UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

JOSUÉ VILA REAL DE ALMEIDA

CANARY DEPLOYMENT: UMA AVALIAÇÃO QUANTITATIVA PARA AMBIENTES DE PEQUENA ESCALA

Ouro Preto
2026

JOSUÉ VILA REAL DE ALMEIDA

CANARY DEPLOYMENT:
UMA AVALIAÇÃO QUANTITATIVA PARA AMBIENTES DE PEQUENA ESCALA

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Tiago Garcia de Senna Carneiro

Ouro Preto
2026

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

A447c Almeida, Josué Vila Real de.
Canary deployment [manuscrito]: uma avaliação quantitativa para ambientes de pequena escala. / Josué Vila Real de Almeida. - 2026.
103 f.: il.: color., gráf., tab.. + Pseudocódigos.

Orientador: Prof. Dr. Tiago Garcia de Senna Carneiro.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Graduação em Ciência da Computação .

1. Software - Desenvolvimento. 2. Integração (Computação). 3. Engenharia de software. 4. Tecnologia de desempenho. I. Carneiro, Tiago Garcia de Senna. II. Universidade Federal de Ouro Preto. III. Título.

CDU 004.41

Bibliotecário(a) Responsável: Renata Mara de Almeida - CRB-6: 2507

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Josué Vila Real de Almeida

CANARY DEPLOYMENT: UMA AVALIAÇÃO QUANTITATIVA PARA AMBIENTES DE PEQUENA ESCALA

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 29 de Julho de 2026.

Membros da banca

Tiago Garcia de Senna Carneiro (Orientador) - Doutor - Universidade Federal de Ouro Preto
Aline Norberta de Brito (Examinadora) - Doutora - Universidade Federal de Ouro Preto
Igor Muzetti Pereira (Examinador) - Universidade Federal de Ouro Preto

Tiago Garcia de Senna Carneiro, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 29/07/2026.

Documento assinado eletronicamente por **Tiago Garcia de Senna Carneiro**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 29/07/2026, às 18:33, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **1142665** e o código CRC **AE747D26**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

À memória de meu pai, Luciano Cardoso de Almeida, por tudo o que sua presença significou em minha vida e pelos valores e ensinamentos que, por meio de sua lembrança, continuam a orientar meus caminhos e minhas conquistas.

Agradecimentos

À minha família, que é a grande base de tudo o que sou, dedico as minhas primeiras palavras. À minha mãe, Cristina, a minha mais profunda gratidão por ter entregado tanto de si à minha criação. Obrigado por cada esforço, pelos sacrifícios e por todas as portas que a sua dedicação me abriu, agradecendo, principalmente, por acreditar no meu potencial mesmo naqueles momentos em que eu mesmo duvidava. Ao meu pai, Luciano, que doou sua força, seu tempo e, muitas vezes, a própria saúde para que eu tivesse oportunidades e pudesse construir o meu próprio caminho. Seus ensinamentos, seus valores e o seu exemplo continuam vivos em mim, guiando cada uma das minhas escolhas, e, embora não esteja fisicamente aqui para celebrar este momento, esta vitória também é inteiramente sua. E à minha irmã, Roberta, o meu obrigado pelas risadas, pela companhia e pelo incentivo constante ao longo de todo este ciclo. Você fez com que os momentos difíceis fossem mais leves e as conquistas, ainda mais especiais.

À minha namorada, Catarina, dedico um agradecimento especial por todo o amor, paciência e parceria com que abraçou essa jornada ao meu lado. Nos dias em que o cansaço e as incertezas pesaram, ter você ali, celebrando cada pequeno passo e me lembrando do que sou capaz, trouxe a leveza e a força de que eu precisava para continuar. Mais do que me apoiar neste ciclo, você me inspira diariamente a ser a minha melhor versão.

Esta caminhada não seria a mesma sem a presença dos grandes amigos que me apoiaram ao longo do caminho. Aos meus companheiros de casa, Carlos, Lucas e Robb, a minha enorme gratidão. Vocês transformaram o nosso convívio em um verdadeiro porto seguro de apoio e alegria, e a companhia e as incontáveis risadas de vocês tornaram o processo de mudança e adaptação muito mais divertido, fazendo com que essa trajetória se tornasse leve e inesquecível. Aos amigos que a Universidade Federal de Ouro Preto me deu e que levarei para a vida, Thiago Cecote, Luísa e Matheus, obrigado pelo companheirismo e por cada instante compartilhado durante a graduação. As conversas, as jogatinas e os desafios que superamos juntos são parte fundamental dessa história e deixaram essa fase muito mais marcante. Da mesma forma, aos grandes amigos que fiz no Efí Bank, João Lucas, Beatriz, Júlia, Kokinha e Hugo, meu muito obrigado. Agradeço não só pelos ensinamentos e pela troca constante de experiências, mas principalmente pela amizade e pelas risadas, pois nossas conversas divertidas na hora do almoço com certeza tornaram os meus dias muito mais leves e alegres.

Por fim, no âmbito acadêmico, expresso minha gratidão ao meu orientador, Thiago, pela confiança e pelos preciosos ensinamentos. Suas orientações e contribuições não apenas foram essenciais para o desenvolvimento deste trabalho, como também para o meu amadurecimento acadêmico. A todos que cruzaram o meu caminho e que, de alguma forma, contribuíram para que esta conquista se tornasse realidade, deixo registrada a minha eterna gratidão.

“Não esquecer que por enquanto é tempo de morangos.”

— Lispector (1998)

Resumo

O cenário contemporâneo de desenvolvimento de *software* impõe a necessidade de entregas rápidas e seguras, criando desafios significativos para ambientes de pequeno porte, que operam com recursos limitados. Este trabalho apresenta um estudo quase-experimental que investiga o impacto da implementação de uma estratégia de *canary deployment* com *rollback* automatizado no desempenho do processo de entrega de software. O objetivo central consiste em avaliar, por meio das quatro métricas DORA (*lead time for changes*, *deployment frequency*, *time to restore service* e *change failure rate*), como a adoção de uma abordagem de implantação progressiva influencia a velocidade e a estabilidade do ciclo de entrega de *software*. A relevância deste estudo reside na mitigação dos riscos de interrupção de serviço e de degradação da experiência do usuário, contribuindo para suprir uma lacuna na literatura acerca da viabilidade técnica dessas práticas em contextos de pequeno porte. Para validar a proposta, projetou-se e implementou-se um *pipeline* de implantação modelo, sobre o qual se conduziu-se experimentos controlados para comparar o desempenho do processo com e sem a estratégia proposta. Os resultados visam subsidiar equipes pequenas na tomada de decisão, demonstrando que é possível adotar práticas de implantação resilientes e eficientes sem incorrer na complexidade associada a grandes infraestruturas.

Palavras-chave: DevOps. Entrega Contínua. Implantação Contínua. Canary Deployment. Métricas DORA. Engenharia de Software.

Abstract

The contemporary software development landscape imposes a need for fast and secure delivery, creating significant challenges for small-scale environments operating with limited resources. This work presents a quasi-experimental study investigating the impact of implementing a *canary deployment* strategy with automated *rollback* on software delivery process performance. The central objective is to evaluate, using the four DORA metrics (*lead time for changes*, *deployment frequency*, *time to restore service*, and *change failure rate*), how adopting a progressive deployment approach influences the speed and stability of the software delivery cycle. The relevance of this study lies in mitigating the risks of service disruption and user experience degradation, contributing to filling a gap in the literature regarding the technical feasibility of these practices in small-scale contexts. To validate the proposal, a model deployment *pipeline* was designed and implemented, upon which controlled experiments were conducted to compare process performance with and without the proposed strategy. The results aim to support smaller teams in decision-making, demonstrating that it is possible to adopt resilient and efficient deployment practices without incurring the complexity associated with large infrastructures.

Keywords: DevOps. Continuous Delivery. Continuous Deployment. Canary Deployment. DORA Metrics. Software Engineering.

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Abreviaturas e Siglas

API	*Application Programming Interface*
BDD	*Behavior Driven Development* (Desenvolvimento Guiado por Comportamento)
CaC	Configuração como Código (*Configuration as Code*)
CD	Entrega Contínua (*Continuous Delivery*)
CFR	Taxa de Falha nas Mudanças (*Change Failure Rate*)
CI	Integração Contínua (*Continuous Integration*)
CI/CD	Integração Contínua e Entrega Contínua (*Continuous Integration and Continuous Delivery*)
CPU	Unidade Central de Processamento (*Central Processing Unit*)
CSS	Cascading Style Sheets
CVS	Sistema de Versões Concorrentes (*Concurrent Versions System*)
CVSS	*Common Vulnerability Scoring System*
DAST	Teste Dinâmico de Segurança de Aplicação (*Dynamic Application Security Testing*)
DECOM	Departamento de Computação
DevOps	DevOps (*Development and Operations*)
DF	Deployment Frequency (Frequência de Implantação)
DORA	*DevOps Research and Assessment*
DSR	*Design Science Research*
E2E	Ponta a Ponta (*end-to-end*)
FK	Foreign Key (Chave Estrangeira)
GCP	*Google Cloud Platform*
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IA	Inteligência Artificial (*Artificial Intelligence*)
IaC	Infraestrutura como Código (*Infrastructure as Code*)
ICEB	Instituto de Ciências Exatas e Biológicas
INPE	Instituto Nacional de Pesquisas Espaciais
IP	Internet Protocol
JSON	JavaScript Object Notation
LTFC	Lead Time for Changes (Tempo de Entrega para Mudanças)
ML	Aprendizado de Máquina (*Machine Learning*)
MLOps	MLOps (*Machine Learning Operations*)
MR	*Merge Request*

MTTR	Tempo para Restaurar o Serviço (*Mean Time To Restore*)
OCI	Open Container Initiative
PK	Primary Key (Chave Primária)
PME	Pequena e Média Empresa (*Small and Medium-sized Enterprise*)
QA	Garantia de Qualidade (*Quality Assurance*)
RAM	Memória de Acesso Aleatório (*Random Access Memory*)
ROI	Retorno sobre o Investimento (*Return on Investment*)
RPO	Objetivo de Ponto de Recuperação (*Recovery Point Objective*)
RTO	Objetivo de Tempo de Recuperação (*Recovery Time Objective*)
SA	Conta de Serviço (*Service Account*)
SDK	Kit de Desenvolvimento de Software (*Software Development Kit*)
SHA	*Secure Hash Algorithm*
SIG	Sistema de Informação Geográfica
SLI	Indicador de Nível de Serviço (*Service Level Indicator*)
SMTP	Simple Mail Transfer Protocol
SRE	Engenharia de Confiabilidade de Sites (*Site Reliability Engineering*)
SRM	*Software Release Manager*
SSH	*Secure Shell*
SVN	Subversion
TBD	Desenvolvimento Baseado em Tronco (*Trunk-Based Development*)
TerraLAB	Laboratório de Pesquisa e Capacitação em Software
UFOP	Universidade Federal de Ouro Preto
UI	Interface de Usuário (*User Interface*)
UQ	Unique Constraint (Restrição de Unicidade)
URL	Uniform Resource Locator (Localizador Uniforme de Recursos)
UTC	Coordinated Universal Time (Tempo Universal Coordenado)
VCS	Sistema de Controle de Versão (*Version Control System*)
VM	Máquina Virtual (*Virtual Machine*)
YAML	YAML Ain't Markup Language

Sumário

1 Introdução

A crescente digitalização de serviços tornou a entrega de *software* de alta qualidade um diferencial competitivo crucial. Para alcançar a velocidade e a confiabilidade que o mercado exige, as organizações têm adotado a cultura DevOps (*Development and Operations*, DevOps), integrando equipes de desenvolvimento e operações para otimizar o ciclo de entrega de *software* (KHAN et al., 2022; JHA et al., 2023). Nesse ecossistema, a automação por meio de *pipelines* de Integração Contínua (*Continuous Integration*, CI) e Entrega Contínua (*Continuous Delivery*, CD) consolidou-se como um pilar fundamental. Conforme sistematizado por Humble e Farley (2010), o *pipeline* de implantação constitui a espinha dorsal da entrega contínua, automatizando o fluxo desde o controle de versão até a produção.

No entanto, a etapa de implantação em produção permanece como uma das fases mais críticas e de maior risco no ciclo de desenvolvimento de software. A Engenharia de Confiabilidade de Sites (*Site Reliability Engineering*, SRE) aponta que a maioria dos incidentes em produção é desencadeada por alterações de código ou configuração (BEYER et al., 2016; BEYER et al., 2018), como evidenciado pelo incidente global de julho de 2024 associado a uma atualização de configuração distribuída pelo CrowdStrike Falcon, que resultou em falhas sistêmicas em sistemas Windows, afetando cerca de 8,5 milhões de dispositivos e gerando impactos econômicos e sociais amplos em serviços críticos (CrowdStrike, 2024; WESTON, 2024). Nesse contexto, estratégias tradicionais de entrega, como o *Big Bang Deployment*, concentram o risco em um único momento, aumentando a probabilidade de interrupções do serviço e a degradação da experiência do usuário (HUMBLE; FARLEY, 2010).

Para mitigar esses riscos, a indústria desenvolveu estratégias de implantação progressiva. Abordagens como o *Blue-Green Deployment*, que alterna o tráfego entre dois ambientes idênticos, oferecem maior segurança e possibilitam retorno imediato à versão anterior; contudo, impõem custos elevados de infraestrutura ao exigir a duplicação de recursos (HUMBLE; FARLEY, 2010; RAMASWAMY, 2024). Como alternativa de menor custo, o *Rolling Deployment* atualiza as instâncias gradualmente, evitando a duplicação de infraestrutura, embora dificulte a reversão instantânea, uma vez que, diante de falhas, a recuperação requer a reimplantação da versão anterior nos nós já atualizados, prolongando o tempo de recuperação (HUMBLE; FARLEY, 2010). Nesse contexto, a estratégia de *Canary Deployment* destaca-se por combinar a eficiência no uso de recursos do *Rolling Deployment* com a segurança da validação por amostragem, pois, ao direcionar a nova versão a um subconjunto restrito de usuários por um período limitado, permite validar a aplicação em produção com risco controlado (BEYER et al., 2016). Entretanto, a ausência de automação nessa etapa transforma a atividade em um procedimento manual moroso e sujeito a variações, o que, segundo Humble e Farley (2010), compromete a repetibilidade e demanda esforço excessivo da equipe.

Essa dicotomia apresenta um desafio particular para ambientes de pequena escala, como *startups* e pequenas empresas. Essas organizações frequentemente operam com restrições orçamentárias que inviabilizam a redundância de infraestrutura do *Blue-Green Deployment*, ao mesmo tempo em que carecem de grandes equipes de operações para monitorar manualmente lançamentos do tipo *Canary Deployment* (TUAPE et al., 2021; KHAN et al., 2022). Nesse cenário, a automação da reversão (isto é, *rollback*) atua como um mecanismo compensatório para a escassez de recursos humanos identificada por Tuape et al. (2021). Ao eliminar o trabalho manual repetitivo e sem valor estratégico, a automação permite que equipes enxutas operem sistemas complexos com a mesma confiabilidade de grandes corporações, desacoplando o crescimento do sistema do tamanho da equipe (BEYER et al., 2016).

Para avaliar a eficácia dessa abordagem, foi necessário ir além da verificação funcional da implementação e analisar seu impacto nos resultados da entrega de *software*. Para esse fim, foram adotadas as métricas *DevOps Research and Assessment* (DORA), a saber: Lead Time for Changes (Tempo de Entrega para Mudanças, LTFC), Deployment Frequency (Frequência de Implantação, DF), Tempo para Restaurar o Serviço (*Mean Time To Restore*, MTTR) e Taxa de Falha nas Mudanças (*Change Failure Rate*, CFR) (FORSGREN; HUMBLE; KIM, 2018). Essas métricas, validadas cientificamente ao longo de uma década de pesquisas, consolidaram-se como referência para mensurar o desempenho de *pipelines* de implantação (DEBELLIS et al., 2024). Diferentemente de métricas tradicionais focadas em volume, as métricas DORA permitem analisar a eficácia global do sistema, evidenciando que velocidade e estabilidade não configuram um *trade-off*, mas capacidades que se reforçam mutuamente (FORSGREN; HUMBLE; KIM, 2018). Neste trabalho, essas métricas foram essenciais para quantificar como a estratégia de *Canary Deployment* influenciou a agilidade da entrega e como a reversão automatizada contribuiu para mitigar instabilidades, permitindo avaliar o alinhamento de um ambiente de pequena escala aos padrões de alto desempenho identificados na literatura (DEBELLIS et al., 2024).

Apesar da ampla literatura sobre DevOps em grandes empresas, existe uma lacuna na avaliação quantitativa dessas estratégias avançadas em contextos de recursos limitados. Este trabalho investiga a interseção entre a estratégia de *Canary Deployment*, a automação de reversão e a avaliação por meio de métricas DORA. A pergunta central que orientou esta pesquisa foi:

> *Qual é o impacto quantitativo da implementação de Canary Deployment com rollback automatizado no desempenho da entrega de software, medido pelas métricas DORA, em um ambiente de pequena escala?*

1.1 Justificativa

A relevância deste estudo fundamenta-se na necessidade crítica de democratizar práticas de alta confiabilidade para cenários de recursos limitados. A argumentação estrutura-se em três pilares interconectados, que dialogam com a literatura contemporânea de engenharia de *software*:

(i) a sustentabilidade cognitiva e econômica de equipes enxutas; (ii) a substituição de custos de capital por automação inteligente; e (iii) a validação empírica dessa arquitetura por meio de métricas padronizadas.

Do ponto de vista econômico e estratégico, a capacidade de entregar *software* com estabilidade tornou-se um requisito de sobrevivência. Em pequenas organizações, nas quais o impacto de falhas é desproporcionalmente severo, a "ansiedade de implantação" descrita por Humble e Farley (2010) frequentemente resulta em processos manuais de verificação que reduzem a produtividade. Pesquisas seminais de Forsgren, Humble e Kim (2018) não apenas correlacionam o alto desempenho de entrega ao sucesso comercial, como também indicam que a automação robusta atua preventivamente contra o *burnout* e o estresse operacional das equipes envolvidas. Nesse contexto, o *rollback* automatizado adotado neste estudo atua na redução do trabalho manual e repetitivo, cujo esforço tende a escalar linearmente (BEYER et al., 2016). Ademais, configura-se como um mecanismo de proteção do capital humano, permitindo que equipes reduzidas operem com a confiança psicológica de grandes times de SRE, concentrando-se em atividades de engenharia, em vez de em recuperação de desastres (BEYER et al., 2016).

No âmbito técnico-operacional, este trabalho aborda o dilema dos custos de infraestrutura. Embora estratégias como o *Blue-Green Deployment* sejam eficazes na mitigação de riscos, elas exigem a duplicação de ambientes de produção, gerando um *custo adicional* de infraestrutura e uma complexidade de roteamento que, conforme analisado por Tuape et al. (2021), pode ser financeiramente proibitiva para *startups* e Pequenas e Médias Empresas (*Small and Medium-sized Enterprises*, PMEs). A presente pesquisa sustenta a tese de que a segurança obtida tradicionalmente pela *redundância espacial* pode ser substituída, em escalas menores, pela detecção e reversão imediatas. Com a adoção do *Canary Deployment* com *rollback* automatizado, buscou-se avaliar se é possível atingir níveis de disponibilidade competitivos, substituindo o investimento massivo em recursos ociosos por uma automação de *software* capaz de conter falhas em segundos.

Por fim, a justificativa metodológica sustenta-se na validação rigorosa desta arquitetura frente ao dilema de *assurance* (isto é, garantia) identificado por Hilton et al. (2017). O autor revela que equipes de desenvolvimento frequentemente operam sob um *trade-off* crítico entre *velocidade* e *garantia*: a busca por ciclos de *feedback* rápidos tende a sacrificar a profundidade da verificação, enquanto uma garantia de qualidade robusta pode inviabilizar a agilidade necessária ao mercado (HILTON et al., 2017). Para avaliar a superação dessa dicotomia, este estudo instrumentou o processo com as métricas do DORA. O objetivo foi fornecer evidências quantitativas de que a automação do ciclo de *feedback* é capaz de manter a estabilidade sem sacrificar a agilidade, corroborando as descobertas de Forsgren, Humble e Kim (2018), mas oferecendo um modelo de referência replicável para maximizar o Retorno sobre o Investimento (*Return on Investment*, ROI) do *pipeline* de implantação, sem os orçamentos de infraestrutura das grandes corporações tecnológicas.

1.2 Objetivos

O objetivo geral deste trabalho foi analisar o impacto da adoção de estratégias de *Canary Deployment* com *rollback* automatizado na eficiência do ciclo de entrega de *software*. O estudo investigou quantitativamente as variações de velocidade e estabilidade em ambientes de pequena escala, utilizando as métricas DORA como indicadores de desempenho. Para alcançar este propósito, foram definidos os seguintes objetivos específicos:

1. ***Estabelecer uma linha de base de desempenho***, caracterizando o cenário atual de entrega de *software* por meio do mapeamento e análise crítica do *pipeline* de implantação vigente;
2. ***Desenvolver uma arquitetura experimental de implantação*** integrada à estratégia de *Canary Deployment*, com mecanismos de observabilidade e rotinas de *rollback* automatizado, instrumentada para coleta automatizada das métricas DORA;
3. ***Validar a eficácia dos mecanismos de recuperação***, submetendo o ambiente a experimentos controlados de injeção de falhas para verificar o acionamento e a precisão do *rollback* automatizado;
4. ***Comparar quantitativamente os cenários***, confrontando os indicadores de desempenho da linha de base com os resultados obtidos após a automação, utilizando as métricas DORA para determinar a viabilidade técnica da abordagem implementada.

1.3 Organização do Trabalho

O restante desta monografia está estruturado em quatro capítulos, organizados de forma a guiar o leitor desde a fundamentação teórica até a discussão dos resultados obtidos. O conteúdo de cada capítulo é descrito a seguir:

Capítulo 2 – Revisão Bibliográfica:
Apresenta a fundamentação teórica necessária, abordando a terminologia de *pipelines* de implantação, a cultura DevOps, a Pirâmide de Testes e as estratégias avançadas de implantação. Define as métricas DORA utilizadas para mensuração de desempenho e analisa trabalhos relacionados que contextualizam o estado da arte.

Capítulo 3 – Materiais e Métodos:
Classifica a pesquisa e descreve a metodologia aplicada. Caracteriza o contexto institucional e o objeto de análise. Detalha o procedimento de caracterização do *pipeline* de implantação vigente por meio de análise documental, entrevistas e medição de cobertura de testes. Define os critérios metodológicos para *gates* de qualidade, a política de *rollback* automatizado e a estratégia de *Canary Deployment*. Explicita premissas e restrições do estudo.

Capítulo 4 – Resultados e Discussão:

Apresenta o mapeamento sistemático do *pipeline* de implantação vigente, identificando seus componentes, fluxos operacionais e gargalos críticos. Descreve a evolução implementada no *pipeline* de implantação, detalhando os mecanismos de *Canary Deployment* com *rollback* automatizado, a instrumentação para coleta das métricas DORA, a avaliação experimental e a comparação entre a linha de base e o *pipeline* implementado.

Capítulo 5 – Considerações Finais:

Sintetiza as conclusões alcançadas a partir do diagnóstico do fluxo de implantação vigente, da implementação realizada e da avaliação experimental. Avalia o atendimento aos objetivos específicos, destacando os resultados obtidos, as implicações práticas, as limitações metodológicas e as oportunidades de trabalhos futuros.

2 Revisão Bibliográfica

A presente seção constrói o alicerce teórico desta pesquisa, delimitando o estado da arte e explicitando os fundamentos que sustentam a investigação desenvolvida. A exposição organiza-se em duas frentes complementares. A seção 2.1 sistematiza o arcabouço conceitual, definindo a terminologia relativa a *pipelines* de implantação, descrevendo os princípios da cultura DevOps, explorando a Pirâmide de Testes e as estratégias avançadas de implantação, e apresentando as métricas de desempenho essenciais para a compreensão dos capítulos subsequentes. Em seguida, a seção 2.2 reconstrói a trajetória evolutiva da área, mapeando a evolução do campo desde as contribuições seminais até as pesquisas contemporâneas que relacionam estratégias de implantação à eficiência em Integração Contínua e Entrega Contínua (*Continuous Integration and Continuous Delivery*, CI/CD), situando este trabalho no contexto científico atual.

