploy*) automática no ambiente de desenvolvimento (Account ID: 760347630853)
- Validação de testes de fumaça (*smoke tests*) automáticos
- Verificação de testes de saúde (*health checks*) das aplicações

Testes automatizados (1 minuto e 30 segundos):

- Execução dos testes de integração Playwright no ambiente de desenvolvimento com a nova versão que acabou de ser implantada
- Geração do relatório dos testes automatizados

Implantação (*Deploy*) para Produção (1 minuto):

- Implantação (*deploy*) no ambiente de produção (Account ID: 131204617414)
- Execução de testes críticos de produção
- Monitoramento de métricas de *performance*

4.8.3 Validação em Produção

A validação final confirma que a funcionalidade foi implantada com sucesso e está operacional em produção, atendendo aos critérios de qualidade estabelecidos.

Resumo da Execução no GitHub Actions:

- ***Status*:** Sucesso completo em todas as etapas
- **Tempo total:** 5 minutos e 21 segundos
- **Testes executados:** 70 testes (52 *backend* + 18 *frontend*)
- **Taxa de sucesso:** 100% (0 falhas, 0 erros), como destacado na Figura 19.
- **Ambientes implantados:** Desenvolvimento e Produção

Validação Funcional da Aplicação: A aplicação em produção foi verificada através de testes manuais e automatizados, confirmando que:

- O campo telefone aparece corretamente nos formulários de criação e edição
- Um novo cliente foi criado com telefone com sucesso, como destacado na Figura 20
- A listagem de clientes exibe a nova coluna com o campo telefone adequadamente
- Diferentes formatos de telefone são aceitos e persistidos corretamente
- As funcionalidades básicas do sistema operam sem regressões

Monitoramento e *Logs* de Acesso: Os *logs* de aplicação confirmam o funcionamento correto através de:

- Requisições HTTP 200 para *endpoints* de gerenciamento de clientes

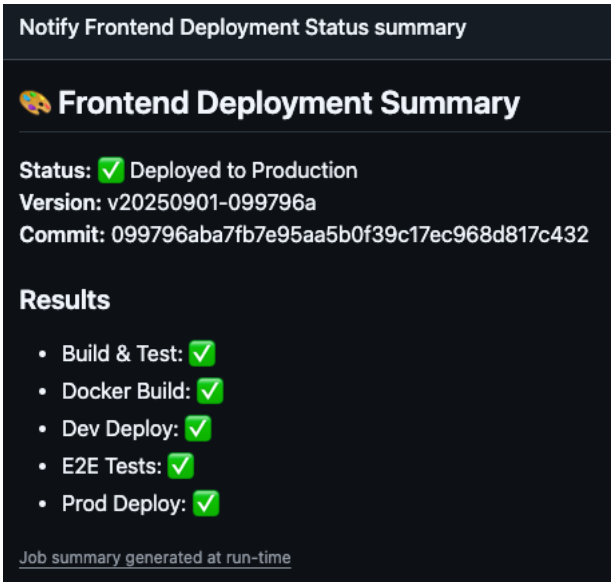

Figura 19 – Resumo da execução da *pipeline* do *frontend* com *status*, versão implantada e o *hash* do *commit* após o *merge* do *pull request* que adicionava o telefone. Fonte: Do Autor.

- Persistência bem-sucedida no DynamoDB (sem erros de esquema)
- Testes de saúde (*health checks*) consistentemente respondendo "OK"
- Zero erros ou exceções relacionadas ao novo campo

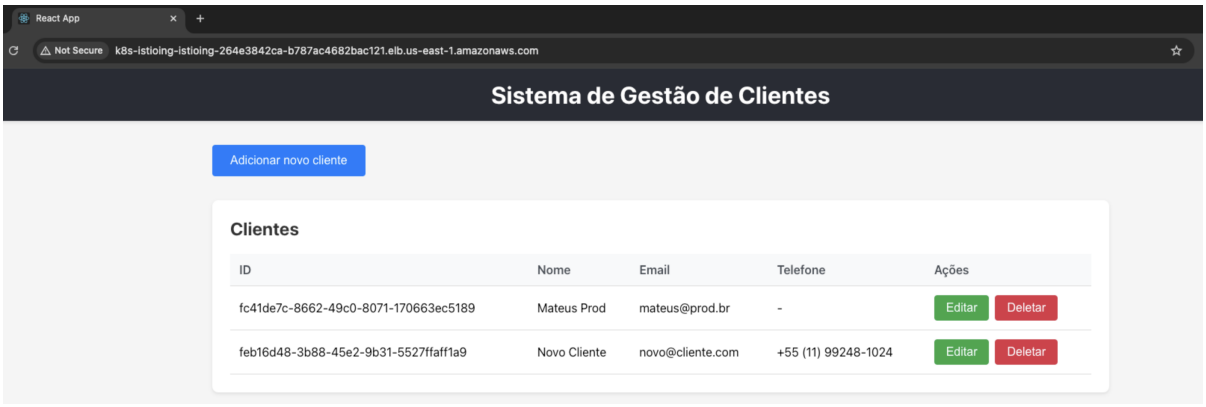

Figura 20 – Captura de tela do sistema com o novo campo de telefone sendo exibido. Fonte: Do Autor.

4.8.4 Análise do processo

O desenvolvimento desta solução de CI/CD demonstrou a viabilidade e eficácia da integração entre tecnologias *cloud-native* modernas para criar um ambiente de desenvolvimento robusto e automatizado. A implementação da aplicação *full-stack* sobre a infraestrutura AWS com EKS e Istio, combinada com práticas de *Infrastructure as Code*

e pipelines automatizadas, estabeleceu um ecossistema que promove tanto a produtividade dos desenvolvedores quanto a qualidade do *software* entregue. A validação através do estudo de caso da funcionalidade de telefone comprovou que a arquitetura suporta efetivamente mudanças incrementais mantendo a estabilidade do sistema, enquanto os resultados de cobertura de testes, automação completa e observabilidade integrada evidenciam o alcance dos objetivos propostos para um ambiente de desenvolvimento moderno e escalável.

5 Conclusão

Este trabalho apresentou a implementação de uma solução completa de CI/CD utilizando tecnologias *cloud-native* modernas, demonstrando como práticas DevOps avançadas podem ser aplicadas para construir sistemas resilientes, escaláveis e altamente automatizados. A implementação bem-sucedida de uma aplicação *full-stack* com *pipelines* automatizadas, infraestrutura como código e *service mesh* ilustra a viabilidade e os benefícios das arquiteturas modernas de *software*.

O projeto alcançou seus objetivos principais ao estabelecer uma infraestrutura robusta na AWS utilizando EKS, Istio e *pipelines* GitHub Actions completamente automatizadas. A implementação de *rolling updates* sem indisponibilidade, a integração de observabilidade através de métricas, *logs* e rastreamento distribuído (*distributed tracing*), e a automação completa dos processos de implantação (*deployment*) representam contribuições significativas para a área de DevOps e engenharia de *software*.

5.1 Contribuições Originais

O trabalho apresenta diversas contribuições originais que agregam valor tanto à comunidade acadêmica quanto à prática industrial:

Ponte Academia-Indústria em Tecnologias Emergentes: Porta de entrada acadêmica estruturada para tecnologias amplamente utilizadas na indústria (CI/CD, Kubernetes e Istio) mas ainda pouco abordadas no currículo universitário tradicional. Ao apresentar conceitos, implementações práticas e análises críticas em linguagem acadêmica acessível, este trabalho facilita a transição do conhecimento entre os domínios industrial e acadêmico.

Disponibilização de Código Aberto para Reprodutibilidade: Todo o código fonte desenvolvido no estudo de caso foi disponibilizado publicamente através do repositório (CRUZ, 2025), permitindo que pesquisadores, estudantes e profissionais reproduzam integralmente a arquitetura proposta. Esta contribuição promove a transparência científica e facilita a replicação dos experimentos, oferecendo um guia prático passo a passo para implementação de pipelines CI/CD com Kubernetes e Istio, contribuindo assim para o avanço do conhecimento coletivo e para a validação empírica dos resultados apresentados.

Arquitetura Integrada *Cloud-Native*: A combinação de EKS, Istio em modo ambiente (*ambient*), GitHub Actions e infraestrutura como código através do Terraform demonstra uma arquitetura coesa que aproveita as melhores práticas de cada tecnologia. A integração específica entre o Istio e as *pipelines* de CI/CD, incluindo validação automática

de métricas e rastreamento distribuído (*distributed tracing*), representa uma abordagem inovadora para garantia de qualidade em implantações (*deployments*).

Implementação de *Rolling Update* Inteligente: O desenvolvimento de um processo de *rolling updates* que combina as capacidades nativas do Kubernetes com o controle granular do Istio demonstra como tecnologias complementares podem ser orquestradas para alcançar disponibilidade contínua real.