2.1 Fundamentação Teórica

Esta seção detalha os pilares técnicos e conceituais necessários à análise do processo de entrega de *software* investigado neste estudo. A discussão inicia-se na subseção 2.1.1, que estabelece o vocabulário técnico relativo aos componentes presentes em cada etapa de um fluxo de CI/CD e descreve cada um deles. Na sequência, a subseção 2.1.2 contextualiza tais elementos sob a ótica da cultura DevOps e da colaboração entre equipes. Em seguida, a subseção 2.1.3 explora as garantias de qualidade por meio da Pirâmide de Testes, descrevendo os tipos de teste que podem ser realizados ao longo de um *pipeline* de implantação. Posteriormente, a subseção 2.1.4 descreve as estratégias avançadas de implantação. Por fim, a subseção 2.1.5 fundamenta o uso das métricas DORA para a mensuração objetiva de desempenho e estabilidade.

2.1.1 Terminologia e Conceitos Fundamentais

Para assegurar a precisão técnica desta pesquisa, torna-se necessário explicitar a terminologia adotada para descrever a infraestrutura de entrega de *software* e seus principais elementos constitutivos. Em particular, esta seção delimita os conceitos que estruturam o fluxo automatizado de mudanças e estabelece a padronização empregada ao longo do trabalho.

2.1.1.1 Controle de Versão

A automação de CI/CD pressupõe a existência de um repositório central capaz de registrar, organizar e auditar a evolução do sistema (FOWLER; FOEMMEL, 2006). Nesse contexto, o Sistema de Controle de Versão (*Version Control System*, VCS) é o mecanismo que materializa a governança técnica sobre mudanças, funcionando como ponto de convergência para o trabalho

colaborativo e como base para rastreabilidade e reprodutibilidade (FOWLER; FOEMMEL, 2006; HUMBLE; FARLEY, 2010). Na literatura sobre entrega contínua, o repositório de versões é descrito como a *única fonte de verdade* do sistema, concentrando não apenas o código da aplicação, mas também *scripts* de construção, descrições de infraestrutura e artefatos de configuração necessários à implantação (HUMBLE; FARLEY, 2010; LEITE et al., 2019).

Assim, qualquer elemento que influencie a construção ou o comportamento do sistema em tempo de execução deve ser versionado (FOWLER; FOEMMEL, 2006). Essa prática viabiliza a rastreabilidade entre as mudanças de código, configurações e versões implantadas, além de permitir a reprodutibilidade de *builds* e a auditoria do processo de liberação (HUMBLE; FARLEY, 2010). Em *pipelines* de implantação maduros, isso inclui não apenas código-fonte, mas também especificações de testes automatizados, manifestos de infraestrutura e *scripts* de automação operacional, aproximando-se das ideias de Infraestrutura como Código (*Infrastructure as Code*, IaC) e Configuração como Código (*Configuration as Code*, CaC) (KIM et al., 2016).

No que tange às ferramentas que operacionalizam esses conceitos, a literatura destaca soluções consolidadas como Git, Subversion (SVN), Mercurial e Sistema de Versões Concorrentes (*Concurrent Versions System*, CVS) (ZOLKIFLI; NGAH; DERAMAN, 2018). O uso dessas ferramentas é fundamental para viabilizar a colaboração, permitindo que múltiplos desenvolvedores trabalhem simultaneamente por meio de mecanismos de registro de histórico e de fusão de alterações, garantindo a integridade dos artefatos ao longo do ciclo de vida do *software* (ZOLKIFLI; NGAH; DERAMAN, 2018).

Neste trabalho, o VCS desempenha três papéis estruturantes: (i) atua como ponto de acionamento do *pipeline* de integração contínua, de modo que cada *commit* dispare o fluxo automatizado; (ii) serve como mecanismo de coordenação entre as equipes, consolidando o estado canônico do sistema; e (iii) provê suporte a estratégias de recuperação e *rollback* em caso de falha.

2.1.1.2 Integração e Entrega Contínua (CI/CD)

Uma vez estabelecido o papel do VCS como fonte de mudança e acionador do fluxo automatizado, é necessário distinguir as práticas que organizam e operacionalizam esse fluxo. Embora o termo CI/CD seja usado de forma agregada, ele abrange níveis diferentes de automação e etapas para liberação, os quais determinam como uma alteração progride do *commit* até a disponibilização ao usuário final (HUMBLE; FARLEY, 2010).

A Integração Contínua é uma prática de desenvolvimento na qual os desenvolvedores integram seu trabalho frequentemente, pelo menos uma vez ao dia, a um repositório central compartilhado (FOWLER; FOEMMEL, 2006). Conforme Fowler e Foemmel (2006), cada integração é verificada por um *build* automatizado e por testes unitários e funcionais, os quais são apresentados na subseção 2.1.3. O objetivo principal da CI é detectar problemas de integração o mais cedo possível, evitando o que os autores denominam *integration hell*, um cenário em que a

fusão de diferentes partes do código, desenvolvidas em paralelo por um longo período, torna-se uma tarefa complexa e demorada, resultando em um emaranhado de erros de difícil resolução (FOWLER; FOEMMEL, 2006). A integração contínua, quando bem-sucedida, deve produzir um artefato de *software* testado e pronto para ser entregue aos estágios seguintes do processo de automação (FOWLER; FOEMMEL, 2006).

A Entrega Contínua é a extensão natural da CI. Segundo Humble e Farley (2010), a CD é a prática que garante que cada alteração que passa por todos os estágios automatizados da integração contínua possa ser liberada para os clientes mediante uma ação controlada e deliberada. O artefato gerado por essa prática passa por estágios adicionais de teste, como os de aceitação e desempenho, em ambientes que simulam o ambiente de produção (HUMBLE; FARLEY, 2010). O objetivo é tornar os *releases* eventos rotineiros e de baixo risco, que podem ocorrer a qualquer momento, sob demanda, resultando em um candidato a *release* validado e pronto para implantação (HUMBLE; FARLEY, 2010).

2.1.1.3 Pipelines de Integração Contínua e de Implantação

Definidos os níveis de maturidade associados a CI e CD, o próximo passo é caracterizar o artefato operacional que os implementa: o *pipeline*. Nesta pesquisa, o *pipeline* é tratado como a representação explícita do encadeamento de estágios automatizados que aplica validações progressivas às mudanças, estabelecendo critérios objetivos de promoção entre etapas (HUMBLE; FARLEY, 2010). Assim, esta subseção descreve o conceito de *deployment pipeline* e estabelece a distinção terminológica entre o *pipeline* de integração contínua e o *pipeline* de implantação (HUMBLE; FARLEY, 2010; FOWLER; FOEMMEL, 2006).

No modelo proposto por Humble e Farley (2010), o fluxo denominado *deployment pipeline* é usualmente representado como uma cadeia de estágios de *build*, testes e aprovação, conectados por pontos de decisão que determinam se a mudança pode ou não avançar para o próximo estágio. Neste trabalho, adota-se a padronização terminológica ilustrada na Figura 2.1 para as etapas do fluxo em questão.

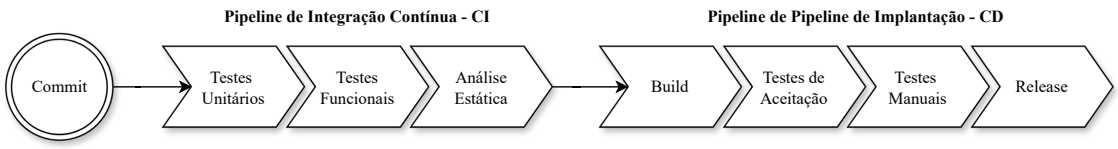

Figura 2.1 – Delimitação entre Pipeline de CI e Pipeline de CD

Fonte: Elaborada pelo autor, baseada em (HUMBLE; FARLEY, 2010).

- ***Pipeline de Integração Contínua* (*Continuous Integration Pipeline*):** corresponde ao primeiro segmento do *pipeline* de implantação, iniciado a cada novo *commit* (FOWLER; FOEMMEL, 2006). Engloba, de forma automatizada, atividades como compilação, execução de testes unitários e verificações estáticas de qualidade. Seu objetivo principal é

fornecer *feedback* rápido sobre a integridade do código e produzir um artefato de *software* testado, armazenado em um repositório central e pronto para ser promovido a estágios posteriores (FOWLER; FOEMMEL, 2006; HUMBLE; FARLEY, 2010). Do ponto de vista instrumental, esta fase é suportada por servidores de automação tradicionais, como Jenkins e TeamCity, ou ferramentas de construção como Maven e Make, embora, mais recentemente, serviços baseados em arquivos YAML Ain't Markup Language (YAML) (por exemplo, GitHub Actions, Travis CI e GitLab CI) tenham se tornado o padrão para automatizar a configuração desses ambientes (GHALEB; ABDULJALIL; HASSAN, 2025; KINSMAN et al., 2021; HUMBLE; FARLEY, 2010).

- ***Pipeline de Implantação* (*Deployment Pipeline*):** representa o processo mais amplo de entrega, que *contém* o *pipeline* de integração contínua como etapa inicial e o estende com estágios adicionais de validação funcional, não funcional e orquestração da implantação em diferentes ambientes (HUMBLE; FARLEY, 2010; KIM et al., 2016). O *pipeline* de implantação termina quando um candidato a *release* é promovido ao ambiente de produção (HUMBLE; FARLEY, 2010). Para suportar esse fluxo em arquiteturas complexas em nuvem, soluções como Kubernetes, Istio e ferramentas de automação são frequentemente empregadas na orquestração da infraestrutura e no suporte a estratégias avançadas de *release*, como o Canary Deployment (MALHOTRA et al., 2024).

Em termos de conjunto de atividades, o *pipeline* de integração contínua é, portanto, um *subconjunto estrito* do *pipeline* de implantação: toda mudança que alcança os estágios finais de implantação necessariamente passou pela CI, mas nem toda mudança integrada precisa, de fato, ser promovida à produção (HUMBLE; FARLEY, 2010; KIM et al., 2016).

2.1.1.4 Ambientes de Implantação

A progressão de uma mudança ao longo do *pipeline* de implantação não ocorre em um único contexto de execução, mas por meio de uma cadeia de ambientes com níveis crescentes de realismo e criticidade (HUMBLE; FARLEY, 2010). A finalidade dessa segmentação é reduzir o risco, permitindo que o artefato seja validado incrementalmente sob condições controladas antes de alcançar usuários reais (FOWLER; FOEMMEL, 2006). Dessa forma, esta subseção define os principais ambientes de implantação considerados no trabalho.

As mudanças validadas pelo *pipeline* de implantação devem progredir por uma sequência de *ambientes de implantação* antes de atingirem a liberação final (HUMBLE; FARLEY, 2010). Humble e Farley (2010) também caracteriza esses ambientes como configurações lógicas ou físicas isoladas, nas quais o *software* é testado sob diferentes cenários e níveis de risco aceitável. Para Fowler e Foemmel (2006), é crucial que tais ambientes se aproximem o máximo possível do ambiente de produção. Nessa perspectiva, quanto maior a paridade entre os ambientes, menor a probabilidade de falhas emergirem apenas após a implantação final (HUMBLE; FARLEY, 2010;

FOWLER; FOEMMEL, 2006). De forma conceitual, os ambientes considerados neste trabalho podem ser organizados da seguinte maneira:

- ***Desenvolvimento* (*Development*):** espaço de trabalho controlado pelo desenvolvedor, onde são realizados experimentos locais, execução de testes unitários e verificação rápida antes da integração. Pode corresponder a ambientes locais ou a instâncias efêmeras provisionadas sob demanda (KIM et al., 2016).
- ***Homologação* (*Testing/Staging*):** ambiente de testes idealmente idêntico à produção para o qual o candidato de *release* é implantado pelo *pipeline* de implantação nele executam-se testes automatizados mais abrangentes para elevar a confiança antes da promoção a produção. (HUMBLE; FARLEY, 2010; KIM et al., 2016)
- ***Produção* (*Production*):** ambiente em que o sistema efetivamente entrega valor a usuários reais e em que as métricas de desempenho de entrega e de confiabilidade são observadas. Nos modelos de *pipeline* de implantação propostos por Humble e Farley (2010), ele constitui o estágio final da implantação, enquanto pesquisas sobre DevOps enfatizam a produção como principal fonte de métricas operacionais (por exemplo, disponibilidade, desempenho, falhas e recuperação) para avaliar o impacto das estratégias de implantação (LEITE et al., 2019). Do ponto de vista deste trabalho, é o destino final do *pipeline* de implantação e o principal ponto de observação para avaliar o impacto das estratégias de implantação estudadas.

2.1.1.5 *Deployment*, *rollback* e recuperação

Após a definição do *pipeline* de implantação e de seus ambientes-alvo, torna-se necessário caracterizar três conceitos operacionais que descrevem como uma mudança chega à produção e como o serviço é mantido ou restabelecido quando essa mudança falha: *deployment*, *rollback* e recuperação. Em particular, a etapa final do *pipeline* de implantação não se resume a "copiar" uma nova versão para os servidores; ela envolve decisões de exposição ao usuário, mecanismos de mitigação de risco e procedimentos para restabelecer o nível de serviço esperado quando incidentes ocorrem (HUMBLE; FARLEY, 2010; KIM et al., 2016).

No âmbito desta pesquisa, o *deployment* (doravante denominado implantação) é entendido como o conjunto de atividades técnicas que instala e prepara uma versão específica da aplicação e suas dependências para execução em um ambiente-alvo (HUMBLE; FARLEY, 2010). Essa preparação inclui, tipicamente, a disponibilização de artefatos, a aplicação de configurações e o provisionamento dos recursos de infraestrutura necessários, de modo que a versão implantada esteja operacionalmente pronta para receber carga (HUMBLE; FARLEY, 2010; KIM et al., 2016). Entretanto, implantar não implica necessariamente tornar a versão visível para os usuários finais. A literatura enfatiza a distinção entre implantar e *release*: enquanto a implantação posiciona o sistema em um novo estado executável, o *release* corresponde ao ato de expor a mudança ao

tráfego real, o que pode ser realizado, por exemplo, por meio de comutação no balanceador de carga ou por habilitação controlada de funcionalidades (HUMBLE; FARLEY, 2010; KIM et al., 2016). Essa separação é particularmente relevante porque permite reduzir o risco operacional: uma organização pode implantar com frequência e, ainda assim, controlar quando e como a mudança será efetivamente consumida por usuários (KIM et al., 2016).

Mesmo com automação e testes, toda implantação em produção introduz a possibilidade de degradações de desempenho e disponibilidade. Nesse contexto, o *rollback* designa o mecanismo de reversão para um estado previamente estável quando a versão recém-implantada se mostra inadequada para a operação (HUMBLE; FARLEY, 2010). Na prática, a reversão pode ser implementada como uma restauração do direcionamento de tráfego para a versão anterior ou como a reimplantação de uma versão previamente validada (HUMBLE; FARLEY, 2010; KIM et al., 2016).

Por fim, a recuperação é um conceito mais amplo do que o *rollback*. Enquanto o *rollback* busca reverter uma mudança específica, a recuperação abrange o conjunto de ações para restabelecer níveis aceitáveis de serviço após um incidente, o que pode envolver a reconstrução de ambientes, a restauração de dados e procedimentos de resposta a incidentes (HUMBLE; FARLEY, 2010; KIM et al., 2016). Para orientar essa capacidade, práticas de continuidade de serviço utilizam objetivos explícitos como Objetivo de Ponto de Recuperação (*Recovery Point Objective*, RPO), que limita a perda aceitável de dados, e Objetivo de Tempo de Recuperação (*Recovery Time Objective*, RTO), que estabelece o tempo máximo para restaurar o serviço; tais objetivos implicam não apenas políticas de *backup*, mas também a capacidade de reproduzir versões corretas da aplicação, infraestrutura e configurações de forma controlada e testada (HUMBLE; FARLEY, 2010).

2.1.2 DevOps e a Cultura de Colaboração

O termo DevOps representa uma mudança cultural e profissional que visa unificar o desenvolvimento de *software* (Dev) e a equipe de operações (Ops) (KIM et al., 2016). Em sua revisão sistemática, Leite et al. (2019) define DevOps como um esforço colaborativo e multidisciplinar em uma organização para automatizar a entrega contínua de novas versões de *software*, garantindo sua correção, confiabilidade e manutenibilidade.

Historicamente, as equipes de desenvolvimento e de operações atuavam em silos, com objetivos conflitantes: o desenvolvimento era voltado para a entrega rápida de novas funcionalidades, enquanto as operações priorizavam a estabilidade do ambiente de produção (LEITE et al., 2019). Essa separação, marcada por processos, ferramentas e bases de conhecimento independentes, resultava em longos ciclos de entrega, processos manuais propensos a erros e dificuldades na resolução de problemas (LEITE et al., 2019). A problemática da colaboração é evidenciada em trabalhos como o de Khan et al. (2022), que reporta a ausência de colaboração e de comunicação como um dos desafios mais críticos para a adoção da cultura DevOps. Segundo

Khan et al. (2022), essa lacuna dificulta o compartilhamento de objetivos e planos, contribuindo para atrasos no desenvolvimento de *software* e na entrega no ambiente de produção.

A principal motivação do DevOps reside em romper essas barreiras, promovendo uma cultura de responsabilidade compartilhada, comunicação e colaboração (KIM et al., 2016). Em vez de uma simples passagem de bastão, as equipes passam a trabalhar conjuntamente durante todo o ciclo de vida da aplicação (KIM et al., 2016). Esse modelo cultural baseia-se em um conjunto de práticas conhecido como as "Três Vias" (*The Three Ways*), cujos princípios foram formalizados em Kim, Behr e Spafford (2013) e, posteriormente, expandidos em Kim et al. (2016), orientando a implementação bem-sucedida da cultura DevOps. As três vias podem ser descritas da seguinte forma:

- ***Primeira Via* (*Flow*):** concentra-se na otimização do fluxo de trabalho da esquerda para a direita, do desenvolvimento até as operações e o cliente final, por meio da redução do tamanho dos lotes de trabalho, da eliminação de gargalos e da prevenção de defeitos (KIM; BEHR; SPAFFORD, 2013; KIM et al., 2016). O fluxo desta via é representado na Figura 2.2.

Figura 2.2 – Primeira Via: *Flow*

Fonte: Adaptado de (KIM, 2012).

- ***Segunda Via* (*Feedback*):** enfatiza a criação e a amplificação de ciclos de *feedback* rápidos da direita para a esquerda, das operações para o desenvolvimento, permitindo a detecção e a correção precoces de problemas por meio de monitoramento contínuo, alertas automatizados e análises pós-incidente (KIM et al., 2016). O fluxo desta via é representado na Figura 2.3.

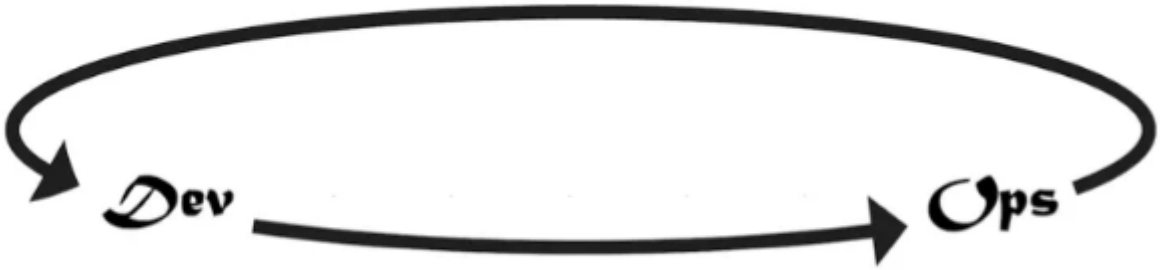

Figura 2.3 – Segunda Via: *Feedback*

Fonte: Adaptado de (KIM, 2012).

- ***Terceira Via* (*Continuous Learning and Experimentation*):** promove uma cultura de experimentação contínua e de aprendizado organizacional, na qual falhas são tratadas como oportunidades de melhoria para fomentar a inovação (KIM et al., 2016). Uma prática exemplar desta via é a Engenharia do Caos (*Chaos Engineering*), que envolve a injeção deliberada e controlada de falhas em um ambiente de produção, por exemplo, pela simulação da queda de um servidor ou pela indução de latência na rede, para descobrir proativamente as fragilidades de um sistema. Ao transformar a experimentação com falhas em uma prática rotineira, as equipes aprendem a construir sistemas mais resilientes e a responder a incidentes de forma mais eficaz, alinhando-se ao princípio de aprendizado contínuo, como demonstra a Figura 2.4 (KIM et al., 2016; LEITE et al., 2019).

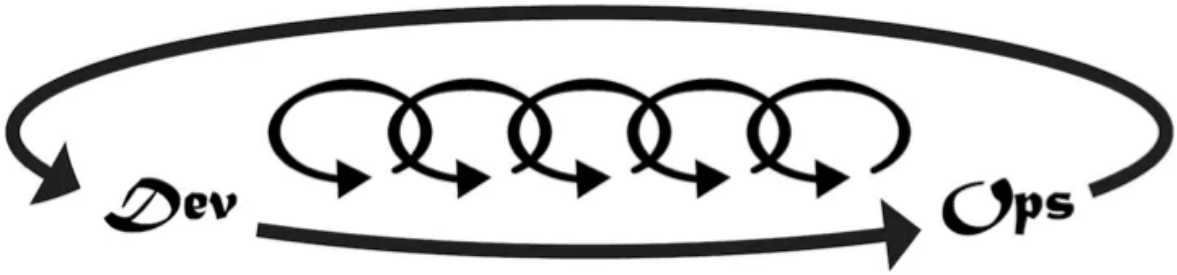

Figura 2.4 – Terceira Via: *Continuous Learning and Experimentation*

Fonte: Adaptado de (KIM, 2012).

No contexto deste trabalho, a adoção de uma mentalidade DevOps viabiliza a implementação de um *pipeline* de implantação eficiente, pois a automação técnica somente atinge seu potencial máximo quando é acompanhada por uma cultura organizacional que a apoie (KIM et al., 2016).

2.1.3 A Pirâmide de Testes e a Automação no *Pipeline* de Implantação

A qualidade dos artefatos gerados por um *pipeline* de implantação depende diretamente da forma como a automação dos testes é estruturada ao longo do fluxo de implantação (HUMBLE; FARLEY, 2010). Nesse contexto, a chamada "Pirâmide de Testes", proposta por Cohn (2009), oferece uma metáfora visual para organizar o portfólio de testes de maneira hierárquica. A ideia central é combinar diferentes granularidades de teste, distribuindo-as de modo que a maior parte da automação se concentre em testes rápidos e de menor custo (isto é, na base da pirâmide), enquanto um número reduzido de testes mais lentos e caros seja reservado aos níveis superiores (COHN, 2009).

A Figura 2.5 ilustra a Pirâmide de Testes, com três camadas principais (a saber: testes unitários, testes de serviço e testes de interface do usuário) que podem ser enriquecidas com categorias adicionais, como testes de aceitação e segurança, sem descaracterizar o formato piramidal (VOCKE, 2018; COHN, 2009). De maneira geral, quanto mais próxima a camada estiver da base, maior é a quantidade de testes automatizados esperada; à medida que se sobe na pirâmide, o número de testes diminui, enquanto o custo e o tempo de execução aumentam (COHN, 2009).