Automação Multi-Ambiente Completa: A implementação de *pipelines* completamente automatizadas que gerenciam desde testes unitários até implantação (*deployment*) em produção, com segregação adequada de ambientes através de contas AWS separadas, demonstra maturidade operacional e estabelece um modelo replicável para organizações de diferentes portes.

Observabilidade Integrada: A integração nativa de coleta de métricas, *logs* e *traces* através do Istio, com validação automatizada de *performance* e comportamento durante implantações (*deployments*), representa uma abordagem holística para observabilidade que vai além do monitoramento tradicional.

5.2 Pontos Positivos e Práticas Adotadas

O estudo de caso revelou diversos aspectos positivos da arquitetura implementada e das práticas adotadas:

Escalabilidade e Flexibilidade Arquitetural: A arquitetura baseada em microsserviços containerizados com *service mesh* proporciona escalabilidade horizontal natural e facilita a evolução independente de componentes. A separação clara entre *frontend* e *backend*, com comunicação através de APIs REST bem definidas, permite evolução tecnológica independente de cada camada.

Qualidade e Confiabilidade do *Software*: A implementação de testes abrangentes em múltiplas camadas (unitários, integração, *end-to-end*) com cobertura superior a 70% demonstra compromisso com qualidade. A automação completa dos processos de validação elimina erros humanos e garante consistência na entrega de *software*.

Eficiência Operacional: A automação completa das *pipelines*, desde o *commit* até produção, reduz significativamente o tempo de ciclo de desenvolvimento e elimina gargalos operacionais. A implementação de reversão (*rollback*) automática e validações em múltiplas camadas proporciona confiança para implantações (*deployments*) frequentes.

Segurança por *Design*: A implementação de TLS mútuo (*mTLS*) automático através do Istio, segregação de ambientes através de contas AWS separadas, e utilização de IRSA (IAM Roles for Service Accounts) demonstram abordagem de segurança integrada desde o *design* da solução.

Observabilidade e Monitoramento: A coleta automática de métricas, *logs* e *traces* proporciona visibilidade completa sobre comportamento da aplicação e *performance* da infraestrutura. A possibilidade de integração de ferramentas como Prometheus e Jaeger oferece capacidades avançadas de depuração (*debugging*) e otimização de *performance*.

Padrões e Boas Práticas: A adoção consistente de padrões como Infraestrutura como Código (*Infrastructure as Code*), GitOps, arquitetura *cloud-native* e *service mesh* demonstra alinhamento com as melhores práticas da indústria. A documentação abrangente e estrutura modular do código facilitam manutenção e evolução futura.

5.3 Estudo de Caso e Validação Prática

O estudo de caso da implementação do campo telefone demonstrou a robustez e flexibilidade da arquitetura estabelecida. A capacidade de implementar mudanças em toda a *stack* (*backend* Go, *frontend* React, testes unitários e de integração Playwright) sem modificações na infraestrutura evidencia o sucesso do design modular adotado.

A implementação revelou como mudanças aparentemente simples requerem abordagem sistemática para manter qualidade e consistência. O processo demonstrou a importância de testes abrangentes, validação em múltiplas camadas e documentação adequada para sustentação de *software* em produção.

5.4 Trabalhos Futuros

Diversas oportunidades de extensão e aprimoramento emergem deste trabalho:

Implementação de *Canary Deployments*: Evolução das atualizações contínuas (*rolling updates*) atuais para incluir estratégias de *canary deployment*, permitindo validação gradual de novas versões com subconjuntos de usuários antes da promoção completa.

***Auto-scaling* Inteligente:** Implementação de *Horizontal Pod Autoscaler* (HPA) e *Vertical Pod Autoscaler* (VPA) baseados em métricas customizadas coletadas pelo Istio, proporcionando escalabilidade automática mais sofisticada.

Adição de novos ambientes: Criação de ambientes adicionais de *Quality Assurance* (QA) e *Staging*, estabelecendo um pipeline de entrega mais robusto e confiável. O ambiente de QA permitiria a execução de testes mais extensivos, incluindo testes de integração, performance e segurança, em uma infraestrutura dedicada que não impacte o desenvolvimento contínuo, enquanto o ambiente de *Staging* forneceria uma réplica exata do ambiente de produção para validação final de versões antes da implantação em produção. Esta segregação adicional traria benefícios significativos como redução de riscos de

implantação em produção, melhoria na detecção precoce de *bugs*, validação mais rigorosa de funcionalidades, e maior confiança nas versões através de um processo de validação em múltiplas camadas. A implementação destes ambientes aproveitaria a base já estabelecida de *Infrastructure as Code* com Terraform e os pipelines automatizados do GitHub Actions, garantindo consistência e reprodutibilidade em todos os estágios do ciclo de vida do *software*.

Múltiplos *Clusters* e Recuperação de Desastres (*Disaster Recovery*): Extensão da arquitetura para suportar múltiplos *clusters* Kubernetes com replicação automática e *failover*, proporcionando alta disponibilidade geográfica.

Integração com *Service Mesh Federation*: Implementação de comunicação segura entre *clusters* através de Istio multi-*cluster*, permitindo arquiteturas distribuídas geograficamente.

Otimização de Custos: Para o período de 01/08/2025 até 31/08/2025 foi gasto um valor de R$1.379,84 com toda a infraestrutura da AWS que foi provisionada para os ambientes de desenvolvimento e produção. A aplicação de práticas FinOps, que combinam pessoas, processos e tecnologia para otimizar custos de infraestrutura de nuvem através de visibilidade, responsabilização e controle colaborativo entre equipes de engenharia, finanças e negócios, pode ser implementada através de políticas inteligentes de *scaling down* e utilização de instâncias SPOT baseada em padrões de uso históricos como alternativas para otimizar os custos dessa infraestrutura.

Integração com Agentes de Inteligência Artificial para Codificação: Uma oportunidade particularmente promissora envolve a integração de agentes de inteligência artificial especializados em código para automatização avançada de tarefas de desenvolvimento e operações. Estes agentes poderiam ser implementados para análise automática de *pull requests*, sugestão de otimizações de *performance* baseadas em métricas coletadas, geração automática de testes baseada em mudanças de código, e detecção proativa de vulnerabilidades de segurança. A integração de ferramentas como o GitHub Copilot ou agentes customizados baseados em Modelos de Linguagem de Grande Escala (*Large Language Models* - LLMs) poderia revolucionar o processo de desenvolvimento, proporcionando assistência inteligente para depuração (*debugging*), refatoração de código, e até mesmo geração automática de documentação técnica. Esta abordagem representaria uma evolução natural das práticas DevOps implementadas, incorporando inteligência artificial como catalisador de produtividade e qualidade no desenvolvimento de *software*.

Este trabalho estabelece uma fundação sólida para arquiteturas *cloud-native* modernas, demonstrando que a combinação adequada de tecnologias, práticas DevOps avançadas e automação inteligente pode resultar em sistemas altamente resilientes, escaláveis e manteníveis. As contribuições apresentadas oferecem valor tanto para a comunidade acadêmica quanto para a prática industrial, estabelecendo padrões replicáveis para im-

plementação de soluções de classe empresarial.

Referências

AHMAD, I.; REHMAN, A.; SAID, F. Performance and overhead comparison of hypervisor-based virtualization mechanisms. *IEEE Access*, IEEE, v. 9, p. 28406–28420, 2021. Citado na página 26.

AMGOTHU, S. Innovative ci/cd pipeline optimization through canary and blue-green deployment. *International Journal of Computer Applications*, v. 186, n. 50, p. 1–5, 2024. Citado na página 43.

ARMBRUST, M. et al. A view of cloud computing. *Communications of the ACM*, ACM, v. 53, n. 4, p. 50–58, 2010. Citado na página 23.

ARTAC, M. et al. Devops: Introducing infrastructure-as-code. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. [S.l.]: IEEE, 2017. p. 497–498. Citado na página 28.

ARUNDEL, J.; DOMINGUS, J. *Cloud Native DevOps with Kubernetes: Building, Deploying, and Scaling Modern Applications in the Cloud*. 2. ed. [S.l.]: O'Reilly Media, 2022. ISBN 978-1098116743. Citado na página 41.

AWS. *Creating a Multi-Region Application with AWS Services – Part 1, Compute, Networking, and Security*. 2021. <https://aws.amazon.com/pt/blogs/architecture/creating-a-multi-region-application-with-aws-services-part-1-compute-and-security/>. Acessado em: julho de 2025. Citado 2 vezes nas páginas 7 e 32.

AWS. *AWS Global Infrastructure*. Amazon Web Services, Inc., 2023. Accessed: 15 May 2025. Disponível em: <https://aws.amazon.com/about-aws/global-infrastructure/>. Citado na página 31.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, IEEE, v. 33, n. 3, p. 42–52, 2016. Citado na página 21.