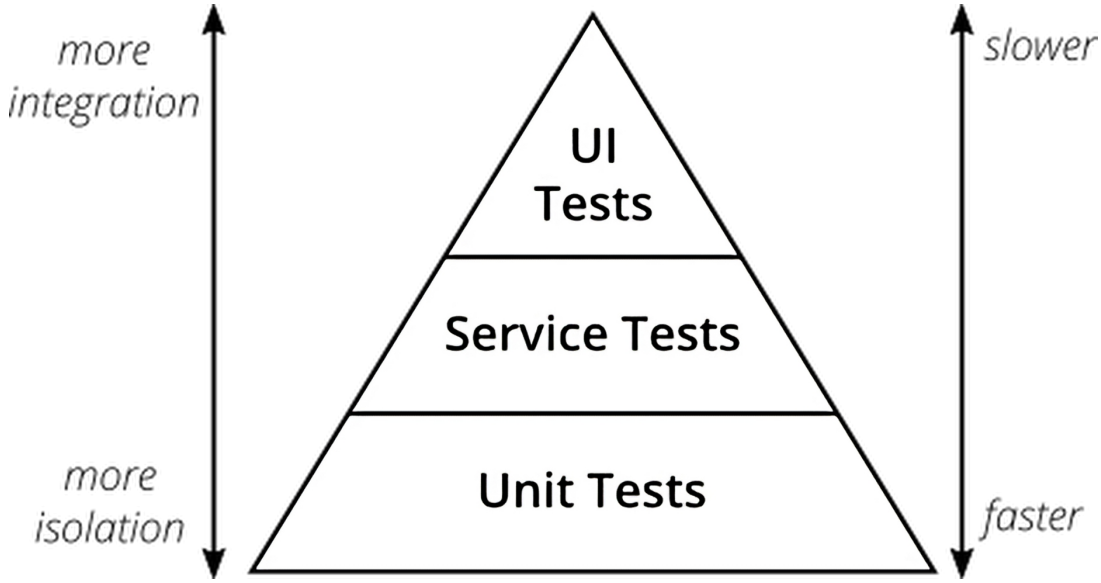

Figura 2.5 – A Pirâmide de Testes

Fonte: (VOCKE, 2018).

2.1.3.1 Base da pirâmide: testes unitários e verificações estáticas

Na base da pirâmide estão os testes unitários (do inglês, *unit tests*), que testam unidades de código isoladas, como funções, métodos ou classes, sem dependência de recursos externos, como banco de dados ou serviços remotos (COHN, 2009). Por serem rápidos e altamente determinísticos, esses testes são candidatos naturais para execução a cada *commit* no estágio de CI, fornecendo *feedback* imediato sobre regressões funcionais e facilitando refatorações seguras (UGWUEZE;

CHUKWUNWEIKE, 2025). Ferramentas como JUnit e NUnit em ambientes *back-end*, bem como *frameworks* como Jest ou `pytest` em camadas de aplicação, são exemplos amplamente utilizados para esse tipo de validação automatizada (VOCKE, 2018; UGWUEZE; CHUKWUNWEIKE, 2025). Em termos de quantidade, espera-se que o número de testes unitários seja de uma ordem de grandeza maior do que o número de testes situados nas camadas superiores, garantindo ampla cobertura das regras de negócios e da lógica de domínio (COHN, 2009; VOCKE, 2018).

Complementando os testes unitários, muitos *pipelines* de integração contínua incorporam verificações estáticas de código e análise de qualidade como parte da etapa de CI (UGWUEZE; CHUKWUNWEIKE, 2025). Ferramentas como SonarQube e Checkstyle realizam inspeções automatizadas em busca de *code smells*, vulnerabilidades conhecidas e violações de padrões de codificação, integrando-se ao *pipeline* de integração contínua para bloquear *builds* que não atendam a critérios mínimos de qualidade (UGWUEZE; CHUKWUNWEIKE, 2025). Embora não façam parte da pirâmide original, essas verificações estáticas são executadas com frequência semelhante à dos testes unitários e contribuem para que os defeitos sejam eliminados o mais cedo possível no fluxo (UGWUEZE; CHUKWUNWEIKE, 2025).

2.1.3.2 Nível intermediário: testes de serviço, integração e aceitação

O nível intermediário da pirâmide é ocupado pelos testes de serviço ou de integração, que verificam o comportamento da aplicação por meio de suas interfaces públicas, frequentemente interagindo com bancos de dados, filas de mensagens ou serviços de terceiros (COHN, 2009; VOCKE, 2018). Devido ao maior custo de configuração do ambiente e de execução, eles são menos numerosos do que os testes unitários; porém, continuam suficientemente rápidos para serem executados em *pipelines* de integração contínua, tipicamente em *builds* completos ou em filas assíncronas após os testes de base (UGWUEZE; CHUKWUNWEIKE, 2025). Ferramentas como REST-assured, Postman e os *integration tests* do Spring Boot são comumente utilizadas para essa camada (VOCKE, 2018; UGWUEZE; CHUKWUNWEIKE, 2025).

2.1.3.3 Topo da pirâmide: testes de interface de usuário e testes ponta a ponta

No topo da pirâmide encontram-se os testes de Interface de Usuário (*User Interface*, UI) e os testes Ponta a Ponta (*end-to-end*, E2E), que testam a aplicação da forma mais próxima possível do uso real, navegando pela interface gráfica ou orquestrando fluxos completos entre múltiplos serviços (COHN, 2009; VOCKE, 2018). Por envolver diversos componentes e dependências externas (por exemplo, bancos de dados reais, serviços de terceiros e infraestrutura de rede), esses testes tendem a ser mais lentos, frágeis e caros de manter (COHN, 2009). Tanto Cohn (2009) quanto Vocke (2018) recomendam que a quantidade dessa categoria seja deliberadamente limitada, concentrando-se nas jornadas de usuário e nos fluxos de integração mais críticos.

Ferramentas como Selenium, Cypress, Playwright e Appium são amplamente utilizadas para a automação de testes de interface e de cenários E2E em aplicações web e móveis (VOCKE,

2018; UGWUEZE; CHUKWUNWEIKE, 2025). Na prática, esses testes são mais frequentemente posicionados na etapa de CD, sendo executados em ambientes de homologação ou pré-produção, muitas vezes em *pipelines* de implantação acionados por *merge* em *branches* de *release* ou por *tags* específicas (UGWUEZE; CHUKWUNWEIKE, 2025). Apenas um subconjunto bem selecionado desses cenários deve ser executado em cada *build*, de forma a não comprometer tempo e os recursos do *pipeline* de implantação (VOCKE, 2018; RANGNAU et al., 2020). Assim, a pirâmide de testes recomenda que esses casos constituam a menor parcela dos testes automatizados, ainda que sejam responsáveis por validar as jornadas mais críticas do ponto de vista do usuário final (COHN, 2009).

2.1.3.4 Testes de segurança

A adoção de práticas de DevSecOps tem impulsionado a integração de técnicas de segurança às rotinas de DevOps, com ênfase na automação de testes de segurança e na incorporação de ferramentas de segurança ao longo do *pipeline* de implantação para reduzir a exposição a vulnerabilidades em ciclos de entrega frequentes (UGWUEZE; CHUKWUNWEIKE, 2025; RANGNAU et al., 2020). No nível mais baixo, verificações estáticas de segurança e ferramentas de análise de dependências são integradas ao *pipeline* de integração contínua para identificar vulnerabilidades conhecidas em bibliotecas, erros de configuração ou *code smells* relacionados à segurança ainda na fase de *build* (UGWUEZE; CHUKWUNWEIKE, 2025). Em camadas superiores, estudos de *continuous security testing* demonstram a viabilidade de integrar ferramentas de Teste Dinâmico de Segurança de Aplicação (*Dynamic Application Security Testing*, DAST) diretamente ao *pipeline* de implantação, executando varreduras automatizadas de vulnerabilidades em ambientes de integração ou homologação (RANGNAU et al., 2020). Esses testes dinâmicos simulam ataques externos contra a aplicação em execução, gerando relatórios com classificação de severidade para apoiar decisões de promoção de artefatos (RANGNAU et al., 2020). Entretanto, devido ao tempo de execução e ao custo de análise dos resultados, a literatura recomenda que a quantidade de varreduras dinâmicas seja controlada (por exemplo, em *builds* noturnos ou em serviços mais críticos), preservando a eficiência global do *pipeline* de implantação (RANGNAU et al., 2020).

Em síntese, a Pirâmide de Testes oferece um modelo conceitual para dimensionar a quantidade relativa de testes por camada, favorecendo uma base ampla de testes rápidos e uma redução progressiva conforme aumenta o custo de execução e manutenção (COHN, 2009; VOCKE, 2018). A literatura reforça que a automação de testes ao longo do *pipeline* de implantação contribui para detectar problemas mais cedo e reduzir custos de correção (UGWUEZE; CHUKWUNWEIKE, 2025). Dessa forma, a combinação de testes unitários e verificações estáticas, uma camada intermediária equilibrada de testes de serviço e de integração e um conjunto restrito de testes de UI tende a produzir um portfólio com *feedback* rápido, boa cobertura e custos controlados de execução e manutenção (COHN, 2009; VOCKE, 2018; UGWUEZE; CHUKWUNWEIKE, 2025).

2.1.4 Estratégias avançadas de *deployment*

Em contextos de entrega contínua, a implantação deixa de ser um evento excepcional e passa a ser executada como um processo sob demanda, que precisa ser rápido, repetível e confiável (HUMBLE; FARLEY, 2010). Nesse enquadramento, Humble e Farley (2010) argumentam que a confiabilidade do processo de liberação depende de torná-lo padronizado e previsível, de modo que a organização consiga promover versões para produção rotineiramente e com menor incerteza operacional (HUMBLE; FARLEY, 2010).

Historicamente, muitas equipes operaram com um modelo de implantação denominado *Big Bang deployment* (HUMBLE; FARLEY, 2010). Esse modelo é análogo ao que Malhotra et al. (2024) descrevem como *recreate deployment*: a atualização ocorre por substituição abrupta do serviço, interrompendo instâncias em execução e iniciando novas instâncias com a versão atualizada, o que implica indisponibilidade durante a transição. Embora simples do ponto de vista do planejamento imediato, esse formato tende a impor *downtime* e a concentrar riscos em uma janela curta (HUMBLE; FARLEY, 2010; MALHOTRA et al., 2024). Na ausência de práticas maduras de integração e entrega contínuas, mudanças podem ser promovidas manualmente, e a troca da versão em produção costuma envolver a paralisação e a substituição de instâncias, resultando em interrupção do serviço para os usuários (MALHOTRA et al., 2024).

A demanda por disponibilidade contínua e por ciclos de entrega menores desloca o *deployment* de um evento raro para uma atividade recorrente; nesse contexto, a indisponibilidade deixa de ser apenas um inconveniente e passa a ser um fator de risco de negócio e de experiência do usuário (RAMASWAMY, 2024; MALHOTRA et al., 2024). Ramaswamy (2024) argumenta que *deployments* sem *downtime* tornaram-se um requisito para sistemas modernos e que a incapacidade de atualizar sem interrupção pode induzir a "aversão a *deploy*", um comportamento em que o receio de causar falhas leva as equipes a desacelerar inovações, com liberações agrupadas, mais arriscadas e condicionadas a janelas de manutenção. Uma forma prática de mitigar esse efeito é desacoplar *deployment* de *release* por meio de *feature flags* (ou *feature toggles*): o código pode ser implantado em produção com a funcionalidade inicialmente desativada e, então, habilitado de forma controlada quando houver evidências de estabilidade (RAHMAN et al., 2016). Nesse arranjo, as *flags* permitem que funcionalidades ainda imaturas não bloqueiem o fluxo de entrega, ao mesmo tempo em que viabilizam reversões rápidas (RAHMAN et al., 2016). Consequentemente, emergem estratégias avançadas de *deployment* que buscam reduzir riscos e indisponibilidade por meio de transições controladas entre versões e de mecanismos de reversão (HUMBLE; FARLEY, 2010; MALHOTRA et al., 2024; RAMASWAMY, 2024).

2.1.4.1 *Rolling deployment*

O *rolling deployment* caracteriza-se pela substituição progressiva de componentes em execução (por exemplo, instâncias de serviço) por uma nova versão, em etapas sucessivas, até que todo o conjunto esteja atualizado (MALHOTRA et al., 2024; RUDRABHATLA, 2020). Em vez

de uma troca abrupta, o sistema passa por uma transição gradual, na qual versões distintas podem coexistir por um intervalo (BUZACHIS et al., 2019; RUDRABHATLA, 2020). Essa lógica é frequentemente associada a *rolling updates*: a troca ocorre incrementalmente, de modo que parte da capacidade permanece atendendo enquanto outra parte é atualizada, como é ilustrado pela Figura 2.6 (RUDRABHATLA, 2020).

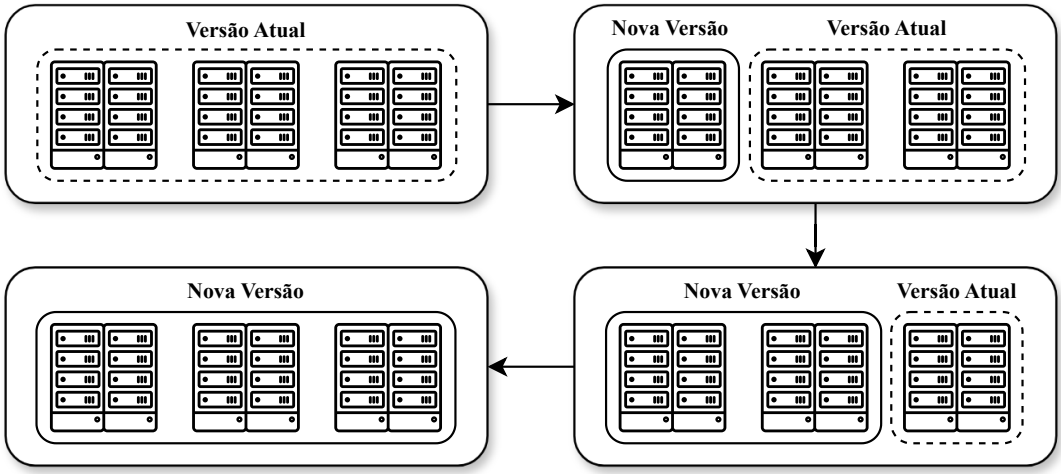

Figura 2.6 – Estratégia *Rolling deployment*

Fonte: Elaborado pelo autor (2026).

Em nível conceitual, a estratégia opera por ciclos repetidos: seleciona-se um subconjunto do serviço para atualização, realiza-se a troca de versão e, uma vez verificada sua estabilidade, avança-se para o próximo subconjunto (MALHOTRA et al., 2024). A principal promessa é evitar uma interrupção total do serviço, distribuindo o risco ao longo do tempo (MALHOTRA et al., 2024; RUDRABHATLA, 2020). Em ambientes modernos, esse tipo de implantação busca manter a continuidade do atendimento, desde que a nova versão seja compatível com a anterior durante o período de coexistência (BUZACHIS et al., 2019). No *RollingUpdate*, essa continuidade depende justamente da compatibilidade retroativa, que permite que novas e antigas versões coexistam durante a atualização (BUZACHIS et al., 2019).

A jornada de atualização de sistemas, contudo, é limitada pela complexidade de fazer diferentes versões coexistirem (BUZACHIS et al., 2019; RUDRABHATLA, 2020). Embora o *rolling deployment* surja como uma solução comum para manter a transição transparente e reduzir interrupções (MALHOTRA et al., 2024), ele impõe custos: exige capacidade excedente de infraestrutura e, por vezes, intervenções manuais para coordenar o processo (RUDRABHATLA, 2020). O impasse crítico surge diante de *breaking changes*, quando não há compatibilidade retroativa e a execução simultânea de versões antigas e novas pode introduzir falhas de contrato e inconsistências. Nesses casos, pode-se recorrer ao *recreate*, interrompendo o serviço antigo para dar lugar ao novo, ou, no âmbito da modernização gradual, adotar o padrão *Strangler Fig*, que

transfere comportamento progressivamente do legado para a nova base, usando a coexistência como ponte segura até a substituição completa (BUZACHIS et al., 2019; FOWLER, 2004).

2.1.4.2 *Blue–green deployment*

O *blue–green deployment* é uma estratégia em que duas cópias do ambiente produtivo são mantidas: uma ativa, intitulada "*blue*" e outra ociosa, a "*green*" (HUMBLE; FARLEY, 2010; MALHOTRA et al., 2024). A nova versão do sistema é implantada no ambiente ocioso, enquanto o ambiente ativo continua atendendo ao tráfego. Após validações e um período de estabilização, o tráfego é comutado do ambiente antigo para o novo (HUMBLE; FARLEY, 2010; MALHOTRA et al., 2024). Por definição, esse mecanismo pode ser descrito como uma abordagem baseada em dois ambientes idênticos e um componente de roteamento que direciona o tráfego ora para um, ora para outro, como observado na Figura 2.7 (HUMBLE; FARLEY, 2010).

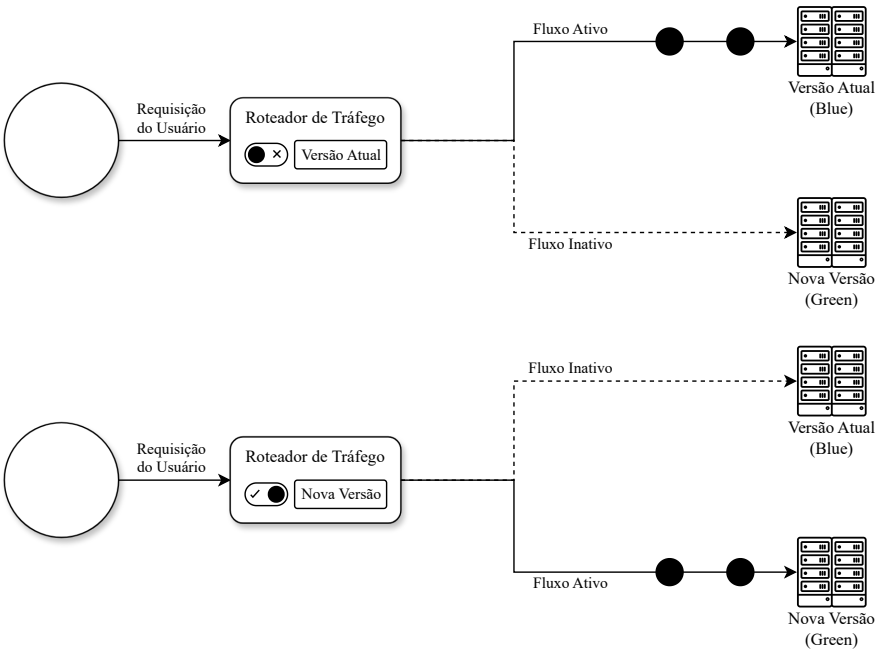

Figura 2.7 – Estratégia *Blue–green deployment*

Fonte: Elaborado pelo autor (2026).

Em formulação similar, Malhotra et al. (2024) descrevem o *blue–green deployment* como a manutenção de um ambiente produtivo paralelo, para o qual o tráfego é comutado somente após a validação da nova versão através de testes. Do ponto de vista conceitual, o padrão *blue–green* separa com clareza os atos de *implantar* e de *expor* uma mudança: a nova versão é implantada em um ambiente inativo, enquanto o ambiente atualmente em produção continua atendendo aos usuários; apenas quando a organização decide alterar o roteamento é que a mudança se torna visível (HUMBLE; FARLEY, 2010). Esse desenho produz dois benefícios centrais. Primeiro, *isolamento*: a validação do novo ambiente em paralelo permite identificar erros de configuração

e problemas não detectados em testes prévios antes que afetem a base usuária (HUMBLE; FARLEY, 2010). Segundo, *rollback* rápido: em caso de incidentes após a comutação, a reversão pode ser realizada pela simples retomada do tráfego para o ambiente anterior, reduzindo o tempo de recuperação (HUMBLE; FARLEY, 2010; BUZACHIS et al., 2019).

Apesar dessas vantagens, há limitações práticas relevantes. A mais imediata é o *custo*: por pressupor dois ambientes produtivos, a estratégia tende a demandar recursos adicionais e pode se tornar economicamente desfavorável em cenários de restrição orçamentária, ponto mencionado na comparação de técnicas de *zero-downtime* (RUDRABHATLA, 2020) e também na discussão comparativa de estratégias de DevOps, que destaca a necessidade de infraestrutura duplicada (RAMASWAMY, 2024). Além disso, Humble e Farley (2010) alertam que o *blue–green* não se reduz à comutação do tráfego de aplicação: quando há dependências compartilhadas (por exemplo, bancos de dados), a troca não é necessariamente instantânea, pois migrações de dados e mudanças de esquema podem exigir cuidados para evitar inconsistências. Nessa linha, Malhotra et al. (2024) também enfatizam que o *blue–green* demanda dois ambientes similares e que o tráfego só deve ser comutado após a nova versão ser testada, o que reforça a necessidade de coordenação e validação antes da transição. Assim, em ambientes produtivos reais, o *blue–green* tende a ser atraente quando o requisito dominante é reduzir o tempo de reversão e aumentar o isolamento entre a versão vigente e a candidata, desde que a organização aceite o custo e a complexidade associados a dependências compartilhadas (HUMBLE; FARLEY, 2010; RAMASWAMY, 2024).

2.1.4.3 *Canary deployment*

O *Canary Deployment* (ou *Canary Releasing*) é uma estratégia de liberação progressiva em que a nova versão é exposta, inicialmente, a uma fração pequena e controlada do tráfego; somente após evidências de estabilidade é que a exposição é ampliada até atingir toda a base de usuários (MALHOTRA et al., 2024; RAMASWAMY, 2024). Malhotra et al. (2024) sintetizam essa lógica ao caracterizar o *canary* como a exposição inicial a um subconjunto de usuários, seguida de expansão quando a validação é bem-sucedida. Em termos análogos, Ramaswamy (2024) descreve *canary releases* como a execução da nova versão lado a lado com a versão base, com roteamento gradual de tráfego para a nova versão, como pode ser observado na Figura 2.8.

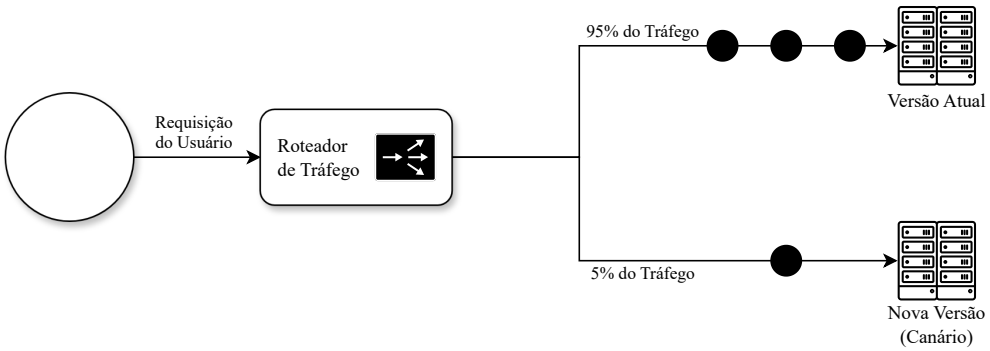

Figura 2.8 – Estratégia *Canary deployment*

Fonte: Elaborado pelo autor (2026).

A origem do termo está associada à metáfora do "canário na mina de carvão": um indicador precoce de condições perigosas. Humble e Farley (2010) explicitam essa analogia ao definirem *Canary Deployment* como a prática de liberar uma nova versão, inicialmente, para um subconjunto de servidores de produção e observar seu comportamento antes de generalizar, comparando diretamente essa exposição limitada ao papel do canário como alerta antecipado em minas de carvão. Essa noção de "exposição limitada" fundamenta o papel do *canary* como instrumento de gestão de risco: o objetivo não é apenas evitar a indisponibilidade, mas restringir o raio de impacto caso a nova versão apresente problemas (RAMASWAMY, 2024).

Conceitualmente, o funcionamento do *canary deployment* pode ser descrito em etapas: (i) prepara-se a nova versão para coexistir com a anterior; (ii) libera-se a nova versão para um grupo reduzido (por exemplo, percentual de tráfego ou grupo de usuários); (iii) coleta-se telemetria e compara-se o comportamento com um limiar; (iv) decide-se por ampliar gradualmente ou retornar à versão anterior; e (v) ao final, consolida-se a nova versão como padrão (RAMASWAMY, 2024; MALHOTRA et al., 2024). Nessa lógica, o *monitoramento* não é acessório: é o mecanismo que viabiliza a decisão de release ou *rollback*. Diante disso, a efetividade do *canary* depende de observabilidade e de controle preciso do roteamento de tráfego, uma vez que o valor da estratégia está na detecção antecipada de falhas com impacto limitado (RAMASWAMY, 2024). De forma convergente, o *canary deployment* pode ser descrito como uma técnica orientada a manter tanto a experiência do usuário quanto a disponibilidade da aplicação, o que pressupõe mecanismos de validação e reversão ao longo do processo (MALHOTRA et al., 2024).

As vantagens do *Canary Deployment* podem ser organizadas em quatro eixos. Primeiro, *mitigação de risco*: por expor inicialmente poucos usuários, defeitos e regressões tendem a ser percebidos antes de afetarem toda a população (RAMASWAMY, 2024). Evidência empírica apresentada por Ramaswamy (2024) sugere que anomalias podem ser detectadas em estágios iniciais (por exemplo, em torno de 5% do tráfego nos experimentos relatados), reduzindo o impacto sobre usuários quando comparado a liberações integrais. Segundo, *feedback em produção*: a estratégia permite observar o comportamento real sob carga e até apoiar experimentos controlados quando

se consegue selecionar quem receberá a nova versão (HUMBLE; FARLEY, 2010). Terceiro, *redução de custo relativo* frente a abordagens que duplicam ambientes completos: o *canary* pode ser aplicado sem a manutenção de dois ambientes paralelos integrais, reduzindo a despesa operacional em comparação ao *blue–green* (MALHOTRA et al., 2024). Quarto, *reversibilidade*: quando combinada a critérios objetivos de promoção, a estratégia favorece *rollbacks* rápidos e localizados, coerentes com a necessidade de praticar e tornar confiável o procedimento de reversão (HUMBLE; FARLEY, 2010).

Como contrapartida, o *Canary Deployment* impõe desafios que precisam ser explicitados. Um primeiro desafio é o *sobrecusto operacional de coordenação*: por envolver estágios e decisões sucessivas, a estratégia tende a exigir maior maturidade operacional, monitoramento contínuo e critérios bem ajustados de alerta (RAMASWAMY, 2024). Em particular, Ramaswamy (2024) destacam que o sucesso do *Canary Deployment* depende fortemente de observabilidade em tempo real e limiares de alerta bem calibrados, pois configurações inadequadas podem gerar falsos positivos e interrupções prematuras do *rollout*. Um segundo desafio está na *gestão de compatibilidade e recursos compartilhados*: a coexistência de versões exige controle cuidadoso de dependências e atenção a recursos comuns (por exemplo, bancos de dados), pois mudanças incompatíveis comprometem o *rollout* progressivo (HUMBLE; FARLEY, 2010; MALHOTRA et al., 2024).

Por fim, a estratégia se integra naturalmente a *pipelines* de implantação ao transformar a liberação em uma sequência de decisões baseadas em evidências observáveis (por exemplo, telemetria, indicadores de erro e desempenho), favorecendo a automação da promoção e do *rollback* quando os critérios definidos não são atendidos (MALHOTRA et al., 2024; RAMASWAMY, 2024). Ao modular o risco em "fatias", o *Canary Deployment* favorece mensurações comparativas entre estágios, coerentes com métricas de entrega e estabilidade, como tempo de recuperação e taxa de sucesso de *deploys* (VANGALA, 2020). Estudos comparativos indicam que essa abordagem está associada a ciclos de entrega mais rápidos e é particularmente utilizada em arquiteturas de microserviços e *Application Programming Interfaces* (APIs) com alto tráfego, nas quais pequenas alterações incrementais precisam ser validadas sob carga real (VANGALA, 2020; RAMASWAMY, 2024).

2.1.5 Mensuração de desempenho com as métricas DORA

Para mensurar de modo objetivo a eficácia do *pipeline* de implantação, este trabalho adota as quatro métricas consagradas pelo programa de pesquisa DORA (LTFC DF, CFR e MTTR) (FORSGREN; HUMBLE; KIM, 2018).[1] No relatório de DeBellis et al. (2024), mantiveram-se essas quatro métricas como base para avaliar a *velocidade* e a *estabilidade* da entrega. No mesmo relatório, a estabilidade é caracterizada por CFR e MTTR, enquanto a velocidade é caracterizada por LTFC e DF (DEBELLIS et al., 2024).

[1] Definições e guias oficiais das quatro métricas DORA: (DevOps Research and Assessment (DORA), 2025).

2.1.5.1 Como os rótulos são definidos

A classificação em quatro níveis (sendo eles: *Elite*, *Alto*, *Médio* e *Baixo*) não é definida *a priori* por limiares arbitrários. Em vez disso, é aplicada uma análise de agrupamento às respostas anuais do levantamento, de modo que os grupos emergem dos dados daquele ano (DEBELLIS et al., 2024). Por esse motivo, as faixas de referência variam ano a ano e, ocasionalmente, podem produzir níveis cujo ordenamento aparente não é monotônico quando se observa cada métrica isoladamente (DEBELLIS et al., 2024). No relatório de DeBellis et al. (2024), por exemplo, o grupo rotulado como *Médio* exibe maior estabilidade que o grupo *Alto*, embora seja mais lento. Os autores explicam explicitamente a opção editorial de denominar como *Alto* as equipes "mais rápidas" e como *Médio* as "mais lentas, porém estáveis".

2.1.5.2 Definições operacionais adotadas

No presente estudo, as quatro métricas são operacionalizadas conforme a literatura seminal: (i) DF: contagem de implantações de produção em um período; (ii) LTFC: tempo do *commit* até a primeira implantação bem-sucedida em produção dessa mudança; (iii) *CFR*: proporção de implantações que causaram falhas em produção e demandaram *rollback* ou *hotfix*; (iv) *MTTR*: tempo desde o início de um incidente causado por implantação até a restauração do serviço (FORSGREN; HUMBLE; KIM, 2018; BEYER et al., 2016; DEBELLIS et al., 2024).

2.1.5.3 Benchmarks DORA 2024 e interpretação dos rótulos

As Tabelas 2.1 e 2.2 reproduzem as faixas de 2024 por nível de desempenho. Esses valores são *descritivos* dos grupos estudados por DeBellis et al. (2024); não constituem metas universais nem padrões normativos. Além disso, o relatório informa a participação estimada de cada nível naquele ano: *Elite* (19%), *Alto* (22%), *Médio* (35%) e *Baixo* (25%) (DEBELLIS et al., 2024).

Tabela 2.1 – Métricas de velocidade — *benchmarks* DORA (2024)

Nível	***Lead time* para mudanças LTFC**	**Frequência de Implantação DF**
Elite	Menos de um dia	Sob demanda (várias implantações por dia)
Alto	Entre um dia e uma semana	Entre uma implantação por dia e uma por semana
Médio	Entre uma semana e um mês	Entre uma implantação por semana e uma por mês
Baixo	Entre um mês e seis meses	Entre uma implantação por mês e uma por semestre

Fonte: Fonte: quadro "Níveis de desempenho" (DEBELLIS et al., 2024).

Tabela 2.2 – Métricas de estabilidade — *benchmarks* DORA (2024)

Nível	Taxa de Falha nas Mudanças CFR	Tempo para Restaurar MTTR
Elite	5%	Menos de uma hora
Alto	20%	Menos de um dia
Médio	10%	Menos de um dia
Baixo	40%	Entre uma semana e um mês

Fonte: Fonte: quadro “Níveis de desempenho” (DEBELLIS et al., 2024).

Os níveis de desempenho são rótulos atribuídos a *clusters* derivados de múltiplas métricas simultaneamente. Em DeBellis et al. (2024), os autores destacam que o grupo *Médio* é “mais estável, porém mais lento” que o *Alto*; por isso, quando se observa somente a CFR, o *Médio* aparece com 10% versus 20% no *Alto*. Além disso, *Alto* e *Médio* compartilham o mesmo intervalo de MTTR (< 1 dia). Essa denominação foi uma decisão editorial para enfatizar a velocidade (isto é, grupos “mais rápidos” rotulados como *Alto*) e reconhecer um grupo “mais lento, porém estável” como *Médio* (DEBELLIS et al., 2024). A Figura 2.9 ilustra precisamente como essas métricas se distribuem entre os níveis, revelando as sobreposições e diferenciações que justificam tal denominação. Adicionalmente, o quadro do relatório apresenta um intervalo de incerteza (89%) para a proporção de participantes por nível, reforçando que os valores são estimativas amostrais daquele ano (DEBELLIS et al., 2024).

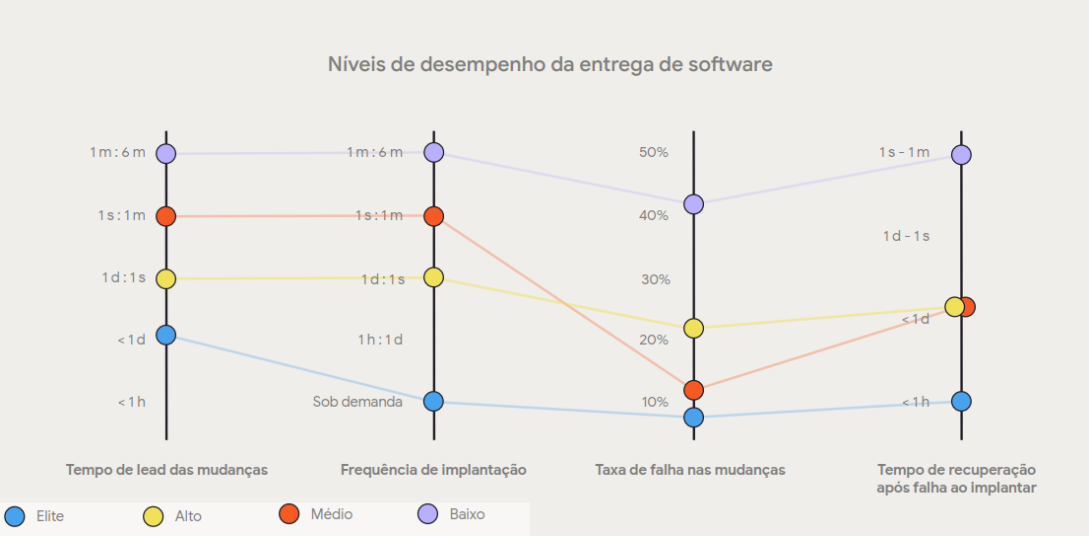

Figura 2.9 – Níveis de desempenho na entrega de software

Fonte: (DEBELLIS et al., 2024).

2.1.5.4 Coleta de dados

A coleta dessas métricas pode ser realizada de forma manual, por meio de questionários e pesquisas qualitativas, ou de forma automatizada, a partir de dados extraídos de ferramentas ligadas ao *pipeline* de implantação (RüEGGER et al., 2024). Em seu estudo, Rüegger et al.

(2024) identificam limitações significativas na abordagem manual: a subjetividade das respostas, que é influenciada por viés humano e pressão política organizacional; a imprecisão inerente a escalas qualitativas vagas; a degradação da qualidade das respostas ao longo do tempo, devido a problemas de memória; a baixa granularidade temporal, uma vez que as pesquisas são conduzidas apenas ocasionalmente; e os custos recorrentes associados à execução e análise de questionários. Em contraste, a coleta automatizada baseada em dados de ferramentas de CI/CD oferece maior precisão, alta granularidade, escalabilidade sem custos adicionais e visibilidade em tempo real do processo de entrega (RüEGGER et al., 2024; SALLIN et al., 2021).

Nesta pesquisa, as métricas DORA foram coletadas de forma automatizada e constituíram o principal instrumento de avaliação quantitativa do impacto da estratégia de implantação implementada, o que permitiu mensurar seus efeitos tanto sobre a velocidade quanto sobre a estabilidade da entrega da aplicação.

2.2 Trabalhos Relacionados

Esta seção revisa a literatura científica, contextualizando a pesquisa e traçando a evolução do conhecimento sobre a entrega de *software*. A análise parte de uma perspectiva histórica na subseção 2.2.1, abordando desde o gerenciamento clássico de *releases* até a formalização da entrega contínua. Em seguida, a subseção 2.2.2 examina a validação científica das práticas DevOps e seus impactos organizacionais. Na sequência, a subseção 2.2.3 aprofunda-se nas ferramentas e estratégias de implantação, com ênfase nas abordagens de *canary deployment* e nas métricas de eficiência. Por fim, a subseção 2.2.4 sintetiza os achados e identifica as lacunas de pesquisa, especialmente no que tange a ambientes com restrição de recursos, posicionando a contribuição deste trabalho frente à literatura analisada.

2.2.1 Do Gerenciamento de Release à Entrega Contínua

As raízes da automação na liberação de *software* remontam a iniciativas anteriores à popularização da terminologia de CI/CD, quando a principal preocupação ainda era tornar o processo de disponibilização de versões menos manual e mais previsível. Em 1997, Hoek et al. (1997) investigaram, no trabalho intitulado *Software Release Management*, os desafios de gerenciar liberações em arquiteturas complexas de "sistemas de sistemas", nas quais uma aplicação é construída pela composição de diversos componentes pré-existentes, produzidos e liberados de forma independente por diferentes organizações.

Para enfrentar a complexidade dessas dependências, os autores propuseram o *Software Release Manager (SRM)*, um protótipo de ferramenta que mantinha um catálogo centralizado de componentes e suas versões, registrava explicitamente as relações de dependência entre eles e automatizava a recuperação de um conjunto consistente de artefatos para cada liberação (HOEK et al., 1997). Em termos conceituais, o SRM funciona como um "bibliotecário automatizado"

responsável por garantir que o usuário obtenha exatamente a combinação correta de componentes para montar o sistema, antecipando práticas hoje comuns em *pipelines* de CI/CD, como o uso de repositórios de artefatos versionados, a resolução automática de dependências e a coordenação da distribuição de componentes entre ambientes, embora o foco do trabalho permaneça restrito à gestão de *releases*, e não à integração contínua (HOEK et al., 1997).

Quase uma década depois daqueles primeiros movimentos em direção à automação das liberações de *software*, a prática de CI foi sistematizada por Fowler e Foemmel (2006) como uma disciplina na qual cada membro da equipe integra suas alterações ao repositório principal pelo menos uma vez por dia. Complementando essa visão conceitual, Duvall, Matyas e Glover (2007) detalham a CI como uma prática operacional focada na redução de riscos e na eliminação de processos manuais repetitivos: ao disparar um *build* automatizado a cada alteração, a equipe garante que o *software* esteja sempre em um estado funcional (FOWLER; FOEMMEL, 2006; DUVALL; MATYAS; GLOVER, 2007).

Para evitar esse cenário, a CI propõe que as integrações sejam frequentes e validadas por mecanismos robustos de verificação. Segundo Duvall, Matyas e Glover (2007), uma CI eficaz vai além da compilação, englobando também a execução de testes automatizados, a inspeção contínua da qualidade do código e até a integração de esquemas de banco de dados. Esse fluxo estabelece um ciclo de *feedback* rápido: qualquer mudança que quebre o sistema é detectada em poucos minutos, permitindo que a falha seja corrigida enquanto os detalhes da implementação ainda estão frescos na memória do desenvolvedor (FOWLER; FOEMMEL, 2006; DUVALL; MATYAS; GLOVER, 2007).

Expandindo esse raciocínio para além da fase de integração, Humble e Farley (2010) formalizam o conceito de CD como a capacidade de manter o *software* sempre em um estado potencialmente liberável, por meio de um processo automatizado que abrange desde o controle de versões até os testes e a implantação em ambientes próximos ao produtivo. Nesse contexto, os autores propõem o *pipeline de Implantação* como um mecanismo que representa, de forma automatizada, todas as etapas necessárias para levar uma mudança do código-fonte até um candidato a *release*, passando por estágios sucessivos de *build*, testes e validação de implantação (HUMBLE; FARLEY, 2010). Assim, o foco desloca-se da simples integração de fluxos de desenvolvimento para a capacidade organizacional de produzir *releases* frequentes, confiáveis e amplamente automatizados (HUMBLE; FARLEY, 2010).

2.2.2 Consolidação do Conhecimento e a Expansão da Cultura DevOps

A consolidação do movimento DevOps não surgiu de forma abrupta, mas sim como resultado da maturação de práticas técnicas, como automação de infraestrutura (IaC), monitoramento contínuo, Integração Contínua e Entrega Contínua, combinadas com uma mudança organizacional orientada à colaboração entre desenvolvimento e operações (KIM et al., 2016). Nessa perspectiva, DevOps passa a ser entendido como um esforço colaborativo e multidiscipli-

nar para automatizar a entrega contínua de novas versões de *software*, assegurando correção e confiabilidade, em contraste com modelos tradicionais baseados em silos funcionais (LEITE et al., 2019).

Em uma influente revisão sistemática da literatura, Leite et al. (2019) consolidam esse entendimento ao mapear os principais conceitos, práticas e desafios de DevOps, propondo uma definição que enfatiza tanto a automação da cadeia de entrega quanto a integração entre papéis historicamente separados, conforme ilustrado no ciclo de vida da Figura 2.10.

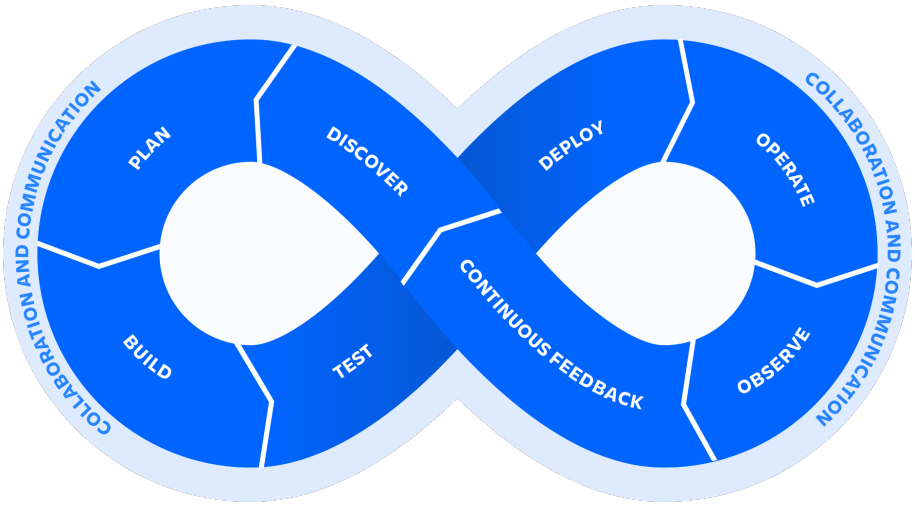

Figura 2.10 – Ciclo de vida do DevOps

Fonte: (Atlassian, 2025).

Os autores destacam que a quebra de silos entre as equipes de desenvolvimento (Dev) e operações (Ops) é uma condição central para o aumento da eficiência: enquanto o desenvolvimento tende a ser avaliado pela velocidade na entrega de novas funcionalidades, as operações são tradicionalmente medidas pela estabilidade e confiabilidade do ambiente de produção, gerando objetivos conflitantes (KIM et al., 2016; LEITE et al., 2019). Esse desalinhamento produz o conhecido *wall of confusion* (“muro de confusão”), no qual requisitos são “atirados por cima do muro” entre as equipes, resultando em ciclos de entrega longos, retrabalho, desconfiança e um ambiente propenso a falhas (KIM et al., 2016; LEITE et al., 2019).

Superando a visão anedótica ou baseada apenas em modelos de maturidade estáticos, Forsgren, Humble e Kim (2018) forneceram a validação científica para esse modelo por meio de quatro anos de pesquisa com dados de milhares de organizações. Ao analisarem o desempenho de entrega de *software*, os autores refutaram estatisticamente o mito de que existe um *trade-off* necessário entre velocidade e estabilidade: as organizações de alto desempenho demonstraram capacidade de implantar código com uma frequência muito superior, mantendo, simultaneamente, taxas menores de falha e tempos mais rápidos de restauração de serviço (FORSGREN; HUMBLE; KIM, 2018). Para Forsgren, Humble e Kim (2018), o foco deve recair no desenvolvimento de

capacidades concretas que predizem o desempenho organizacional, mensurável por meio das quatro métricas chave do DORA.

Em paralelo à consolidação científica e prática de DevOps, emergiu a percepção de que a segurança não poderia continuar sendo tratada como uma etapa tardia, manual e reativa do ciclo de vida de *software* (YASAR; KONTOSTATHIS, 2016). A partir dessa constatação, o conceito de DevSecOps (ou *Secure DevOps*) passou a integrar explicitamente profissionais e práticas de segurança ao fluxo contínuo de desenvolvimento e operações, tornando a segurança uma responsabilidade compartilhada ao longo de todo o processo (YASAR; KONTOSTATHIS, 2016). Em vez de atuar como um gargalo, a segurança é incorporada de forma automatizada aos *pipelines* de implantação, por meio de verificações estáticas e dinâmicas, testes de intrusão automatizados e políticas de configuração codificadas, inclusive com a integração de testes dinâmicos de segurança diretamente nos *pipelines* de implantação (YASAR; KONTOSTATHIS, 2016; RANGNAU et al., 2020). Nessa metáfora, o ciclo de desenvolvimento é visualizado como uma linha do tempo que progride da esquerda (isto é, codificação e integração) para a direita (isto é, implantação e operação): *shifting security to the left* ("deslocar a segurança para a esquerda") significa antecipar e repetir continuamente essas práticas desde as primeiras fases de construção e integração do código, reduzindo o custo e o impacto de vulnerabilidades descobertas tardiamente (YASAR; KONTOSTATHIS, 2016).

À medida que as práticas de CI/CD foram amplamente adotadas, a comunidade acadêmica passou a sistematizar o conhecimento acumulado. Em uma revisão sistemática de abordagens, ferramentas e desafios relacionados a CI/CD, Shahin, Babar e Zhu (2017) oferecem uma visão abrangente do estado da arte, identificando, entre outras recomendações, a necessidade de otimizar o tempo de *build* e teste, aumentar a visibilidade dos resultados do *pipeline* de implantação e apoiar testes automatizados em múltiplos níveis (isto é, unitário, integração e aceitação) (SHAHIN; BABAR; ZHU, 2017). A revisão enfatiza, ainda, que a dependabilidade, a confiabilidade e a segurança dos *pipelines* de implantação são fatores críticos para que as organizações consigam sustentar ciclos de entrega realmente contínuos, sem degradar a qualidade do produto entregue ao usuário final (SHAHIN; BABAR; ZHU, 2017).

O gerenciamento do código-fonte, matéria-prima de todo o processo de *deployment*, também foi objeto de reflexão específica. Em sua análise de padrões de ramificação (*branching patterns*), Fowler (2020) argumenta que estratégias baseadas em Desenvolvimento Baseado em Tronco (*Trunk-Based Development*, TBD), nas quais todos os desenvolvedores integram pequenas alterações diretamente na linha principal de código (*trunk*), são as mais alinhadas aos princípios de CI (FOWLER, 2020). Ao incentivar integrações frequentes e limitar a duração e o escopo de *branches* temporários, o TBD reduz a complexidade dos *merges*, mitiga o *integration hell* e mantém a base de código em um estado constantemente passível de *deploy*, reforçando, na prática, os objetivos de CI/CD (FOWLER, 2020).

2.2.3 Ferramentas, Estratégias e Métricas de Implantação

Com a consolidação dos princípios de CI/CD, parte da literatura recente passou a tratar o *pipeline* de implantação como um artefato técnico passível de instrumentação e avaliação, composto por etapas automatizadas que, tipicamente, incluem gerenciamento de código-fonte, *build*, execução de testes, implantação e monitoramento Goyal (2024). Nessa linha, estudos direcionam esforços para quantificar e aprimorar o desempenho do *pipeline* de implantação a partir de métricas operacionais, como tempo de *build*, tempo de execução de testes, taxa de sucesso e frequência de implantação, buscando reduzir atrasos e aumentar a previsibilidade do fluxo de entrega (GOYAL, 2024; SINGH et al., 2019; KHAN et al., 2022). Como consequência, o foco desloca-se de modelos de liberação pouco frequentes e de maior risco (por exemplo, o modelo *big bang deployment*), associados a entregas extensas em uma única implantação, para estratégias de publicação mais incrementais, com menor impacto em ambiente de produção e maior capacidade de resposta a falhas (GOYAL, 2024).