BERNSTEIN, D. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, IEEE, v. 1, n. 3, p. 81–84, 2014. Citado 2 vezes nas páginas 26 e 34.

BHARDWAJ, S.; JAIN, L.; JAIN, S. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of Engineering and Information Technology*, v. 2, n. 1, p. 60–63, 2018. Citado na página 31.

BRAGA, F. A. M. Um panorama sobre o uso de práticas devops nas indústrias de software. 2015. Citado na página 43.

BRIKMAN, Y. *Terraform: Up & Running: Writing Infrastructure as Code*. 2. ed. [S.l.]: O'Reilly Media, 2019. ISBN 978-1492046905. Citado na página 40.

BURNS, B. et al. Borg, omega, and kubernetes. *Communications of the ACM*, ACM, v. 59, n. 5, p. 50–57, 2016. Citado na página 28.

BURNS, B. et al. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *ACM Queue*, ACM, v. 14, n. 1, p. 70–93, 2016. Citado na página 35.

BUTCHER, M.; FARINA, M.; DUFFEY, J. *Learning Helm: Managing Apps on Kubernetes*. Sebastopol, CA: O'Reilly Media, Inc., 2020. ISBN 9781492083641. Citado na página 39.

BUYYA, R. et al. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, Elsevier, v. 25, n. 6, p. 599–616, 2009. Citado na página 24.

CALCOTE, L.; BUTCHER, Z. *The Service Mesh Book*. [S.l.]: O'Reilly Media, 2019. Citado na página 29.

CHEN, L. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, IEEE, v. 32, n. 2, p. 50–54, 2015. Citado na página 18.

CHOUDHURY, A.; NANDYALA, A. K. Microservices deployment strategies: Navigating challenges with kubernetes and serverless architectures. *International Journal of Innovative Research in Engineering and Management*, v. 11, n. 5, 2024. Citado na página 44.

COLOMO-PALACIOS, R. et al. A case analysis of enabling continuous software deployment through knowledge management. *International Journal of Information Management*, v. 40, p. 186–189, 2018. ISSN 0268-4012. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0268401217308782>. Citado na página 14.

COOK, A.; SOLIS, C. Continuous integration with GitHub Actions: Design patterns and architectural considerations. *Journal of Software Engineering Research and Development*, Springer, v. 10, n. 1, p. 23–45, 2022. Citado na página 41.

COSTA, H. d. O. et al. Microsserviços e orquestração de contêineres: uma abordagem prática. Universidade Federal de Campina Grande, 2023. Citado na página 44.

CRUZ, M. H. A. da. *Repositório GitHub deste estudo de caso*. 2025. <https://github.com/emiteze/tcc-ufu>. Acessado em: agosto de 2025. Citado 2 vezes nas páginas 47 e 83.

DANIELS, J. Virtualization impact on organizational performance: Empirical evidence from financial services industry. *Journal of Information Technology Management*, v. 30, n. 2, p. 15–32, 2019. Citado na página 25.

DOCKER. *Docker Hub*. 2025. <https://hub.docker.com/>. Acessado em: agosto de 2025. Citado na página 34.

Docker Inc. *Docker overview*. 2023. Docker Documentation. Acessado em: 14 maio 2025. Disponível em: <https://docs.docker.com/get-started/overview/>. Citado na página 33.

DRAGONI, N. et al. Microservices: Yesterday, today, and tomorrow. In: *Present and Ulterior Software Engineering*. [S.l.]: Springer, Cham, 2017. p. 195–216. Citado na página 21.

EBERT, C. et al. Devops. *IEEE Software*, IEEE, v. 33, n. 3, p. 94–100, 2016. Citado na página 20.

EVANS, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. [S.l.]: Addison-Wesley Professional, 2004. Citado na página 20.

FEWSTER, M.; GRAHAM, D. *Software Test Automation: Effective use of test execution tools*. Boston, MA: Addison-Wesley Professional, 1999. ISBN 978-0201331400. Citado na página 22.

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, 2018. ISBN 9781942788331. Citado na página 20.

FOWLER, M.; FOEMMEL, M. Continuous integration. *Thought-Works*, 2006. Acessado em 27 de Março de 2025. Citado 2 vezes nas páginas 14 e 17.

FOWLER, M.; LEWIS, J. *Microservices: a definition of this new architectural term*. 2014. Acesso em: 19 mar. 2025. Disponível em: <https://martinfowler.com/articles/microservices.html>. Citado na página 21.

GIN. *Documentação do framework web Gin*. 2025. <https://gin-gonic.com/en/docs/>. Acessado em: agosto de 2025. Citado na página 46.