No que se refere às estratégias avançadas para aumentar a segurança e o controle no processo de *release*, a literatura destaca a importância de reduzir o impacto de falhas por meio de exposições controladas. Shahin, Babar e Zhu (2017) identificam práticas de liberação parcial (por exemplo, *canary deployment*) como mecanismos essenciais para mitigar riscos. Ao expor a nova versão apenas a um subconjunto de usuários ou ocultar funcionalidades, essas estratégias permitem comparar o comportamento do novo código em relação ao antigo antes de uma liberação total (SHAHIN; BABAR; ZHU, 2017).

No entanto, a transposição dessas práticas avançadas para o contexto de PME e equipes enxutas enfrenta fricções específicas de ordem operacional e financeira. Tuape et al. (2021) argumentam que muitas das ferramentas e processos de engenharia de *software* disponíveis no mercado são projetados sob a ótica de grandes corporações, apresentando uma complexidade inerente e curvas de aprendizado que resultam em subutilização ou inadequação para empresas de menor porte, cujo foco reside na sobrevivência e na entrega imediata de funcionalidades. Esse descompasso é corroborado por Khan et al. (2022), que identificam a infraestrutura complicada e a falta de mão de obra especializada como barreiras críticas para a adoção de uma cultura DevOps plena. Do ponto de vista da infraestrutura física, Debroy e Miller (2020) demonstram, por meio de um relato de experiência em uma empresa de pequeno porte, que a migração para arquiteturas modernas impõe uma pressão significativa sobre o *pipeline* de implantação; segundo os autores, a escalabilidade horizontal ingênua de recursos de *build* e *deploy* para suportar múltiplos serviços simultâneos acarreta um aumento linear e proibitivo nos custos de nuvem. Consequentemente, para organizações com restrições orçamentárias, a duplicação de ambientes exigida por estratégias como *blue-green deployment* torna-se inviável, exigindo abordagens que maximizem a automação e a orquestração de recursos limitados, sem comprometer a estabilidade da entrega (DEBROY; MILLER, 2020; ABBASS et al., 2019).

Diante dessas restrições, Shahin, Babar e Zhu (2017) reforçam que a falta de “infraes-

trutura apropriada" se torna um dos fatores mais críticos para o fracasso na implementação de práticas contínuas, uma vez que a automação do fluxo de ponta a ponta exige recursos computacionais adicionais que impactam o orçamento organizacional. É nesse cenário que a capacidade de recuperação se torna vital. Segundo Chen (2015), a confiabilidade do processo reside na capacidade do *pipeline* de implantação de validar automaticamente cada alteração de código, abortando o processo caso falhas sejam detectadas antes de chegar ao ambiente de produção. Essa automação da reversão é citada como um fator determinante para reduzir drasticamente o risco da liberação e o estresse das equipes de engenharia, transformando o *deploy* em uma atividade rotineira e segura (CHEN, 2015; RAMASWAMY, 2024). A partir dessa base, a automação do processo decisório durante *rollouts* progressivos torna-se objeto de investigação, com propostas que combinam monitoramento em tempo real, detecção de anomalias e técnicas de Aprendizado de Máquina (*Machine Learning*, ML) para recomendar continuidade, desaceleração ou reversão do *deployment* com base em métricas de saúde do sistema (SOLANKI; THASON, 2025).

Apesar do potencial, abordagens de automação baseadas em ML enfrentam restrições relevantes de aplicabilidade. Em particular, Solanki e Thason (2025) identificam a automação de *canary deployment* em ambientes dinâmicos como um desafio, uma vez que variações rápidas de tráfego e do comportamento do sistema podem degradar a robustez das inferências, elevando a incerteza sobre o impacto de uma nova versão. Resultados de DeBellis et al. (2025) reforçam que os ganhos com Inteligência Artificial (*Artificial Intelligence*, IA) não se materializam de forma uniforme: a tecnologia tende a atuar como um "amplificador" das condições sociotécnicas existentes, podendo coexistir com instabilidade e redução de *throughput*. Tal dependência de condições prévias encontra paralelo na literatura de engenharia de software: revisões sistemáticas indicam que a eficácia preditiva de modelos de ML está intrinsecamente ligada à qualidade e à disponibilidade de dados históricos e métricas padronizadas, cuja escassez limita a generalização dos modelos, mesmo em tarefas estáticas, como a previsão de manutenibilidade (ALSOLAI; ROPER, 2019). Nesse sentido, a incorporação de ML ao *pipeline* de implantação pressupõe fundamentos sólidos de observabilidade, instrumentação consistente e disciplina de processos; do contrário, a automação pode introduzir variabilidade adicional e elevar o risco do *release* em vez de mitigá-lo (DEBELLIS et al., 2025; SOLANKI; THASON, 2025; ALSOLAI; ROPER, 2019).

Por fim, a mensuração consolidou-se como elemento estruturante da melhoria do *pipeline* de implantação, ao viabilizar decisões orientadas por dados sobre gargalos e compromissos entre velocidade e estabilidade. Nesse contexto, as métricas DORA, estabelecidas pela pesquisa de Forsgren, Humble e Kim (2018), são apresentadas como uma referência amplamente adotada para quantificar o desempenho na entrega e relacioná-lo aos resultados de desenvolvimento e operações. Com o objetivo de superar limitações de soluções proprietárias e dependentes de infraestrutura específica, Wilkes, Milani e Storey (2023) propõem um *framework* que operacionaliza essas métricas de forma independente do ciclo de vida e do tipo de implantação, demonstrando aplicabilidade ao calcular indicadores de estabilidade e velocidade em repositórios de código

aberto. Entretanto, o desafio se intensifica em cenários de escala e arquitetura distribuída: Rüegger et al. (2024) observam que a coleta manual é lenta, sujeita a erros e, frequentemente, restrita ao nível de equipes, o que dificulta análises granulares em ecossistemas com múltiplos microserviços. Como resposta, os autores desenvolvem e disponibilizam um sistema de medição totalmente automatizado, capaz de coletar eventos ao longo do ciclo de desenvolvimento e calcular as quatro métricas DORA quase em tempo real, evidenciando diferenças relevantes entre o desempenho de serviços individuais e indicadores agregados da equipe (RüEGGER et al., 2024).

2.2.4 Discussão e Considerações Finais do Capítulo

A literatura analisada indica um consenso: a maturidade em CI/CD e o uso de métricas, como as do DORA, são essenciais para entregas de *software* mais rápidas e estáveis (FORSGREN; HUMBLE; KIM, 2018; DEBELLIS et al., 2024). Contudo, emerge uma barreira prática significativa. Estratégias de implantação mais seguras, como *blue-green deployment* e *canary deployment*, são frequentemente discutidas no contexto de grandes organizações com recursos abundantes. Em ambientes de menor escala, a complexidade de orquestração e, principalmente, o custo de duplicar a infraestrutura tornam-se obstáculos proibitivos (VANGALA, 2020; MALHOTRA et al., 2024).

Essa realidade evidencia uma lacuna pragmática. Embora a literatura sugira que o *canary deployment* seja adequado para limitar o impacto de falhas, sua execução manual tende a ser mais arriscada e mais lenta, pois exige vigilância contínua (HUMBLE; FARLEY, 2010). Por outro lado, a automação integral desse processo pode detectar erros e acionar a reversão de versão sem intervenção humana; no entanto, essa abordagem permanece pouco explorada como uma solução viável para equipes enxutas (RAMASWAMY, 2024). A questão que permanece em aberto é se a automação pode substituir a necessidade de infraestrutura redundante, oferecendo um nível de segurança similar, a baixo custo (RAMASWAMY, 2024).

Portanto, esta pesquisa se posiciona na intersecção entre a segurança avançada e a realidade de recursos limitados. A investigação buscou preencher essa lacuna ao avaliar quantitativamente, por meio das métricas DORA, se a adoção de *canary deployment* com *rollback* automatizado permite que ambientes de pequena escala atinjam níveis de estabilidade e velocidade comparáveis aos de grandes operações, sem a sobrecarga operacional e financeira tradicionalmente associada a essas práticas.

3 Materiais e Métodos

O presente capítulo transpõe para o plano prático os fundamentos discutidos no Capítulo 2, apresentando a metodologia e o delineamento quase-experimental empregados para avaliar empiricamente se a automação do *rollback* em *canary deployment* promove níveis elevados de velocidade e estabilidade, mesmo em ambientes de pequena escala. Na seção 3.1, classifica-se a pesquisa quanto à natureza, abordagem, objetivos e procedimentos técnicos. Na seção 3.2, delimitam-se o contexto organizacional e o objeto de estudo. Na seção 3.3, descreve-se o estabelecimento da linha de base para comparação após a implantação das contribuições deste trabalho no contexto organizacional. Na seção 3.4, apresenta-se o projeto experimental de implantação. Na seção 3.5, definem-se as métricas e os critérios de avaliação. Por fim, na seção 3.6, detalha-se o protocolo de experimentos controlados de injeção de falhas.

3.1 Classificação da Pesquisa

Quanto à natureza, esta pesquisa classifica-se como aplicada e é conduzida sob o paradigma da *Design Science Research* (DSR). Tal categorização justifica-se pelo fato do trabalho objetivar a construção e a avaliação de um artefato prático, que consiste em um *pipeline* de implantação de *software* otimizado. Esse artefato volta-se à resolução de um problema concreto enfrentado por PMEs, caracterizadas pela necessidade de mitigar riscos operacionais sem a duplicação onerosa de infraestrutura. Conforme Hevner et al. (2004), a DSR é adequada quando o objetivo é gerar conhecimento por meio da criação de artefatos inovadores capazes de resolver problemas reais.

No que tange à abordagem do problema, adota-se a perspectiva quantitativa, uma vez que a validação da solução implementada fundamenta-se na mensuração numérica de indicadores de desempenho. Em consonância com Prodanov e Freitas (2013), a abordagem quantitativa recorre a técnicas estatísticas para traduzir o conhecimento em valores observáveis. Neste trabalho, utilizam-se quatro métricas DORA como variáveis dependentes para aferir, de forma objetiva, a velocidade e a estabilidade do processo de entrega.

Em relação aos objetivos, o estudo classifica-se como simultaneamente *exploratório* e *explicativo*. É *exploratório*, pois busca ampliar a familiaridade com a aplicação de *canary deployment* em ambientes de recursos limitados, um contexto ainda pouco sistematizado na literatura. Tal abordagem também alinha-se à definição apresentada por Gil (2002), segundo a qual se caracterizam como exploratórias as pesquisas que visam proporcionar maior familiaridade com o problema ou torná-lo mais explícito. Adicionalmente, configura-se como *explicativo* por investigar a influência da automação do *rollback* sobre as métricas de desempenho, identificando os fatores determinantes para a confiabilidade do sistema (GIL, 2002).

Por fim, quanto aos procedimentos técnicos, a pesquisa caracteriza-se como bibliográfica e quase-experimental. A etapa bibliográfica forneceu o embasamento teórico necessário à concepção e ao desenvolvimento do artefato, enquanto a etapa quase-experimental possibilitou avaliar empiricamente os efeitos da intervenção proposta em ambiente controlado. Conforme Gil (2002), o delineamento quase-experimental caracteriza-se pela manipulação deliberada de uma variável independente sem que haja controle pleno sobre todas as variáveis externas, distinguindo-se do experimento propriamente dito, sobretudo pela ausência de designação aleatória dos sujeitos ou objetos de estudo aos grupos de comparação. Essa classificação mostra-se adequada a este trabalho, uma vez que a intervenção foi aplicada a um único objeto de estudo e seus efeitos foram avaliados a partir da comparação entre condições anteriores e posteriores à sua introdução, sem a constituição de um grupo de controle equivalente executado em paralelo. Conforme Wohlin et al. (2012), estudos conduzidos nessas condições podem produzir evidências empíricas relevantes, desde que sejam estruturados de forma sistemática, com critérios de observação e comparação previamente definidos e com o devido reconhecimento de suas limitações. Nesse sentido, os procedimentos experimentais empregados neste trabalho seguem as diretrizes de experimentação em engenharia de *software* propostas por Wohlin et al. (2012), enquanto os detalhes relativos à configuração dos experimentos, às condições analisadas e aos critérios de avaliação são apresentados nas seções subsequentes.

3.2 Contexto do Estudo e Objeto de Análise

A delimitação do contexto organizacional e do objeto de estudo é essencial para a interpretação dos resultados e para a avaliação da aplicabilidade da solução implementada. Na subseção 3.2.1, caracteriza-se a organização na qual o estudo foi conduzido e seu ambiente de desenvolvimento e operação de software. Na subseção 3.2.2, descreve-se o produto analisado e define-se a unidade de análise. Na subseção 3.2.3, apresentam-se o ambiente-alvo e os ambientes de teste utilizados nos experimentos. Por fim, na subseção 3.2.4, explicitam-se as premissas e restrições que condicionaram a execução do trabalho.

3.2.1 Contexto organizacional

O presente trabalho foi desenvolvido no âmbito do Laboratório de Pesquisa e Capacitação em Software (TerraLAB), vinculado ao Departamento de Computação (DECOM) do Instituto de Ciências Exatas e Biológicas (ICEB) da Universidade Federal de Ouro Preto (UFOP). O laboratório, concebido originalmente em parceria com o Instituto Nacional de Pesquisas Espaciais (INPE), consolidou-se como um ambiente de atuação relevante na articulação entre atividades acadêmicas e demandas do setor produtivo, promovendo a transferência de conhecimento e a formação aplicada em engenharia de software.

O laboratório opera segundo um modelo híbrido que integra ensino, pesquisa e extensão,

orientado à capacitação prática de discentes de graduação e pós-graduação por meio de sua inserção em projetos reais de inovação tecnológica. Nesse sentido, o TerraLAB diferencia-se de ambientes estritamente acadêmicos ao adotar processos e rotinas compatíveis com contextos profissionais, estruturando fluxos de trabalho semelhantes aos de organizações de desenvolvimento de software orientadas à entrega. Essa abordagem permite atender a demandas de parceiros governamentais e do setor privado, preservando, simultaneamente, o compromisso com objetivos formativos e científicos (Universidade Federal de Ouro Preto, 2017; TerraLAB, 2020).

No que se refere à estrutura organizacional, o laboratório adota metodologias ágeis para gestão e desenvolvimento, organizando suas equipes em células multidisciplinares. Essa configuração favorece a divisão de responsabilidades e a colaboração entre perfis técnicos distintos, abrangendo etapas como elicitação e modelagem de requisitos, projeto de arquitetura, implementação, garantia da qualidade e operações de infraestrutura (TerraLAB, 2025). Como resultado, o ambiente institucional simula condições típicas de equipes de desenvolvimento e operação de produtos e serviços de software, incluindo práticas de entrega contínua, com ênfase em ciclos iterativos e integração entre desenvolvimento e validação.

3.2.2 Produto analisado e unidade de análise

O sistema analisado, doravante denominado TerraPlanner[1], consiste em uma plataforma *web* de Sistema de Informação Geográfica (SIG) destinada ao planejamento e à gestão de dados territoriais. A solução foi concebida com o propósito de ampliar o acesso a ferramentas de análise espacial, permitindo que gestores e pesquisadores visualizem, consultem e manipulem informações georreferenciadas, sem a necessidade de instalação local de *software* especializado em geoprocessamento. Do ponto de vista conceitual, o TerraPlanner enquadra-se na categoria de *Web GIS*, ou seja, sistemas de informação geográfica disponibilizados via navegador, que possibilitam a visualização e a análise de dados espaciais em ambiente *web*.

O TerraPlanner adota a separação de responsabilidades em duas camadas principais: o *back-end*, responsável pela lógica de negócio e pela persistência de dados; e o *front-end*, responsável pela interface com o usuário, pela comunicação com o *back-end* e pela lógica de apresentação cartográfica. Neste trabalho, o recorte analítico concentra-se no *pipeline* de implantação do *front-end*. Tal delimitação justifica-se por dois aspectos centrais. Primeiro, o *front-end* constitui a camada de contato imediato com o usuário e, portanto, sua estabilidade visual e funcional é determinante para a usabilidade e para a percepção de confiabilidade da aplicação. Segundo, a validação automatizada dessa camada desempenha um papel estratégico na integridade do sistema como um todo: ao simular interações do usuário e o consumo de dados/serviços, os testes de *front-end* atuam como verificação de integração, contribuindo para a identificação de regressões e para a detecção precoce de falhas no *back-end*, como indisponibilidades e quebras de contrato de API.

[1] Disponível em: <https://terraplanner.org/>. Acesso em: 3 de março de 2026.

3.2.3 Ambiente-alvo e ambientes de teste

Os experimentos e a avaliação da abordagem implementada foram conduzidos considerando três ambientes de implantação, correspondentes a etapas progressivas de validação antes da disponibilização ao usuário final. O ambiente-alvo deste estudo foi o ambiente produtivo, no qual foram realizadas as avaliações finais de viabilidade técnica e segurança operacional do *pipeline* de implantação implementado.

No ambiente-alvo, a aplicação é executada em contêineres distribuídos em Máquinas Virtuais (*Virtual Machines*, VMs) provisionadas em nuvem, com a exposição de tráfego realizada por meio de um *proxy reverso* implementado com o NGINX[2]. A análise buscou assegurar que versões potencialmente instáveis não avançassem para o ambiente-alvo, enquanto versões aprovadas em etapas anteriores fossem implantadas de forma consistente e rastreável.

Os experimentos de validação e de injeção controlada de falhas foram conduzidos prioritariamente em um projeto apartado mantido pelo laboratório, com a finalidade específica de orientar e validar testes de *pipelines* de implantação.

3.2.4 Premissas e restrições

Assumiu-se a disponibilidade de infraestrutura baseada em VMs no *Google Cloud Platform* (GCP), com capacidade moderada, o que impôs a priorização de soluções de baixo custo operacional e, sempre que possível, baseadas em ferramentas de código aberto e/ou auto-hospedadas. Como premissa de viabilidade e de adoção, a solução implementada foi concebida para permanecer aderente ao ecossistema tecnológico vigente, evitando mudanças disruptivas em plataformas e processos. Nesse sentido, considerou-se a utilização do GitLab CI como plataforma de automação, do Docker para conteinerização, de Grafana e PostgreSQL para observabilidade e de um docker registry para armazenamento de artefatos, de modo que a implementação pudesse ser incorporada ao fluxo existente sem migração de infraestrutura.

No período de execução do trabalho, a execução de mudanças no ambiente-alvo esteve condicionada às políticas internas de gestão de mudanças e às permissões operacionais aplicáveis, o que restringiu a realização de experimentos potencialmente disruptivos, isto é, intervenções que pudessem degradar a disponibilidade, a performance ou a integridade do sistema. Ademais, o projeto encontrava-se em fase de *freezing*, ou seja, um período de congelamento de mudanças no qual implantações e alterações eram restritas, permitindo-se apenas intervenções essenciais, com o objetivo de estabilizar o produto. Essa decisão decorria da baixa confiabilidade das implantações, com regressões recorrentes em fluxos previamente estáveis, e da priorização do planejamento e implementação de novas funcionalidades para o *go-live* do produto. Embora existisse um ambiente produtivo disponível, não havia usuários ativos naquele momento, o que afetava diretamente a estratégia de validação do *Canary Deployment*. Consequentemente, a

[2] Documentação disponível em: <https://nginx.org/en/docs/>. Acesso em: 9 fev. 2026.

avaliação experimental dependeu de tráfego sintético e controlado para fins de mensuração e observação do comportamento. Além disso, a estratégia de *rollout* foi planejada para contemplar tanto o cenário pré-*go-live*, caracterizado pela ausência de tráfego real, quanto o cenário pós-*go-live*, no qual haveria tráfego real e requisitos mais estritos de estabilidade.

A execução de parte das intervenções pressupôs a colaboração entre equipes com responsabilidades distintas. Essa dependência organizacional constituiu uma restrição operacional que pôde influenciar a cadência da implementação por fases, exigindo o alinhamento prévio para a execução em janelas e procedimentos definidos pela governança interna. Como medida de mitigação para permitir uma adoção incremental e não bloqueante, considerou-se a adoção de *feature flags*, isto é, chaves de configuração que permitem habilitar ou desabilitar funcionalidades de forma controlada, seletiva e reversível, sem a necessidade de novos *deployments*. Esse e outros mecanismos equivalentes de habilitação gradual visaram desacoplar a entrega de mudanças da sua ativação efetiva em fluxos críticos, garantindo a maturação e a validação em ambiente controlado.

3.3 Estabelecimento da Linha de Base

A caracterização do estado atual do processo de entrega constitui etapa fundamental para a avaliação comparativa da solução implementada. Na subseção 3.3.1, descreve-se o procedimento de caracterização do *pipeline* de implantação vigente, incluindo análise documental, entrevistas com *stakeholders* e medição da cobertura de testes. Na subseção 3.3.2, apresentam-se a fonte de dados e o procedimento de extração das métricas DORA que compuseram a linha de base para comparação.

3.3.1 Caracterização do *pipeline* de implantação vigente

A caracterização do *pipeline* de implantação vigente foi conduzida por triangulação de evidências, combinando (i) análise documental dos arquivos de configuração versionados, (ii) observação direta de execuções registradas na interface do GitLab e (iii) entrevistas semiestruturadas com *stakeholders*. Adicionalmente, foi realizada uma medição pontual da cobertura de testes automatizados para qualificar o grau de validação automatizada atualmente disponível no fluxo de integração contínua.

3.3.1.1 Análise documental e inspeção de configuração

A caracterização do *pipeline* de implantação foi realizada em duas frentes complementares: (i) a inspeção documental do arquivo `.gitlab-ci.yml` do projeto TerraPlanner e (ii) a observação das execuções via UI do GitLab. A análise do arquivo de configuração permitiu identificar estágios, *jobs*, regras condicionais, dependências e artefatos. Para tanto, estabeleceu-se o acesso ao projeto em análise, mantido pela equipe de *front-end*, bem como ao projeto secundário acionado via

Multi-Project Pipeline utilizado na execução de testes E2E, sob responsabilidade da equipe de Garantia de Qualidade (*Quality Assurance*, QA).

Em virtude do período de *freezing* instituído a partir de 2025, analisou-se a última versão em ambiente de produção, implantada em 18 de setembro de 2024, com base no histórico e no registro de execução da UI do GitLab. Para complementar a inspeção documental, observaram-se execuções em dois fluxos ativos: (i) *feature branch* → *development* e (ii) *development* → *staging*. A investigação contemplou o exame de *jobs*, respectivos *logs*, tempos de execução por estágio e intervenções manuais em *gates* de aprovação. O levantamento abrangeu os seguintes elementos: (i) estágios e sequência de execução, com a identificação do encadeamento entre fases; (ii) *jobs* por estágio, incluindo o mapeamento de *scripts* e variáveis utilizadas; (iii) regras condicionais de acionamento por *branch* ou evento; (iv) fluxo de artefatos gerados; (v) integrações externas e mecanismos de bloqueio; e (vi) interação e execução de implantações nas instâncias do GCP.

Por fim, foram analisados os arquivos `docker-compose.yml` de *staging* e produção, obtidos no repositório do projeto, para caracterizar a configuração de *runtime*, detalhando as portas expostas, os volumes e as variáveis de ambiente utilizadas durante a construção da imagem. Também foi observado o sistema de construção de imagens por meio das *tags* geradas durante a execução do pipeline de implantação vigente.

3.3.1.2 Entrevistas semiestruturadas com *stakeholders*

Para validar e complementar as informações obtidas por meio da análise documental, foram realizadas entrevistas semiestruturadas com quatro perfis de *stakeholders*:

1. ***Tech Leads***: responsáveis pela governança do *pipeline* de implantação, aprovação de *Merge Requests* (MRs) e decisões sobre *rollback*. As entrevistas focaram em políticas de promoção entre ambientes, critérios de aprovação e procedimentos de resposta a incidentes.

2. ***Equipe de QA***: responsável pela manutenção da suíte de testes E2E e pelas validações manuais periódicas em ambiente de homologação. A entrevista abordou o estado atual do *job* `test-e2e`, os motivos da suspensão e os planos de reativação.