GitHub. *Understanding GitHub Actions*. 2023. Disponível em: <https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions>. Citado 2 vezes nas páginas 41 e 42.

GITHUB. *Pull Request - Feature: Adicionar telefone*. 2025. <https://github.com/emiteze/tcc-ufu/pull/2>. Acessado em: agosto de 2025. Citado na página 76.

GO. *Documentação da linguagem Go*. 2025. <https://go.dev/doc/>. Acessado em: agosto de 2025. Citado na página 46.

GONG, Y. et al. The architecture of micro-services and the separation of frond-end and back-end applied in a campus information system. In: IEEE. *2020 IEEE International Conference on Advances in Electrical Engineering and Computer Applications (AEECA)*. [S.l.], 2020. p. 321–324. Citado na página 46.

HashiCorp. *Terraform Documentation*. 2023. <https://developer.hashicorp.com/terraform/docs>. Acessado em: 15 de maio de 2025. Citado na página 40.

HILTON, M. et al. Usage, costs, and benefits of continuous integration in open-source projects. In: IEEE/ACM. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2016. p. 426–437. Citado na página 17.

HUMBLE, J.; FARLEY, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Boston, MA: Addison-Wesley Professional, 2010. ISBN 978-0321601919. Citado na página 18.

Istio Documentation. *Architecture Overview*. 2024. Acessado em: abril de 2025. Disponível em: <https://istio.io/latest/docs/concepts/what-is-istio/>. Citado na página 38.

Istio Documentation. *Components*. 2024. Acessado em: abril de 2025. Disponível em: <https://istio.io/latest/docs/ops/deployment/architecture/>. Citado 2 vezes nas páginas 7 e 37.

JABBARI, R. et al. What is devops? a systematic mapping study on definitions and practices. In: *Proceedings of the Scientific Workshop Proceedings of XP2016*. [S.l.]: ACM, 2016. p. 1–11. Citado na página 18.

JAMSHIDI, P.; AHMAD, A.; PAHL, C. Cloud migration research: A systematic review. *IEEE Transactions on Cloud Computing*, IEEE, v. 1, n. 2, p. 142–157, 2013. Citado na página 24.

Kubernetes. *Kubernetes Documentation: Components*. 2023. <https://kubernetes.io/docs/concepts/overview/components/>. Acessado em: abril de 2025. Citado 2 vezes nas páginas 7 e 36.

Kubernetes. *Kubernetes Documentation: Concepts*. 2023. <https://kubernetes.io/docs/concepts/>. Acessado em: abril de 2025. Citado 2 vezes nas páginas 35 e 36.

KULYK, A. *What is DevOps and where is it applied?* 2019. <https://shalb.com/blog/what-is-devops-and-where-is-it-applied/>. Acessado em: junho de 2025. Citado 2 vezes nas páginas 7 e 19.

LEPPÄNEN, M. et al. The highways and country roads to continuous deployment. *IEEE Software*, IEEE, v. 32, n. 2, p. 64–72, 2015. Citado na página 18.

LI, W. et al. Performance analysis and optimization of service mesh in cloud-native environments. *IEEE Transactions on Cloud Computing*, v. 9, n. 3, p. 1023–1037, 2021. Citado na página 29.

LóPEZ, M. A.; LOBATO, A. G. P.; DUARTE, O. C. M. B. A performance comparison of cloud computing platforms for big data applications. *IEEE Transactions on Cloud Computing*, IEEE, v. 10, n. 4, p. 2347–2360, 2022. Citado na página 33.

MARKETPLACE, A. *Solo.io Istio Distribution EKS Add-On*. 2025. <https://aws.amazon.com/marketplace/pp/prodview-kvr2wqekzmuhi>. Acessado em: agosto de 2025. Citado na página 54.

MEDEL, V. et al. Modelling performance & resource management in Kubernetes. In: IEEE. *Proceedings of the 9th International Conference on Utility and Cloud Computing*. [S.l.], 2018. p. 257–262. Citado na página 37.

MELL, P.; GRANCE, T. *The NIST definition of cloud computing*. [S.l.], 2011. Disponível em: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf>. Citado na página 23.

MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, Belltown Media, v. 2014, n. 239, p. 2, 2014. Disponível em: <http://dl.acm.org/citation.cfm?id=2600239.2600241>. Citado na página 35.

Microsoft. *Playwright: Fast and reliable end-to-end testing for modern web apps*. 2025. Acesso em: 10 jul. 2025. Disponível em: <https://playwright.dev/>. Citado na página 42.