3. ***Desenvolvedores***: membros ativos da equipe de desenvolvimento do TerraPlanner e Infraestrutura. As entrevistas exploraram a percepção sobre gargalos no fluxo de entrega, frequência de *rollbacks* e tempo típico de resposta a incidentes. Adicionalmente, as entrevistas realizadas com o time de infraestrutura abordaram assuntos como a estrutura e as especificações das instâncias de hospedagem no GCP e os projetos responsáveis pelo *proxy* de requisições.

4. ***Egressos e membros antigos***: colaboradores que atuaram no projeto em períodos anteriores. Foi necessário realizar reuniões com membros antigos do TerraLAB devido à alta

rotatividade característica de seu perfil profissionalizante em ambiente universitário. Determinados detalhes sobre o fluxo só puderam ser esclarecidos por meio do contato com esses membros.

As entrevistas foram realizadas remotamente, via *Google Meet*, no período de novembro de 2025 a janeiro de 2026, com duração média de 30 minutos. Os participantes foram informados sobre os objetivos da pesquisa, e as respostas foram registradas em notas estruturadas, posteriormente consolidadas em um documento de síntese utilizado na redação desta pesquisa. Foram entrevistados 5 participantes no total, escolhidos com base no conhecimento sobre o produto em análise.

3.3.1.3 Medição da cobertura de testes automatizados

A cobertura de testes executados pelo *job* `test_integration` foi medida por meio da execução do comando `jest -coverage` em ambiente de desenvolvimento local. A medição foi realizada com as seguintes especificações: sistema operacional Windows 11 Pro 22H2, Node.js versão `v22.15.0`, Jest versão `v29.7.0` e npm versão `v10.9.2`.

A implementação de um *script* para execução local do relatório de cobertura foi necessária devido à ausência desse recurso no projeto. O relatório foi configurado para considerar, de forma geral, os arquivos que concentram a lógica principal da aplicação, excluindo componentes que normalmente não representam comportamento passível de teste. Assim, foram desconsiderados, por exemplo, arquivos exclusivamente de definição e declaração, inicialização e ponto de entrada, módulos de agregação e reexportação, estilos visuais, configurações, listas fixas de valores, estruturas de tipos e contratos, material de teste e demonstração, recursos estáticos e dados puramente estáticos. O relatório gerado foi armazenado no diretório `coverage/` e, posteriormente, analisado para extração das métricas de *statements*, *branches*, *functions* e *lines*.

3.3.2 Fonte de dados e procedimento de extração da linha base

Para classificar o desempenho da organização conforme os arquétipos estabelecidos pelo relatório DeBellis et al. (2024), realizou-se o levantamento dos dados históricos de implantação. Essa extração foi realizada por meio da API do GitLab[3], que fornece acesso programático a informações sobre *pipelines*, *jobs*, *commits*, MRs e *deployments*.

Para automatizar a extração, também foi desenvolvido um *script* em Python utilizando a biblioteca `python-gitlab`, que realizou as seguintes operações: (i) autenticação via *token* de acesso pessoal com permissões de leitura; (ii) consulta aos *endpoints* de *pipelines* e *deployments* do projeto; (iii) filtragem de execuções associadas à *branch* `main`; (iv) extração de *timestamps* de início e término de *jobs*, *status* de execução e metadados de *commits*; e (v) persistência dos dados em formato JSON estruturado para análise posterior.

[3] Documentação disponível em: <https://docs.gitlab.com/ee/api/>. Acesso em: 10 fev. 2026.

O intervalo de análise compreende a totalidade das implantações realizadas entre janeiro e dezembro de 2024, configurando o período anual mais recente antes do *freezing* das operações. A opção por um recorte temporal de doze meses permite mitigar vieses sazonais, tais como os períodos de férias dos integrantes do TerraLAB e as janelas destinadas exclusivamente ao planejamento.

As métricas DF e LTFC foram calculadas diretamente a partir dos dados extraídos via API, utilizando os *timestamps* de criação e conclusão de *pipelines* e MRs. Por sua vez, as métricas CFR e MTTR exigiram análise qualitativa complementar, uma vez que a identificação de falhas em produção e a caracterização de *hotfixes* não são capturadas de forma estruturada pelo sistema. Para essas métricas, foi realizada inspeção manual do histórico de implantações na UI do GitLab, considerando os seguintes indicadores: (i) *pipelines* de implantação que falharam no estágio de *deployment*; (ii) implantações consecutivas realizadas em intervalo inferior a quatro horas; (iii) MRs rotulados com *labels* de correção (por exemplo, `hotfix`, `fix`); e (iv) mensagens de *commit* que indiquem reversão ou correção emergencial. A triangulação desses indicadores permitiu inferir com maior precisão os eventos de falha e o tempo de recuperação associado.

3.4 Projeto experimental de implantação

Este capítulo apresenta o projeto experimental da abordagem implementada. Na subseção 3.4.1, definem-se os *gates* automatizados de qualidade e segurança e suas evidências. Em seguida, na subseção 3.4.2, descreve-se a política de *rollback* automatizado baseada em sinais observáveis durante o *rollout*. Na sequência, na subseção 3.4.3, detalha-se a estratégia metodológica de *canary deployment*, incluindo modalidades de roteamento e critérios de progressão e interrupção. Por fim, apresenta-se a estratégia incremental e não bloqueante adotada para a instrumentação de evidências que viabilizam a comparação entre cenários.

3.4.1 Critérios metodológicos para *gates* de qualidade e segurança

O *pipeline* de implantação implementado incorpora *gates* automatizados de qualidade e segurança, definidos como verificações executadas antes da integração e/ou promoção entre ambientes, cujo resultado pode bloquear a progressão do fluxo ou apenas registrar evidências para acompanhamento. O primeiro *gate* corresponde à *análise estática de código*, voltada à detecção de inconsistências de estilo e potenciais problemas de manutenção. Por se tratar de uma verificação determinística e de baixo custo de correção, sua reprovação é tratada como *condição bloqueante* para integração e progressão do *pipeline*. O segundo *gate* corresponde à *auditoria automatizada de dependências*, com classificação de severidade baseada em métricas padronizadas, como o *Common Vulnerability Scoring System* (CVSS)[4]. Por critério de governança e viabilidade operacional, apenas achados no nível mais crítico são *bloqueantes* para a integração;

[4] Disponível em: <https://www.first.org/cvss/>. Acesso em: 15 de Janeiro de 2026.

vulnerabilidades de severidade inferior são registradas como *alertas rastreáveis*, sem interromper o fluxo, a fim de evitar paralisações indevidas e permitir a priorização incremental de correções. O terceiro *gate* consiste na *medição da cobertura de testes automatizados*, produzindo indicadores de linhas, ramificações, funções e instruções. Nesta pesquisa, a cobertura é utilizada primariamente como um *gate* não bloqueante, gerando evidências estruturadas para o acompanhamento temporal da maturidade dos testes. Caso necessário, sua conversão para *gate* bloqueante pode ser feita de forma gradual, em conformidade com a estratégia incremental e não bloqueante definida neste capítulo.

3.4.2 Política de *rollback* automatizado

O mecanismo de *rollback* automatizado é definido como uma política de recuperação orientada por evidências, acionada quando sinais observáveis indicam degradação durante ou após o *Canary Deployment*. Neste trabalho, tais sinais correspondem a anomalias mensuráveis em Indicadores de Nível de Serviço (*Service Level Indicator*s, SLIs) (por exemplo, aumento da taxa de erro, regressão da latência e queda de disponibilidade).

A decisão de reversão é fundamentada em validações de SLIs que avaliam o comportamento recente da aplicação com base em requisições direcionadas à mesma, consolidando indicadores como latência média e taxas de sucesso e de erro, produzindo um status final a partir da análise desses dados. A segunda fonte de evidência corresponde às validações automatizadas executadas no *pipeline* de implantação, incluindo testes de integração e, posteriormente, os testes E2E. Além disso, falhas explícitas em *jobs* críticos do processo de implantação são consideradas como sinal suficiente para a interrupção do *rollout*.

Dois cenários operacionais são considerados. No Cenário A, a reversão é acionada durante a progressão do *canary*, antes de sua promoção total. Nessa condição, o *rollback* tem como efeito abortar o *rollout* e redirecionar integralmente o tráfego para a versão estável previamente em execução, evitando que o artefato defeituoso seja promovido a `stable`. No Cenário B, a reversão é acionada após a promoção da nova versão para `stable`, isto é, quando a versão recém-implantada já está atendendo integralmente ao tráfego. Nesse caso, o mecanismo restaura a última versão saudável conhecida, apoiando-se em um registro confiável do histórico de implantações, que permite identificar o artefato anterior a ser restaurado. Embora o *rollback* automatizado seja o mecanismo primário de recuperação investigado neste trabalho, mantém-se a possibilidade de acionamento manual como alternativa de contingência. Tal alternativa é aplicável em situações em que a automação não esteja disponível, apresente comportamento ambíguo ou requeira intervenção sob governança operacional.

3.4.3 Estratégia de *Canary Deployment* em nível metodológico

Neste trabalho, a estratégia de *Canary Deployment* é definida como um mecanismo de entrega progressiva, no qual uma nova versão é exposta gradualmente ao tráfego antes de sua promoção completa. As decisões de avanço são condicionadas a evidências verificáveis de saúde e confiabilidade. Essa configuração buscou reduzir o raio de impacto das mudanças, favorecer a detecção precoce de regressões e estabelecer um protocolo reprodutível de comparação entre o *pipeline* de implantação vigente e o *pipeline* de implantação implementado.

A exposição gradual é operacionalizada por meio de duas modalidades de roteamento, selecionadas em função do contexto operacional do produto. No cenário pré-*go-live*, caracterizado pela ausência de usuários ativos e pela validação predominantemente interna, é empregada a modalidade de lista de permissão (doravante denominada *allowlist*), na qual apenas um subconjunto controlado de usuários (por exemplo, membros do TerraLAB e equipes de QA) é direcionado à versão *canary*. No cenário pós-*go-live*, com a presença de tráfego real e a necessidade de validação sob carga representativa, é empregada a modalidade de roteamento progressivo, na qual a fração de tráfego destinada ao contêiner *canary* é ampliada em estágios até a promoção total. As duas modalidades podem ser mantidas em paralelo, uma vez que não são mutuamente exclusivas; nesse caso, é atribuída preferência ao cookie em relação à *allowlist*, quando o cookie é fornecido.

A progressão do *canary* é estruturada em estágios, cada qual composto por (i) aplicação da política de roteamento correspondente e (ii) um período de observação, no qual evidências são coletadas. No modo de roteamento progressivo, cada estágio corresponde a um percentual predefinido de tráfego direcionado ao *canary*, evoluindo de forma incremental até a promoção total. São monitorados SLIs críticos, em especial a taxa de erro e a latência de resposta. Para cada estágio, são estabelecidos critérios de aceitação e janelas de observação. A violação desses critérios implica a interrupção da progressão do *rollout* e o corte do tráfego direcionado ao *canary*.

3.5 Métricas e critérios de avaliação

A validação da solução implementada fundamenta-se em critérios objetivos e mensuráveis. Na subseção 3.5.1, estabelecem-se as definições operacionais e os procedimentos de cálculo das métricas DORA, utilizadas para caracterizar a linha de base e avaliar o impacto do *pipeline* implementado. Na subseção 3.5.2, definem-se os limiares de SLIs que orientam as decisões automatizadas de progressão, interrupção e reversão durante o *Canary Deployment*.

3.5.1 Métricas DORA

As métricas DORA constituem os indicadores centrais para a avaliação do desempenho do processo de entrega. Para assegurar a reprodutibilidade e comparabilidade entre a linha de base e o cenário automatizado, são estabelecidas definições operacionais precisas para cada métrica, alinhadas às recomendações de Forsgren, Humble e Kim (2018) e adaptadas ao contexto do projeto analisado.

3.5.1.1 Frequência de Implantação (*Deployment Frequency*)

- *Definição operacional:* uma implantação é contabilizada quando uma nova versão é promovida com sucesso para o ambiente de produção e se torna a versão `stable`.
- *Procedimento de cálculo:*

$$DF = \frac{D}{t} \tag{3.1}$$

onde D é o número total de implantações bem-sucedidas registradas e t é a duração do período de observação. A razão entre essas grandezas expressa a frequência média de implantações, quantificando a cadência de entregas em produção.

3.5.1.2 Tempo de Entrega de Mudanças (*Lead Time for Changes*)

- *Definição operacional:* para cada implantação bem-sucedida j, identifica-se o conjunto de mudanças entregues como os *commits* no intervalo entre a última implantação bem-sucedida e a implantação corrente[5]. O tempo de entrega da implantação j é calculado a partir do *commit* mais antigo desse conjunto, utilizado como aproximação do início da mudança entregue.
- *Procedimento de cálculo por implantação:*

$$LTFC_j = T_j^{deploy} - \min(T_j^{commit}) \tag{3.2}$$

onde T_j^{deploy} é o instante em que a implantação j foi concluída em ambiente de produção e $\min(T_j^{commit})$ é o marcador temporal do *commit* mais antigo dentre as mudanças entregues na implantação j. A diferença entre essas grandezas expressa o tempo de entrega da mudança.

- *Agregação no período:*

$$LTFC = \frac{\sum_{j=1}^{D} LTFC_j}{D} \tag{3.3}$$

onde D é o número de implantações bem-sucedidas no período analisado e $LTFC_j$ é o tempo de entrega da implantação j, conforme a Equação 3.2. Assim, $LTFC$ corresponde à média dos tempos de entrega por implantação no período.

[5] Práticas como *squash merge*, que consolidam múltiplos *commits* em um único *commit* de fusão, podem reduzir a granularidade da análise. Nesses casos, a unidade de mudança passa a ser o *merge commit*.

3.5.1.3 Taxa de Falha de Mudanças (*Change Failure Rate*)

- *Definição operacional:* considera-se falha de mudança quando uma implantação está associada a um evento de remediação atribuível à própria mudança entregue. São contabilizadas três situações: (i) reversão automática durante o *canary* por violação de SLIs ou falha em validações críticas; (ii) reversão manual após promoção total, quando o incidente é identificado após a versão tornar-se `stable`; e (iii) aplicação de correção emergencial dentro de uma janela operacional pré definida após a implantação, mantida constante ao longo do estudo para fins de comparabilidade.

- *Procedimento de cálculo:*

$$CFR = \frac{N_{failed}}{N_{total}} \times 100\% \tag{3.4}$$

onde N_{failed} é o número de implantações associadas a evento de falha/remediação e N_{total} é o número total de implantações realizadas no período. A razão entre essas grandezas, multiplicada por 100, expressa o percentual de mudanças que causaram degradação no serviço.

3.5.1.4 Tempo Médio de Recuperação (*Mean Time to Recovery* – MTTR)

- *Definição operacional:* a duração do incidente é medida desde o instante de detecção até o instante de recuperação completa do serviço. A detecção pode ocorrer por mecanismos automatizados ou por identificação manual durante a validação operacional. A recuperação é caracterizada pela restauração do atendimento em condição considerada saudável.

- *Procedimento de cálculo:*

$$MTTR = \frac{\sum_{i=1}^{I}(T_i^{recovery} - T_i^{detect})}{I} \tag{3.5}$$

onde T_i^{detect} é o instante de detecção do incidente i, $T_i^{recovery}$ é o instante de recuperação completa do serviço, e I é o número total de incidentes registrados no período. A razão entre a soma das durações individuais e o número de incidentes expressa o tempo médio necessário para restaurar o serviço após falhas.

3.5.2 Limiares de SLI

Os limiares de SLI definem os limites aceitáveis para o comportamento da aplicação em produção, orientando o *pipeline* na detecção de degradações e no acionamento de *rollback*. Estabelecem-se, a seguir, as definições operacionais e os procedimentos de cálculo para a taxa de erro, latência e disponibilidade.

3.5.2.1 Latência das Requisições

- *Definição operacional:* define-se a latência como o tempo total de resposta percebido durante o uso da aplicação em ambiente de produção.
- *Procedimento de cálculo:* a fração de requisições que atendem a um limiar de latência T na janela de observação W é dada por:

$$S_T(W) = \frac{N_{\leq T}(W)}{N_{\text{ok}}(W)}, \tag{3.6}$$

  onde $N_{\text{ok}}(W)$ denota o número de requisições bem-sucedidas consideradas na janela de tempo W e $N_{\leq T}(W)$ denota o número de requisições, dentre as consideradas, cuja latência satisfaz $\ell_i \leq T$. Para cada requisição i, a latência é definida por $\ell_i = t_i^{\text{fim}} - t_i^{\text{início}}$.
- *Limiar inicial:* adotam-se dois objetivos de desempenho, conforme exemplificado por Beyer et al. (2018): (i) $S_{400\,\text{ms}}(W) \geq 0.90$, isto é, ao menos 90% das requisições com $\ell_i \leq 400$ ms; e (ii) $S_{850\,\text{ms}}(W) \geq 0.99$, isto é, ao menos 99% das requisições com $\ell_i \leq 850$ ms. No entanto, foi desenvolvido um mecanismo que torna esse limiar configurável. Para viabilizar a execução dos testes e a maturação inicial do *pipeline*, adotou-se, nesta fase, um limiar mais permissivo; o padrão de referência para a métrica permanece o definido pelos objetivos de desempenho citados.

3.5.2.2 Disponibilidade

- *Definição operacional:* define-se a disponibilidade como a proporção de verificações de saúde bem-sucedidas em ambiente de produção. Considera-se uma verificação como **sucesso** quando o *endpoint* de saúde retorna código HTTP `2XX` e indica estado "saudável" para todos os serviços avaliados. Respostas HTTP `5XX`, bem como a ausência de resposta, são contabilizadas como *falhas*.
- *Procedimento de cálculo:*

$$A(W) = \frac{N_{\text{ok}}(W)}{N_{\text{hc}}(W)}, \tag{3.7}$$

  onde W é a janela de observação; $N_{\text{hc}}(W)$ é o número total de verificações de saúde executadas em W; e $N_{\text{ok}}(W)$ é o número de verificações de saúde bem-sucedidas em W, conforme o critério definido no item anterior.
- *Limiar inicial:* adota-se como referência o SLO de 99% de disponibilidade discutido por Beyer et al. (2018), estabelecendo-se o objetivo $A(W) \geq 0.99$. No entanto, foi desenvolvido um mecanismo que torna esse limiar configurável.

3.6 Experimentos controlados de injeção de falhas

A validação experimental do *pipeline* de implantação implementado foi conduzida por meio de injeção controlada de falhas, técnica que permite avaliar a capacidade de detecção e recuperação automatizada em condições adversas representativas de cenários reais de degradação. Foram injetadas três categorias de falhas: falhas de taxa de erro, caracterizadas pela introdução de exceções não tratadas em rotas críticas que resultam em respostas HTTP 500; falhas de latência, simuladas por meio de atrasos artificiais de 3 segundos em operações de *backend*; e falhas de disponibilidade, provocadas por *crashes* intermitentes do contêiner que simulam instabilidade de *runtime*. Cada categoria de falha foi injetada em uma versão *canary* dedicada, permitindo a observação isolada do comportamento do mecanismo de detecção e recuperação.

Três cenários experimentais foram conduzidos para avaliar aspectos distintos do mecanismo implementado. O primeiro cenário avaliou a detecção e o *rollback* automático durante a progressão do *canary*, verificando se o *pipeline* de implantação identifica falhas injetadas e aciona a reversão antes que a versão defeituosa atinja 100% do tráfego; como cada uma das três categorias de falha descritas anteriormente foi injetada e avaliada de forma isolada, esse cenário resultou em três execuções distintas, denominadas Cenários 1a, 1b e 1c nos resultados apresentados no Capítulo 4. O segundo cenário validou a promoção bem-sucedida de versões sem falhas, assegurando que o mecanismo não produza falsos positivos e permita a progressão completa até a promoção para `stable` sem intervenção manual. O terceiro cenário avaliou o *rollback* pós-promoção, verificando se falhas detectadas após a versão tornar-se `stable` acionam corretamente a reversão para a versão anterior, conforme descrito no Cenário B da política de recuperação. Em conjunto, esses três cenários conceituais totalizaram cinco execuções experimentais.

O critério de sucesso da validação foi definido pelo atendimento a parâmetros objetivos e mensuráveis. Adotou-se como critério que 100% das falhas injetadas fossem detectadas pelo mecanismo de validação de SLI antes de atingir 100% do tráfego, que 100% dos *rollbacks* fossem executados automaticamente e restaurassem o serviço ao estado estável anterior em tempo inferior a 2 minutos, e que 100% das versões sem falhas fossem promovidas para `stable` sem falsos positivos. Adicionalmente, a avaliação considerou como indicativo de efetividade a redução mensurável no MTTR e no CFR em comparação com a linha de base, bem como a manutenção ou aumento da DF e redução do LTFC.

4 Resultados e Discussão

O presente capítulo dedica-se à apresentação dos resultados obtidos neste trabalho, consolidando a análise do contexto organizacional vigente, a concepção e a implementação da solução proposta e a validação experimental e quantitativa de seus mecanismos. O capítulo abrange, de forma integral: (i) a caracterização sistemática do *pipeline* de implantação vigente, identificando seus componentes, fluxos operacionais e pontos críticos; (ii) a definição arquitetural e a implementação do *pipeline* de implantação proposto, materializando a solução de *Canary Deployment* com *rollback* automatizado; e (iii) a validação experimental controlada e a comparação quantitativa entre a linha de base e o *pipeline* implementado, por meio das métricas DORA.

Para organizar a apresentação desses resultados, o capítulo estrutura-se em oito seções complementares. Inicialmente, a seção 4.1 apresenta o mapeamento sistemático do *pipeline* de implantação vigente, detalhando a arquitetura de entrega atual e expondo gargalos operacionais, débitos técnicos e fontes de risco que comprometem tanto a velocidade quanto a estabilidade do processo; em seguida, a seção 4.2 discute criticamente esses achados. Na sequência, a seção 4.3 quantifica o desempenho vigente por meio das métricas DORA, estabelecendo a linha de base para comparação. A seção 4.4 formaliza a arquitetura conceitual do *pipeline* proposto, e a seção 4.5 detalha sua implementação. A seção 4.6 descreve o protocolo de experimentos controlados de injeção de falhas, cujos resultados são apresentados na seção 4.7. Por fim, a seção 4.8 compara quantitativamente a linha de base e o *pipeline* de implantação proposto por meio das quatro métricas DORA.

4.1 Mapeamento do Pipeline de Implantação vigente

Esta seção apresenta o mapeamento sistemático do *pipeline* de implantação vigente, caracterizando seus componentes, fluxos operacionais e mecanismos de controle de qualidade. A subseção 4.1.1 descreve a organização em estágios sequenciais e os *jobs* que compõem cada etapa do ciclo de entrega. A subseção 4.1.2 especifica os gatilhos de execução e os modos de acionamento do *pipeline* CI/CD, diferenciando rotinas de validação e de entrega. A subseção 4.1.3 caracteriza a infraestrutura de execução na GCP, incluindo o provisionamento de ambientes e o ciclo de atualização via contêineres Docker. A subseção 4.1.4 consolida o fluxo operacional de ponta a ponta. A subseção 4.1.5 aprofunda a análise dos principais *jobs* de teste automatizado. Por fim, a seção 4.2 procede à análise crítica do *pipeline* vigente.

4.1.1 Gatilhos de execução e estágios

No GitLab CI, um *job* encapsula instruções e parâmetros executados em um estágio do *pipeline*, podendo gerar artefatos intermediários consumidos nas etapas subsequentes. A execução é realizada por *runners*, responsáveis por prover recursos computacionais e disparar as tarefas quando as condições do *pipeline* são atendidas. No cenário vigente, o fluxo é organizado em quatro estágios sequenciais — *build*, *test*, *version* e *deploy* — que operam como portões de qualidade (*quality gates*), pois a promoção do artefato depende do êxito nas verificações previstas em cada etapa.

O estágio *build* constrói o artefato a ser promovido, materializado como uma imagem Docker gerada a partir do código-fonte e publicada em um registro privado de imagens. O estágio *test* concentra validações automatizadas de caráter bloqueante, incluindo execuções em ambiente isolado com injeção controlada de variáveis de ambiente e dependências; a validação sistêmica E2E, quando aplicável, pode ser delegada a um projeto dedicado de testes e complementada por validações manuais de aceitação. Em seguida, o estágio *version* aplica versionamento semântico automatizado para rastrear as versões por ambiente, inferindo incrementos a partir do histórico de mudanças e gerando etiquetas específicas para cada contexto.