MORRIS, K. *Infrastructure as Code: Managing Servers in the Cloud*. 2. ed. [S.l.]: O'Reilly Media, 2020. ISBN 978-1492043904. Citado na página 28.

NASCIMENTO, A. S.; SILVA, M. R.; OLIVEIRA, C. T. Service mesh adoption patterns and performance implications: A multi-case study. *Journal of Systems and Software*, v. 196, p. 111598, 2023. Citado na página 30.

NEWMAN, S. *Building Microservices: Designing Fine-Grained Systems*. [S.l.]: O'Reilly Media, Inc., 2015. Citado na página 20.

PAHL, C. et al. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*, IEEE, v. 7, n. 3, p. 677–692, 2019. Citado 2 vezes nas páginas 27 e 29.

PONTES, T. B.; ARTHAUD, D. D. B. Metodologias Ágeis para o desenvolvimento de softwares. *Ciência e Sustentabilidade*, v. 4, n. 2, p. 173–213, mar. 2019. Disponível em: <https://periodicos.ufca.edu.br/ojs/index.php/cienciasustentabilidade/article/view/314>. Citado na página 14.

POPEK, G. J.; GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, ACM, v. 60, n. 1, p. 86–93, 2017. Citado na página 25.

RAHMAN, A. A. U. et al. Synthesizing continuous deployment practices used in software development. In: *2015 Agile Conference*. [S.l.: s.n.], 2015. p. 1–10. Citado na página 14.

RAJ, P. *Terraform Architecture Overview — Structure and Workflow*. 2023. <https://medium.com/@impradeep.techie/terraform-architecture-overview-structure-and-workflow-fdc60697941a>. Acessado em: junho de 2025. Citado 2 vezes nas páginas 7 e 40.

REACT. *Documentação da biblioteca React*. 2025. <https://react.dev/>. Acessado em: agosto de 2025. Citado na página 46.

ROSENBLUM, M.; GARFINKEL, T. Virtual machine monitors: Current technology and future trends. *Computer*, IEEE, v. 38, n. 5, p. 39–47, 2005. Citado na página 24.

SAHOO, J.; MOHAPATRA, S.; LATH, R. Virtualization: A survey on concepts, taxonomy and associated security issues. In: *Proceedings of the 2nd International Conference on Computer and Network Technology*. [S.l.]: IEEE, 2010. p. 222–226. Citado na página 24.

SHAHIN, M.; BABAR, M. A.; ZHU, L. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, IEEE, v. 5, p. 3909–3943, 2017. Citado na página 17.

SHARMA, R.; SARANGDEVOT, K. Enhancing microservices security and observability with istio service mesh. *International Journal of Advanced Computer Science and Applications*, The Science and Information Organization, v. 13, n. 5, p. 324–335, 2022. Citado na página 38.

SILVA, K. R. T. et al. Implementação e orquestração automatizada de clusteres kubernetes com gitops: um estudo de caso. IFPE Belo Jardim, 2023. Citado na página 43.

SOMMERVILLE, I. *Engenharia de Software*. 10. ed. São Paulo: Pearson Education do Brasil, 2019. ISBN 978-8543020563. Citado na página 22.

THOENES, J. Microservices. *IEEE Software*, IEEE, v. 32, n. 1, p. 116–116, 2015. Citado na página 20.

TYPESCRIPT. *Documentação da linguagem TypeScript*. 2025. <https://www.typescriptlang.org/docs/>. Acessado em: agosto de 2025. Citado na página 46.

WICKRAMASINGHE, S. P. *Sneaking into Docker*. 2021. <https://dev.to/sewvandiii/sneaking-into-docker-1cda>. Acessado em: junho de 2025. Citado 2 vezes nas páginas 7 e 34.

YIN, R. *Estudo de Caso - Planejamento e Métodos*. Bookman Editora, 2015. ISBN 9788582602324. Disponível em: <https://books.google.com.br/books?id=EtOyBQAAQBAJ>. Citado na página 15.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, Springer, v. 1, n. 1, p. 7–18, 2010. Citado na página 23.

ZHANG, Q. et al. A comparative study of containers and virtual machines in big data environment. In: *Proceedings of the IEEE 11th International Conference on Cloud Computing*. [S.l.]: IEEE, 2018. p. 178–185. Citado na página 25.

ZHANG, Y. et al. Service mesh for cloud-native applications: A comprehensive analysis of istio architecture and performance. *IEEE Transactions on Cloud Computing*, IEEE, v. 9, n. 4, p. 1589–1602, 2021. Citado na página 37.