Por fim, o estágio *deploy* atualiza o ambiente-alvo ao consumir a imagem publicada no registro privado e aplicá-la na infraestrutura de execução, adotando a estratégia *Big Bang Deployment*. A promoção das mudanças segue três ambientes lógicos: *development*, focado na integração contínua na branch `development` sem publicação de ambiente; *staging*, destinado à homologação pré-produção com alta paridade operacional e ênfase em testes E2E e de integração, além de aceitação manual quando aplicável; e *main*, que representa a produção estável, recebendo apenas versões promovidas após critérios de qualidade e aprovações, preservando rastreabilidade e consistência entre validação e liberação.

4.1.2 Gatilhos e modos de execução

A execução do *pipeline* de implantação vigente é orientada a eventos e estruturada em dois modos complementares de acionamento, ambos vinculados ao processo de revisão e integração de código via MR. Em termos práticos, esses gatilhos determinam se o *runner* executará rotinas de *validação*, focadas em verificações automatizadas de caráter bloqueante, ou rotinas de *entrega*, que materializam a construção, o versionamento e a implantação nos ambientes publicados:

1. ***Pré-integração (MR)***: acionado quando uma alteração é submetida por meio de um MR. Nessa condição, o *pipeline* de implantação contínua é executado em modo de verificação, priorizando validações automatizadas de caráter bloqueante. A fusão do código é condicionada ao sucesso das verificações e à aprovação por pares.

2. ***Pós-integração (push/merge)***: disparado após a aprovação do MR e a fusão do código

em *branches* estáveis. Nessa fase, o *pipeline* de implantação executa seu ciclo completo, contemplando a preparação determinística de configurações de ambiente, a validação sistêmica E2E, o versionamento semântico e a atualização do artefato implantado no ambiente-alvo.

Esses gatilhos se repetem nas transições relevantes do fluxo de entrega: (i) *feature branch* → *development*, (ii) *development* → *staging* e (iii) *staging* → *main*. Nesse arranjo, os MRs implementam pontos de controle que segregam integração, validação e liberação, garantindo que apenas mudanças verificadas e aprovadas sejam promovidas para os ambientes efetivamente publicados.

4.1.3 Infraestrutura de execução do *pipeline* de implantação

Os ambientes do projeto em análise são executados, atualmente, em instâncias na GCP[1], plataforma de computação em nuvem do Google que oferece serviços de provisionamento e operação de infraestrutura (por exemplo, máquinas virtuais, redes, armazenamento e recursos de segurança). Do ponto de vista operacional, cada ambiente é provisionado e mantido em uma instância dedicada, utilizada como unidade administrativa para a alocação de recursos de computação, rede e armazenamento.

Nesse arranjo, a aplicação não é executada diretamente na instância de forma nativa; em vez disso, o ambiente executa contêineres Docker correspondentes à versão do serviço em execução. Assim, o ciclo de atualização consiste em obter a imagem do serviço a partir de um *Docker Registry* privado e, em seguida, criar ou recriar o contêiner com base nessa imagem, colocando a versão atualizada em execução. Além do *front-end* do TerraPlanner, o *pipeline* de implantação vigente depende de componentes auxiliares executados em instâncias na GCP, incluindo o *runner* do GitLab responsável pela execução dos *jobs* o registro privado de contêineres, responsável por armazenar as imagens Docker.

4.1.4 Fluxo operacional ponta a ponta: do requisito à entrega

Com as definições e processos estabelecidos, a execução operacional do ciclo de entrega é particionada em três *fluxos de implantação*: (i) *Feature Branch* → *development*; (ii) *development* → *staging*; e (iii) *staging* → *main*.

4.1.4.1 Fluxo I — *feature branch* → *development*

***Gatilho de Pré-integração (MR)*:** O trabalho é isolado em uma *feature branch* derivada de *development*. A abertura de um MR para a *branch development* aciona o *pipeline* de CI em modo de verificação, executando o *job* `test_integration`; eventuais falhas bloqueiam a

[1] Disponível em: <https://docs.cloud.google.com>. Acesso em: 15 jan. 2026.

aprovação e, consequentemente, o *merge*. Em paralelo, a revisão por pares fornece validação semântica das alterações.

***Gatilho de Pós-integração (push/merge)*:** Após validação, aprovação e o *merge* do MR em *development*, o *job* `create-env` consolida as variáveis de ambiente no arquivo `.env`, padronizando a configuração utilizada nas etapas subsequentes. Além disso, o *job* `test-e2e` aciona o *pipeline* de um projeto externo dedicado a testes E2E. Esse fluxo é sintetizado na Figura 4.1.

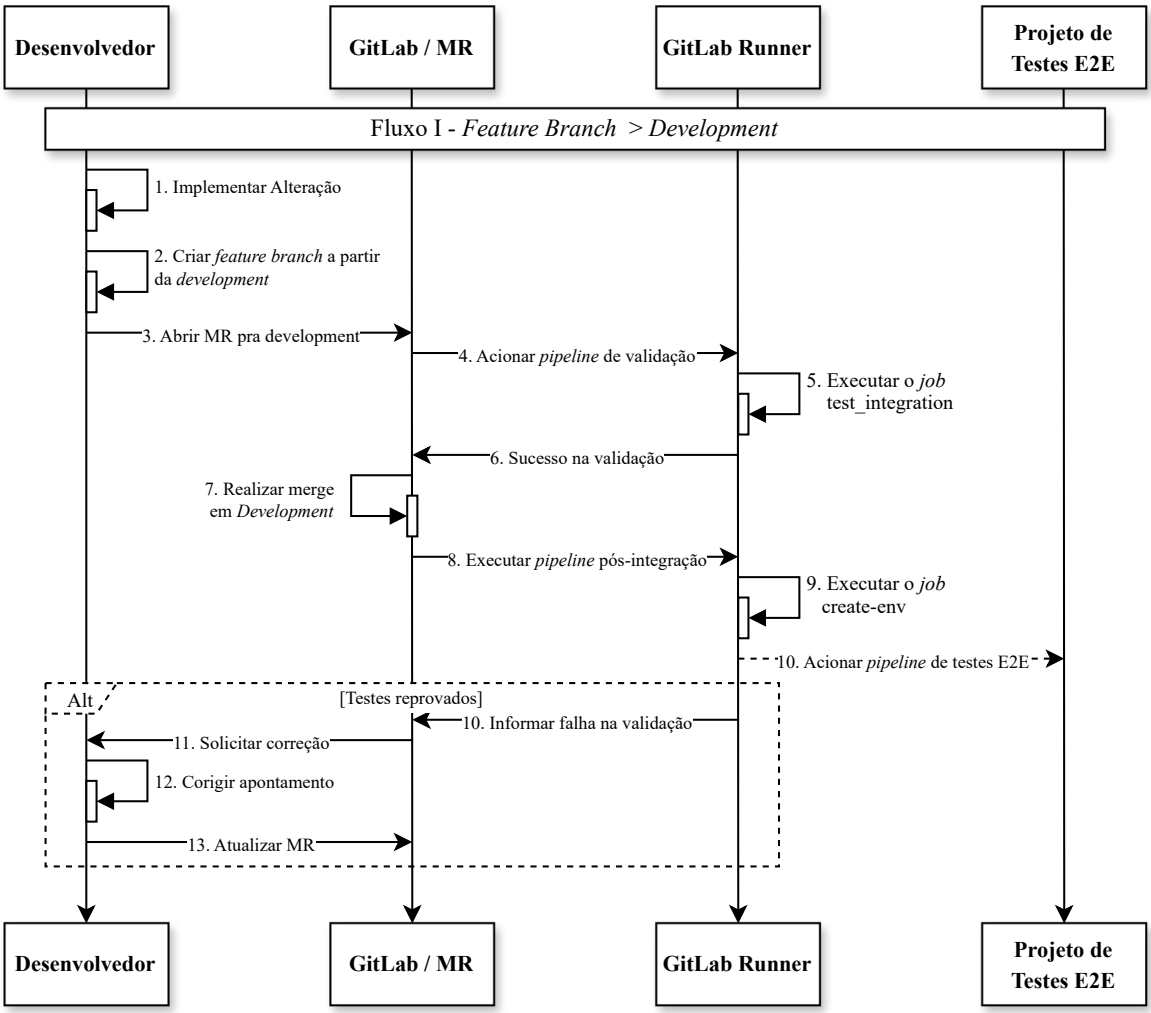

Figura 4.1 – Diagrama de sequência do Fluxo I do *pipeline* de implantação vigente

Fonte: Elaborado pelo autor (2026).

4.1.4.2 Fluxo II — *development* → *staging*

***Gatilho de Pré-integração (MR)*:** Abre-se um MR de *development* para *staging*. O MR executa o *pipeline* de integração contínua em modo de verificação, com testes bloqueantes e requer aprovação por pares.

***Gatilho de Pós-integração (push/merge)*:** Após a fusão do MR, o *push* em *staging* aciona o *pipeline* completo do ambiente-alvo. O *job* `create-env` provisiona, de forma segura, as variáveis de ambiente a partir de variáveis protegidas. Na sequência, o *job* `test-e2e` delega a validação sistêmica E2E a um repositório dedicado. Em seguida, o *job* `semver_tag_staging` infere o incremento de versão a partir do histórico de *commits*, em conformidade com o padrão Conventional Commits[2], e gera, em *staging*, uma *tag* de *release candidate* com sufixo incremental (por exemplo, `-rc.1`). O *job* `delete-old-image-staging` remove a *tag* associada à versão anterior do ambiente no registro privado. Por fim, o *job* `push-image-staging` constrói uma imagem de contêiner imutável a partir do código validado e a publica com a marcação apropriada ao ambiente; e o *job* `deploy_staging` atualiza a aplicação no servidor-alvo por meio de operações remotas, contemplando (i) a propagação do `docker-compose` atualizado, (ii) a autenticação no registro, o *pull* da nova imagem e a recriação dos serviços, e (iii) saneamento pós-implantação, removendo contêineres parados, redes não utilizadas e imagens órfãs para liberação de recursos computacionais. Esse fluxo é sintetizado na Figura 4.2.

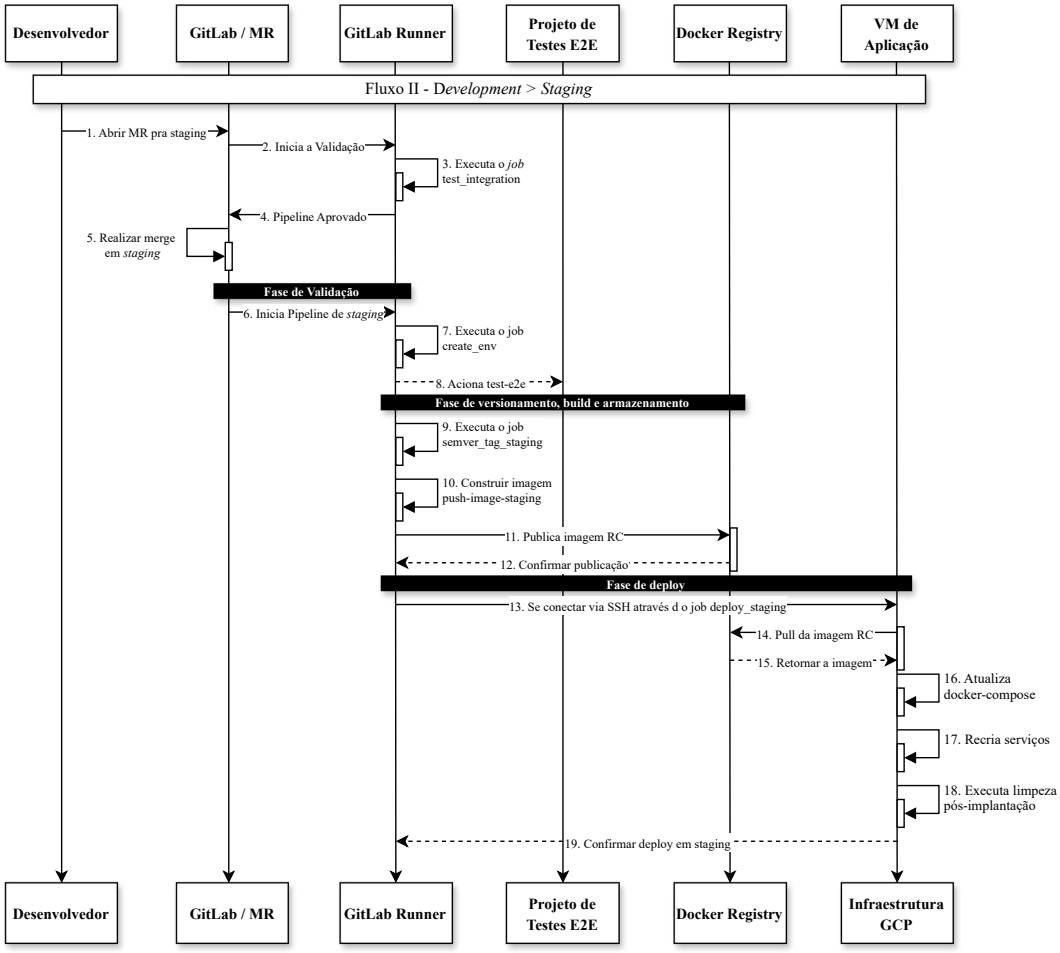

Figura 4.2 – Diagrama de sequência do Fluxo II do *pipeline* de implantação vigente

Fonte: Elaborado pelo autor (2026).

[2] Documentação disponível em: <https://www.conventionalcommits.org/en/v1.0.0/>. Acesso em: 15 fev. 2026.

4.1.4.3 Fluxo III — *staging* → *main*

***Gatilho de Pré-integração (MR)*:** A liberação ao usuário final inicia-se por meio de um MR de *staging* para *main*, mantendo-se o critério de aprovação por pares antes da fusão. Nesta etapa, o *job* `test_integration` não é executado.

***Gatilho de Pós-integração (push/merge)*:** Após a aprovação e fusão em *main*, executa-se um *pipeline* de implantação análogo ao de *staging*, com duas diferenças centrais: (i) o versionamento final é realizado pelo *job* `semver_tag_main`, sem o sufixo `-rc.N`, caracterizando uma versão estável; e (ii) a imagem é publicada com a *tag stable*, após a remoção da marcação estável anterior por meio do `delete-old-image-production`. Em seguida, o *job* `push-image-production` publica o novo artefato, e o *job* `deploy-production` atualiza a instância de produção por meio do `docker-compose`, apontando para a nova versão. Por fim, realiza-se o saneamento pós-implantação. Esse fluxo é sintetizado na Figura 4.3.

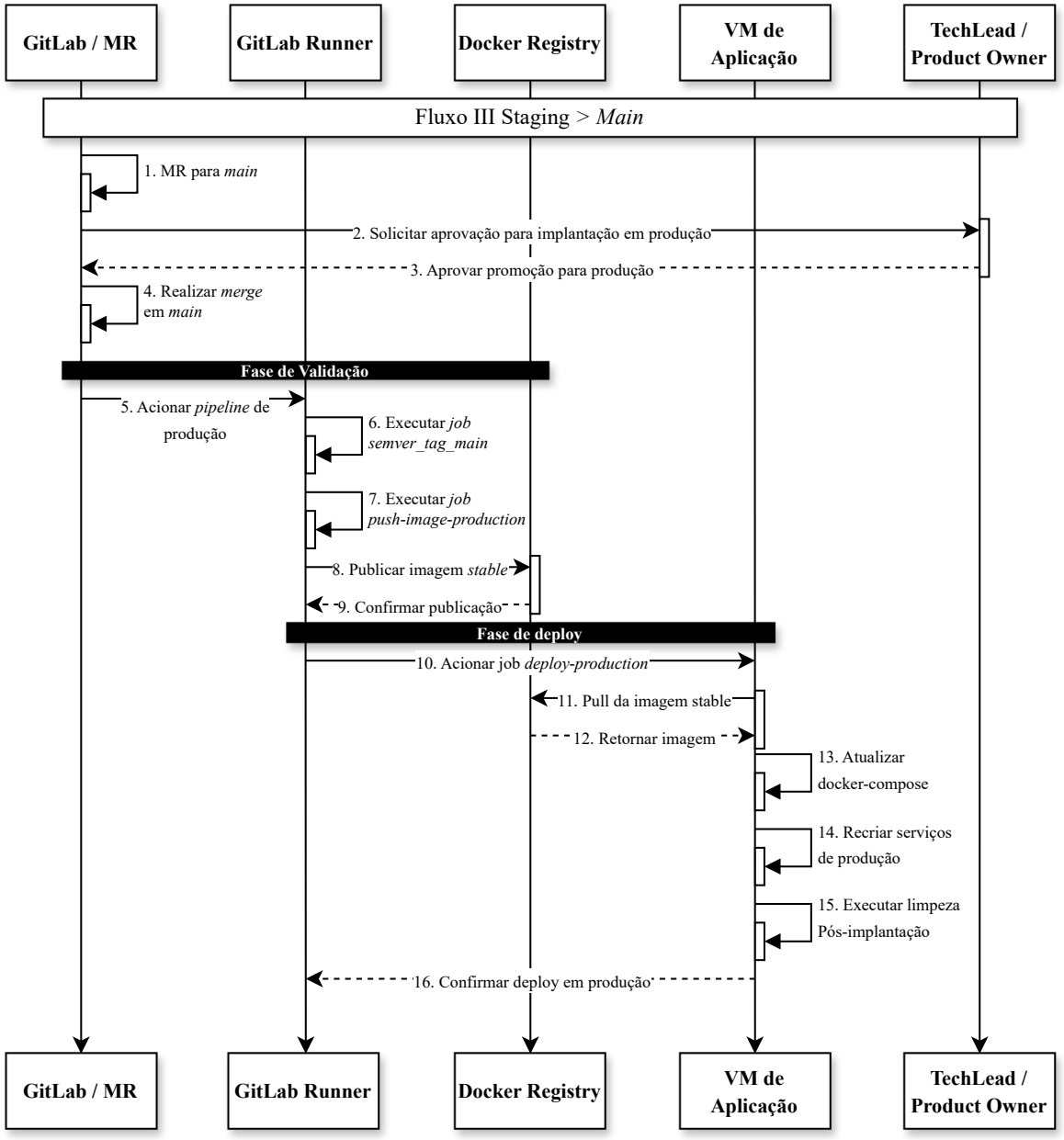

Figura 4.3 – Diagrama de sequência do Fluxo III do *pipeline* de implantação vigente

Fonte: Elaborado pelo autor (2026).

4.1.4.4 Fluxo IV — Processo de *rollback*

***Política institucional*:** A política institucional vigente para o tratamento de incidentes adota a estratégia denominada *roll forward*. Esse mecanismo é acionado manualmente por meio da UI do próprio GitLab. Nessa interface, especificamente nos detalhes do último *job* executado, encontra-se a opção "*Revert*". Ao acionar essa funcionalidade, o sistema cria automaticamente uma nova *branch* contendo alterações inversas àquelas introduzidas pelo *merge* que originou o incidente. Esse procedimento gera um novo MR direcionado à *branch* `main`, o que desencadeia a reexecução integral do *pipeline* de implantação. O fluxo processual percorre a sequência completa

de estágios, culminando na geração de uma nova versão e na publicação de uma imagem atualizada no registro privado. No que tange à governança, a execução de *rollbacks*, bem como a aprovação de MRs, é prerrogativa exclusiva dos *Tech Leads* de cada equipe, sob atenta e próxima interação com o *Product Owner*.

4.1.5 Detalhamento dos *jobs* responsáveis pelos testes automatizados

Esta subseção apresenta uma análise mais aprofundada da implementação e da finalidade dos principais *jobs* de teste no *pipeline* de implantação vigente. Inicialmente, descrevem-se os aspectos técnicos e o escopo do *job* `test_integration`, incluindo a abrangência da suíte e os indicadores de cobertura de código. Em seguida, examinam-se as particularidades e o estado atual do *job* `test-e2e`, responsável pela validação sistêmica de ponta a ponta.

4.1.5.1 *job* `test_integration`

Este *job* executa validações em ambiente isolado por meio de um contêiner Docker. Durante sua execução, a aplicação é submetida a uma suíte de testes unitários, funcionais e de componentes, cujo objetivo é detectar regressões e assegurar a integridade do código antes da integração ao repositório principal. Os testes, elaborados pela equipe de desenvolvimento, localizam-se no diretório `/tests` da aplicação e utilizam a mesma linguagem do projeto. Ressalta-se que, no escopo deste *job*, as interações com dependências externas são substituídas por *mocks*.

No que se refere à abrangência da suíte, registrou-se uma captura da cobertura de código por meio da execução manual do comando `jest -coverage` na `branch staging`. Os indicadores resultantes são detalhados na Tabela 4.1.

Tabela 4.1 – Métricas de cobertura de código da suíte de testes automatizados

Atributo	Descrição	Valor
Statements	Proporção de instruções individuais no código que foram executadas durante os testes.	19,04%
Branches	Percentual de caminhos possíveis em estruturas de decisão que foram executados durante os testes.	14,98%
Functions	Proporção de funções ou métodos que foram chamados durante os testes.	15,26%
Lines	Percentual de linhas de código-fonte efetivamente executadas durante os testes.	19,42%

Fonte: Elaborado pelo autor (2026).

4.1.5.2 *Job* `test-e2e`

O *job* `test-e2e` tem como objetivo validar o comportamento do sistema de ponta a ponta, com foco nos fluxos críticos da aplicação. Para isso, atua como gatilho de um repositório externo

dedicado à garantia de qualidade da interface *web*, por meio de uma orquestração distribuída. A integração entre os repositórios é realizada via *Multi-Project Pipeline*, recurso que permite que o *pipeline* de um projeto acione e acompanhe *pipelines* de outros projetos, formando uma cadeia de execução. Nesse repositório externo, os testes seguem o padrão *Behavior Driven Development* (Desenvolvimento Guiado por Comportamento, BDD), com cenários descritos em linguagem natural para facilitar o alinhamento entre desenvolvedores e *stakeholders*.

4.2 Discussão acerca do Pipeline de Implantação Vigente

A partir do mapeamento sistemático apresentado nas seções anteriores, procede-se à análise crítica do *pipeline* de implantação vigente, identificando gargalos operacionais, débitos técnicos e fontes de risco que comprometem a velocidade e a estabilidade da entrega. Essa análise fundamenta as mudanças necessárias para que a solução proposta seja efetiva e estabelece a base para a comparação quantitativa entre os cenários. A discussão estrutura-se em três dimensões analíticas, trianguladas com os fundamentos teóricos do Capítulo 2, evidenciando o impacto potencial sobre as métricas DORA.

4.2.1 Estratégia de implantação do tipo *Big Bang* e concentração de risco

O *pipeline* vigente adota *Big Bang Deployment*, concentrando o risco de regressões em um único momento de liberação e expondo imediatamente todos os usuários a potenciais defeitos (HUMBLE; FARLEY, 2010). A ausência de mecanismos de exposição gradual tende a amplificar a CFR, pois falhas não detectadas em homologação manifestam-se sistemicamente em produção (FORSGREN; HUMBLE; KIM, 2018). Adicionalmente, a necessidade de reversão completa pressiona o MTTR, prolongando a indisponibilidade do serviço (KIM et al., 2016).

4.2.2 Fragilidade dos *quality gates* automatizados

Conforme evidenciado pela Tabela 4.1, os índices de cobertura comprometem a detecção precoce de regressões, contrariando as recomendações de Cohn (2009) sobre a base da pirâmide de testes. Defeitos escapam das etapas iniciais, sendo detectados apenas em estágios posteriores ou em produção, caracterizando uma falha no princípio fundamental da CI (FOWLER; FOEMMEL, 2006). A ausência de análise estática (isto é, a analise feita por ferramentas como SonarQube, Checkstyle) constitui uma lacuna adicional, impedindo a identificação de *code smells* e vulnerabilidades antes da fusão (UGWUEZE; CHUKWUNWEIKE, 2025). Ainda nessa problemática, informa-se na subseção 4.1.5.2 que o *job* `test-e2e` encontra-se suspenso, enfraquecendo criticamente o *quality gate* no topo da pirâmide. Embora os testes E2E devam ser mantidos em quantidade reduzida devido ao custo, sua ausência elimina a última barreira automatizada antes da implantação em ambiente de produção (COHN, 2009; VOCKE, 2018), elevando a CFR e

dificultando a identificação das causas raiz, o que prolonga o MTTR (FORSGREN; HUMBLE; KIM, 2018).