Apêndices

APÊNDICE A – Desenvolvimento: Pré-requisitos

A implementação completa deste estudo de caso requer um conjunto específico de recursos, ferramentas e conhecimentos técnicos que devem ser previamente configurados e compreendidos. Esta seção detalha todos os pré-requisitos necessários para reproduzir o ambiente e os processos descritos neste trabalho.

A.1 Contas e Recursos de Cloud

Contas AWS: O estudo de caso utiliza duas contas AWS distintas para garantir isolamento completo entre ambientes:

- **Conta de Desenvolvimento:** Uma conta AWS dedicada ao ambiente de desenvolvimento, testes e validação inicial de recursos
- **Conta de Produção:** Uma conta AWS separada para o ambiente de produção, garantindo segurança e isolamento de recursos críticos

Permissões IAM: Cada conta deve possuir usuários com permissões adequadas para:

- Criação e gerenciamento de clusters EKS
- Administração de repositórios ECR (Elastic Container Registry)
- Configuração de VPCs, *subnets* e grupos de segurança
- Gerenciamento de tabelas DynamoDB
- Configuração de roles IAM e service accounts
- Administração de *Load Balancers* e recursos de rede

A.2 Ferramentas de Desenvolvimento

Linguagens de Programação e Runtimes:

- **Go 1.21+:** Para desenvolvimento e *build* da aplicação *backend*

- **Node.js 18+:** Para desenvolvimento do *frontend* React e execução de testes automatizados
- **TypeScript:** Para tipagem estática no desenvolvimento *frontend*

Ferramentas de Containerização:

- **Docker:** Para construção, teste e execução local de contêineres
- **Docker Compose:** Para orquestração local de múltiplos serviços durante desenvolvimento

Ferramentas de Infraestrutura:

- **Terraform 1.5+:** Para provisionamento de infraestrutura como código
- **AWS CLI 2.x:** Para interação com serviços AWS via linha de comando
- **kubectl:** Para gerenciamento de recursos Kubernetes
- **Helm 3.12+:** Para empacotamento e *deployment* de aplicações Kubernetes
- **istioctl:** Para gerenciamento de recursos do Istio

Ferramentas de Automação e Testes:

- **GitHub Actions:** Habilitado no repositório para execução dos *pipelines* CI/CD
- **Playwright:** Para execução de testes *end-to-end* automatizados
- **Make:** Para padronização de comandos de desenvolvimento e *build*

A.3 Configuração de Repositório

Repositório Git: Um repositório GitHub configurado com:

- Regras de proteção para a *branch* `main`
- *Secrets* configurados para credenciais AWS de ambos os ambientes
- Ambientes configurados para desenvolvimento e produção
- *Workflows* habilitados para GitHub Actions

***Secrets* Necessários:**

- `DEV_AWS_ACCESS_KEY_ID` e `DEV_AWS_SECRET_ACCESS_KEY`
- `PROD_AWS_ACCESS_KEY_ID` e `PROD_AWS_SECRET_ACCESS_KEY`

A.4 Conhecimentos Técnicos Requeridos

Desenvolvimento:

- Programação em Go com *framework* Gin
- Desenvolvimento *frontend* com React e TypeScript
- Conceitos de APIs RESTful e integração *frontend-backend*
- Padrões de arquitetura limpa e separação de responsabilidades

DevOps e Infraestrutura:

- Conceitos fundamentais de containerização com Docker
- Orquestração de contêineres com Kubernetes
- Gerenciamento de service mesh com Istio
- Práticas de Infrastructure as Code com Terraform
- Configuração e operação de *pipelines* CI/CD

Serviços AWS:

- EKS para orquestração de contêineres
- ECR para repositórios de imagens Docker
- DynamoDB para persistência de dados
- VPC, *subnets* e configurações de rede AWS
- IAM roles, policies e service accounts
- *Application Load Balancer* e *Network Load Balancer*

A.5 Configuração do Ambiente Local

Para desenvolvimento e testes locais, é necessário configurar:

- **Ambiente Go:** GOPATH configurado e módulos Go habilitados
- **Ambiente Node.js:** npm ou yarn configurados para gerenciamento de dependências

- **Docker Desktop:** Para execução local de contêineres e Docker Compose
- **AWS CLI:** Configurado com profiles para cada ambiente (dev/prod)
- **kubectl:** Configurado com contextos para *clusters* EKS de cada ambiente