4.2.3 Dependência da intervenção humana na resposta a incidentes

A governança vigente introduz uma dependência crítica da detecção humana para restaurar a estabilidade. A Terceira Via do DevOps, introduzida na subseção 2.1.2, preconiza a automação de respostas a incidentes, reduzindo a variabilidade e viabilizando a recuperação previsível (KIM et al., 2016). A dependência manual constitui um gargalo em incidentes fora do horário comercial ou ocorrências concorrentes, prolongando o MTTR (FORSGREN; HUMBLE; KIM, 2018). A ausência de critérios automatizados baseados em telemetria contrasta com as recomendações de Malhotra et al. (2024) e Ramaswamy (2024), segundo os quais *Canary Deployment* deve ser sustentado por evidências observáveis e limiares calibrados. No fluxo vigente, a reversão depende de uma percepção subjetiva, introduzindo variabilidade e retardando a resposta.

4.3 Linha de Base das Métricas DORA

Esta seção apresenta a caracterização quantitativa do desempenho do *pipeline* de implantação vigente por meio das métricas DORA, estabelecendo a linha de base necessária para a comparação com o cenário proposto. Conforme descrito na subseção 3.3.2, os dados foram extraídos da API do GitLab, abrangendo o período de janeiro a dezembro de 2024, anterior ao *freezing* das operações. A subseção 4.3.1 apresenta a frequência de implantação, a subseção 4.3.2 analisa a taxa de falha de mudanças, a subseção 4.3.3 caracteriza o tempo de entrega de mudanças e a subseção 4.3.4 quantifica o tempo médio de recuperação.

4.3.1 Deployment Frequency (Frequência de Implantação, DF)

No contexto deste trabalho, uma implantação foi considerada válida somente quando houve evidência de execução bem-sucedida de um *job* responsável por promover uma versão para o ambiente produtivo. Dessa forma, o cálculo da métrica foi baseado exclusivamente em implantações concluídas com sucesso, descartando execuções interrompidas, canceladas ou finalizadas com erro, uma vez que tais ocorrências não materializam, de fato, uma nova versão em produção.

Para a coleta dos dados, foram consideradas as implantações realizadas a partir das *branches* `main` e `gcloud-cicd`. A inclusão da *branch* `main` decorre de seu papel como principal linha de desenvolvimento associada ao ambiente produtivo. Já a inclusão da *branch* `gcloud-cicd` foi definida após análise do *pipeline* de implantação vigente e da inspeção manual das execuções disponíveis na interface do GitLab. Essa análise evidenciou que a referida *branch* foi criada e utilizada durante um período específico para promover versões diretamente para produção com a *tag* `stable`, sem passar pelos estágios usuais de promoção para a *branch* `staging` ou pelos

mecanismos bloqueantes do *pipeline* de implantação. Na prática, essa configuração caracterizava um mecanismo de contorno do fluxo regular de entrega, isto é, um *bypass* operacional, entendido como uma alternativa temporária que permite evitar etapas intermediárias ou validações normalmente exigidas pelo processo padrão.

A extração dos dados foi realizada por meio da biblioteca `python-gitlab`, conforme descrito na subseção 3.3.2, a partir da consulta aos *pipelines* e *jobs* associados ao projeto. Foram aplicados os seguintes filtros: (i) *branches* `main` e `gcloud-cicd`; (ii) *jobs* com os nomes `deploy-production` e `deployProduction`; (iii) *status* final `success`; e (iv) intervalo temporal de 1º de janeiro a 31 de dezembro de 2024, conforme estabelecido na subseção 3.3.2. A escolha dos nomes dos *jobs* decorreu da análise histórica dos *pipelines* de implantação executados. Embora o arquivo `.gitlab-ci.yml` vigente utilize a nomenclatura `deploy-production`, também foram identificadas execuções anteriores nomeadas como `deployProduction`. Essa diferença indica, muito provavelmente, uma alteração na definição do *pipeline* de implantação ao longo do tempo, na qual a nomenclatura anterior em *camelCase*[3] foi substituída pela forma atual em *kebab-case*[4]. Dessa forma, ambos os nomes foram incluídos no *script* de coleta para preservar a rastreabilidade histórica e evitar a exclusão de implantações válidas realizadas sob uma configuração anterior do *pipeline*. O filtro por *status* `success` foi adotado para assegurar que apenas implantações efetivamente concluídas fossem contabilizadas, em conformidade com a definição operacional estabelecida na subseção 3.5.1.

Ao final da coleta, foram identificadas 11 implantações bem-sucedidas ao longo do ano de 2024, conforme ilustrado na Figura 4.4. Aplicando a definição operacional estabelecida na Equação 3.1, e substituindo os valores observados ($D = 11$ implantações e $t = 12$ meses), obtém-se $\text{DF} = 11/12 \approx 0{,}92$ implantações por mês. De acordo com a classificação DORA apresentada por DeBellis et al. (2024), esse valor enquadra a organização no nível *Baixo*, caracterizado por uma frequência de implantação entre uma vez por mês e uma vez por semestre. Além do valor agregado anual, a distribuição mensal das implantações evidencia concentração no primeiro semestre, especialmente no mês de junho, seguido de uma redução acentuada no segundo semestre. Esse comportamento sugere que a cadência de entrega não foi homogênea ao longo do ano, podendo estar associada a fatores organizacionais e contextuais, como ciclos de planejamento, priorização de demandas, disponibilidade da equipe e períodos de férias acadêmicas.

[3] *CamelCase* é uma convenção de nomenclatura em que identificadores compostos são escritos sem espaços ou separadores, com inicial maiúscula nas palavras subsequentes, como em `deployProduction`.

[4] *Kebab-case*, também denominado *dash-case*, é uma convenção de nomenclatura em que as palavras de um identificador são escritas em letras minúsculas e separadas por hífen, como em `deploy-production`.

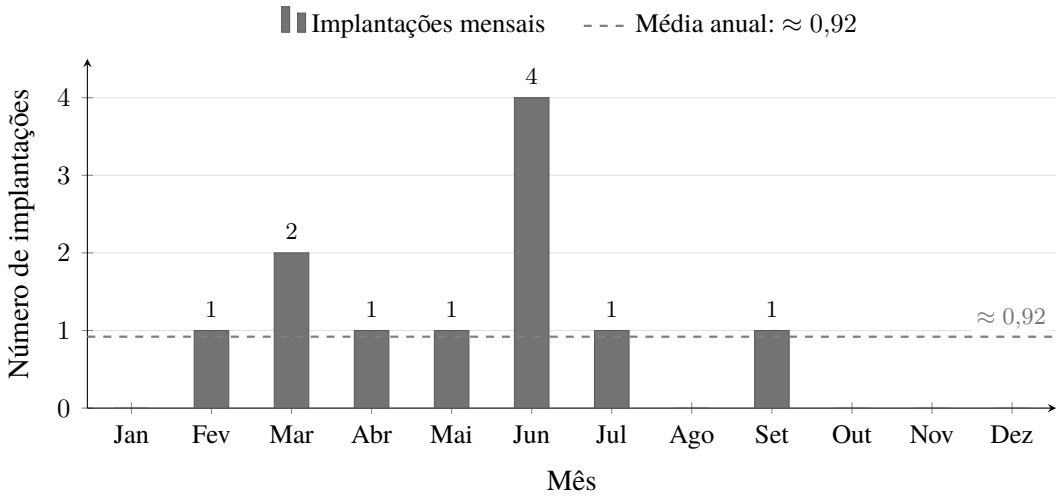

Figura 4.4 – Evolução mensal da frequência de implantações no TerraPlanner em 2024

Fonte: Elaborada pelo autor (2026).

4.3.2 Taxa de Falha nas Mudanças (*Change Failure Rate*, CFR)

Como a organização não adota um processo formal de registro de incidentes, a identificação das falhas exigiu uma abordagem indireta. Para isso, foi desenvolvido um *script* em Python que triangula evidências provenientes do repositório Git e do sistema de *issues* do GitLab, com o objetivo de reduzir falsos negativos. Foram considerados cinco critérios de detecção: (i) implantações consecutivas com intervalo reduzido, nas quais entregas em produção realizadas em menos de 8 horas foram interpretadas como indício de correção emergencial; (ii) mensagens de *commit* indicativas de correção, considerando *commits* implantados em até 48 horas após a implantação anterior e contendo termos como `fix`, `hotfix` ou `revert`; (iii) *issues* de produção temporalmente correlacionadas, criadas em até 72 horas após a implantação e identificadas por *labels* como `bug`, `hotfix`, `production-issue` ou `critical`, ou por palavras-chave no título; (iv) reversões por ancestralidade Git, nas quais *rollbacks* foram inferidos pela verificação de que o *Secure Hash Algorithm* (SHA) de uma implantação subsequente é ancestral do SHA anterior no histórico Git, ou pela constatação de que o *timestamp* de criação do *commit* implantado é anterior ao da implantação precedente, caracterizando retrocesso temporal; e (v) MRs de correção emergencial, fundidos em até 48 horas após uma implantação e identificados por *labels* (`hotfix`, `urgent`, `emergency`, `critical`) ou palavras-chave no título, associando a implantação anterior a uma falha que demandou correção emergencial.

Essa estratégia reconhece que falhas em produção podem se manifestar de diferentes formas no ciclo de desenvolvimento. Embora a ausência de um registro formal de incidentes limite a precisão absoluta da métrica, a triangulação de evidências representa a aproximação mais adequada ao contexto organizacional analisado. A extração dos dados foi realizada com a biblioteca `python-gitlab`, aplicando os critérios descritos às 11 implantações identificadas na subseção 4.3.1. Cada implantação foi classificada como falha quando ao menos um critério

foi satisfeito, privilegiando a sensibilidade da detecção, ainda que com risco de falsos positivos.

A Figura 4.5 apresenta a distribuição temporal das falhas identificadas em 2024. Das 11 implantações analisadas, 5 foram classificadas como falhas. Aplicando a definição operacional estabelecida na Equação 3.4, e substituindo os valores observados ($N_{failed} = 5$ e $N_{total} = 11$), obtém-se CFR $= \left(\frac{5}{11}\right) \times 100\% = 45{,}45\%$. Segundo a classificação DORA apresentada por DeBellis et al. (2024), esse valor enquadra a organização no nível *Baixo*, caracterizado por taxa de falha superior a 30%.

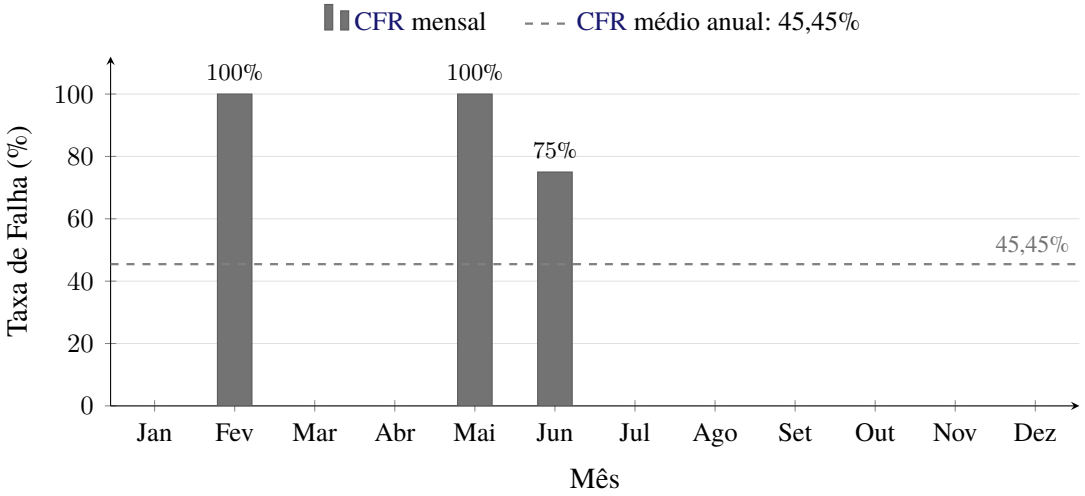

Figura 4.5 – Evolução mensal da CFR no TerraPlanner em 2024

Fonte: Elaborada pelo autor (2026).

4.3.3 Lead Time for Changes (Tempo de Entrega para Mudanças, LTFC)

A extração dos dados foi realizada por meio da biblioteca `python-gitlab`, aplicando uma estratégia de rastreamento retrospectivo a partir de cada implantação bem-sucedida. Para cada *deploy*, o *script* identifica o SHA do *commit* implantado e percorre o histórico Git para localizar o *commit* mais antigo incluído naquela entrega, isto é, aquele que marca o início efetivo do desenvolvimento da mudança. O *lead time* é então calculado como a diferença temporal entre a data de autoria desse *commit* inicial e o *timestamp* de conclusão do *job* de implantação. Quando há um MR associado à implantação, o *script* compara a data de autoria do *commit* mais antigo com a data de criação do MR, utilizando a mais antiga como ponto de partida, a fim de capturar o tempo total de desenvolvimento, incluindo eventuais períodos de trabalho em *feature branch* antes da abertura formal do MR. Para assegurar a validade dos dados e evitar distorções causadas por artefatos do processo de versionamento, foram aplicados filtros de saneamento. Foram excluídos *commits* automáticos, identificados por mensagens contendo termos como `merge`, `bump`, `[skip ci]` ou referências a documentação, uma vez que tais *commits* não representam mudanças funcionais e podem introduzir ruído na medição.

No período analisado, foram registradas 11 implantações, com um *lead time* médio de

12,58 dias após a exclusão do *outlier* de março, discutida a seguir. Esse valor foi calculado conforme a definição operacional estabelecida na Equação 3.2 para cada implantação individual e, posteriormente, agregado segundo a Equação 3.3 sobre as 10 implantações remanescentes. De acordo com a classificação DORA apresentada por DeBellis et al. (2024), esse valor enquadra a organização no nível *Médio*, caracterizado por um tempo de entrega entre 7 e 30 dias. A Figura 4.6 ilustra a distribuição temporal dos *lead times* ao longo de 2024.

O valor extremo de 109,28 dias registrado em março constitui um *outlier* causado por uma *branch* antiga retomada após um período de inatividade, conforme evidenciado pela análise do histórico Git; esse padrão não reflete o processo normal de desenvolvimento, mas sim uma exceção operacional. Esse registro foi removido apenas do cálculo da média, permanecendo, contudo, visíveis na Figura 4.6 para preservar a transparência sobre a distribuição bruta dos dados. Das 11 implantações do período, apenas a de março ultrapassa esse limiar.

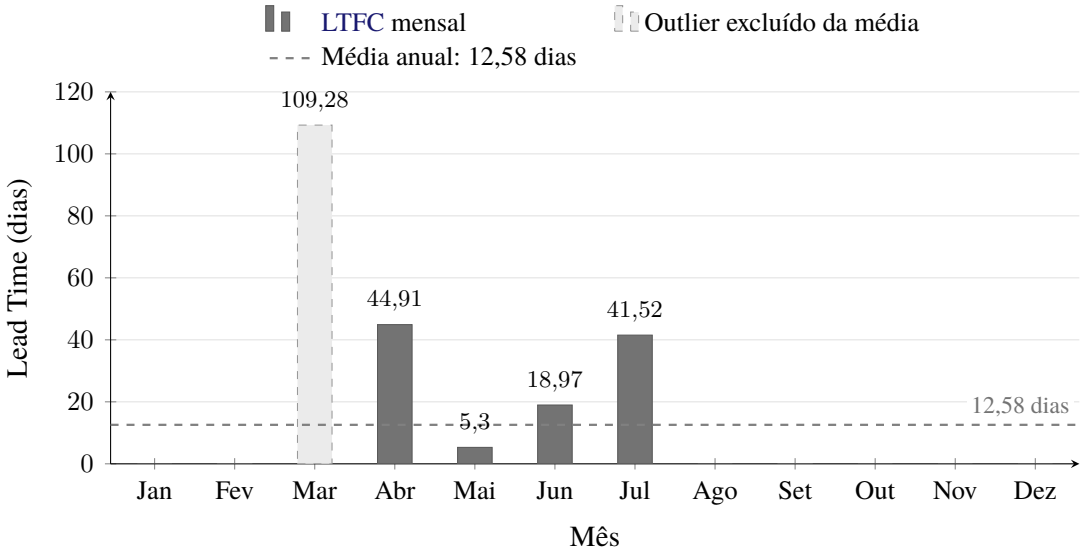

Figura 4.6 – Evolução mensal do *Lead Time for Changes* no TerraPlanner em 2024

Fonte: Elaborada pelo autor (2026).

4.3.4 Tempo para Restaurar o Serviço (*Mean Time To Restore*, MTTR)

Neste trabalho, o MTTR foi calculado para cada incidente identificado na subseção 4.3.2, considerando o tempo decorrido desde a conclusão da implantação que introduziu a falha até a conclusão da implantação que restaurou a estabilidade do sistema. A extração dos dados foi realizada por meio da biblioteca `python-gitlab`, aplicando uma estratégia de rastreamento sequencial a partir de cada implantação classificada como falha. Para cada incidente, o *script* identifica a próxima implantação bem-sucedida que não apresenta evidências de falha segundo os critérios estabelecidos na subseção 4.3.2, assumindo-a como o momento de recuperação. O MTTR é então calculado como a diferença temporal entre o *timestamp* de conclusão da implantação com falha e o *timestamp* de conclusão da implantação de recuperação.

No período analisado, foram identificados 5 incidentes associados às implantações com falha detectadas na subseção 4.3.2. Aplicando a definição operacional estabelecida na Equação 3.5, onde $I = 5$ incidentes e a soma das durações individuais $(T_i^{recovery} - T_i^{detect})$ totaliza 1.306,45 horas, obtém-se um MTTR médio de 261,29 horas, equivalente a aproximadamente 10,89 dias. De acordo com a classificação DORA apresentada por DeBellis et al. (2024), esse valor enquadra a organização no nível *Baixo*, caracterizado por um tempo de recuperação superior a uma semana. A Figura 4.7 ilustra a distribuição temporal dos tempos de recuperação ao longo de 2024. Observa-se uma variabilidade significativa entre os incidentes, com tempos de recuperação variando de 7,72 horas no incidente de junho a 647,71 horas (isto é, aproximadamente 27 dias) no incidente de maio. Além disso, a concentração de incidentes no período de fevereiro a junho, coincidindo com o pico de atividade de implantações observado na subseção 4.3.1, reforça a correlação entre a intensificação da cadência de entregas e a incidência de falhas em produção.

Essa amplitude de variação sugere que o tempo de recuperação é fortemente influenciado por fatores contextuais, tais como a complexidade da falha, a disponibilidade da equipe para diagnóstico e correção, e a necessidade de coordenação entre múltiplos *stakeholders* para aprovação da reversão.

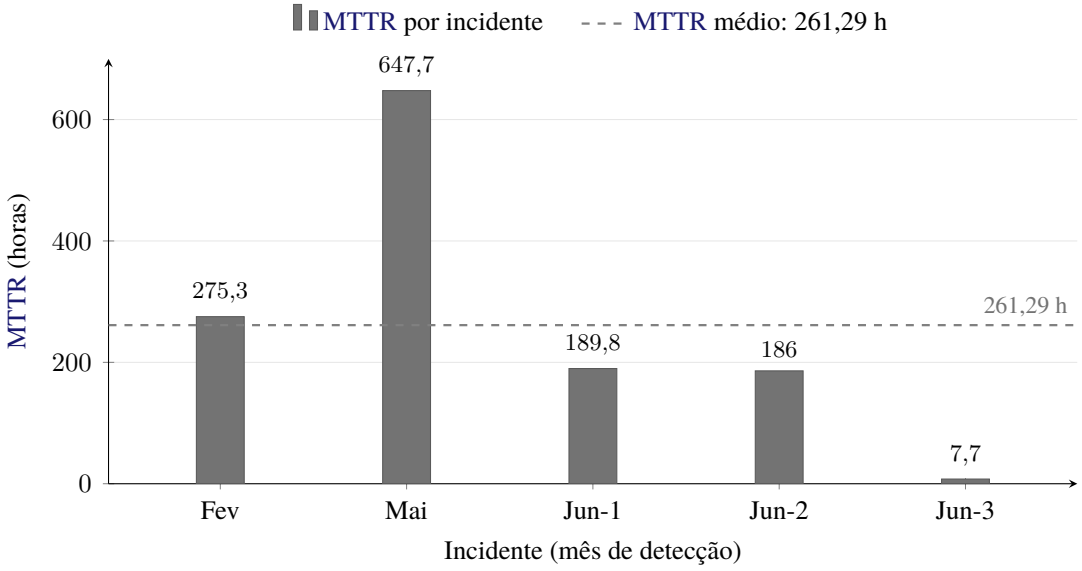

Figura 4.7 – MTTR por incidente no TerraPlanner em 2024

Fonte: Elaborada pelo autor (2026).

4.4 Proposta de Evolução do Pipeline de Implantação

Com base nas limitações identificadas no *pipeline* de implantação vigente, esta seção apresenta a arquitetura conceitual da solução proposta, formalizando os mecanismos de *Canary Deployment* com *rollback* automatizado que visam mitigar os gargalos operacionais e aprimorar as métricas de velocidade e estabilidade. A subseção 4.4.1 detalha a arquitetura lógica do *pipeline* de implantação proposto, descrevendo os fluxos operacionais de promoção entre ambientes, os

artefatos gerados em cada etapa, as estratégias de roteamento progressivo e os procedimentos de reversão automatizada em cenários de falha. Em seguida, a subseção 4.4.2 especifica o método de coleta e cálculo das métricas DORA.

4.4.1 Arquitetura lógica do *pipeline* de implantação proposto

Nesta subseção, apresenta-se a arquitetura lógica do *pipeline* de implantação proposto, delineando os fluxos operacionais, os artefatos gerados e consumidos em cada etapa e os pontos de decisão que governam a promoção de mudanças. Assim como no *pipeline* de implantação vigente, o modelo proposto organiza-se em fluxos de promoção entre *branches*. A seguir, descrevem-se os três fluxos principais que o estruturam, o fluxo de *rollback* utilizado em cenários de falha e o *job* agendado de observabilidade que coleta eventos para as métricas DORA.

4.4.1.1 Fluxo I — Feature Branch → development

***Gatilho de Pré-integração (MR)*:** Quando um MR é aberto com *branch* alvo `development`, o *pipeline* de integração contínua é executado em modo de verificação, priorizando validações automatizadas de caráter bloqueante. Nessa etapa, são executados os seguintes *jobs*:

- `lint`: validação de qualidade de código e formatação via ESLint, com publicação de relatório estruturado. Falhas são bloqueantes quando a respectiva *feature flag* de bloqueio está ativa.
- `test_unit`: execução de testes unitários, validando unidades de código isoladamente. O *job* gera relatório JUnit para integração com a aba de testes da UI do GitLab.
- `test_integration`: execução de testes de integração, validando interações entre componentes e também gerando relatório JUnit.
- `audit`: auditoria de dependências para identificar vulnerabilidades conhecidas. Vulnerabilidades de severidade `critical` podem bloquear a integração, enquanto vulnerabilidades de menor severidade geram avisos e relatórios de diagnóstico.
- `coverage`: medição de cobertura de testes automatizados, com geração de relatório compatível com a interface do GitLab. Por padrão, esse controle atua como aviso não bloqueante, salvo configuração explícita em sentido contrário.

***Gatilho de Pós-integração (push/merge)*:** Após a aprovação manual e a fusão do MR em `development`, um novo gatilho é disparado na *branch* de destino. Nessa fase, os *jobs* `test_unit`, `test_integration` e `audit` são reexecutados para verificar a consistência pós-*merge* e detectar efeitos colaterais da resolução de conflitos. Esse fluxo é sintetizado na Figura 4.8.

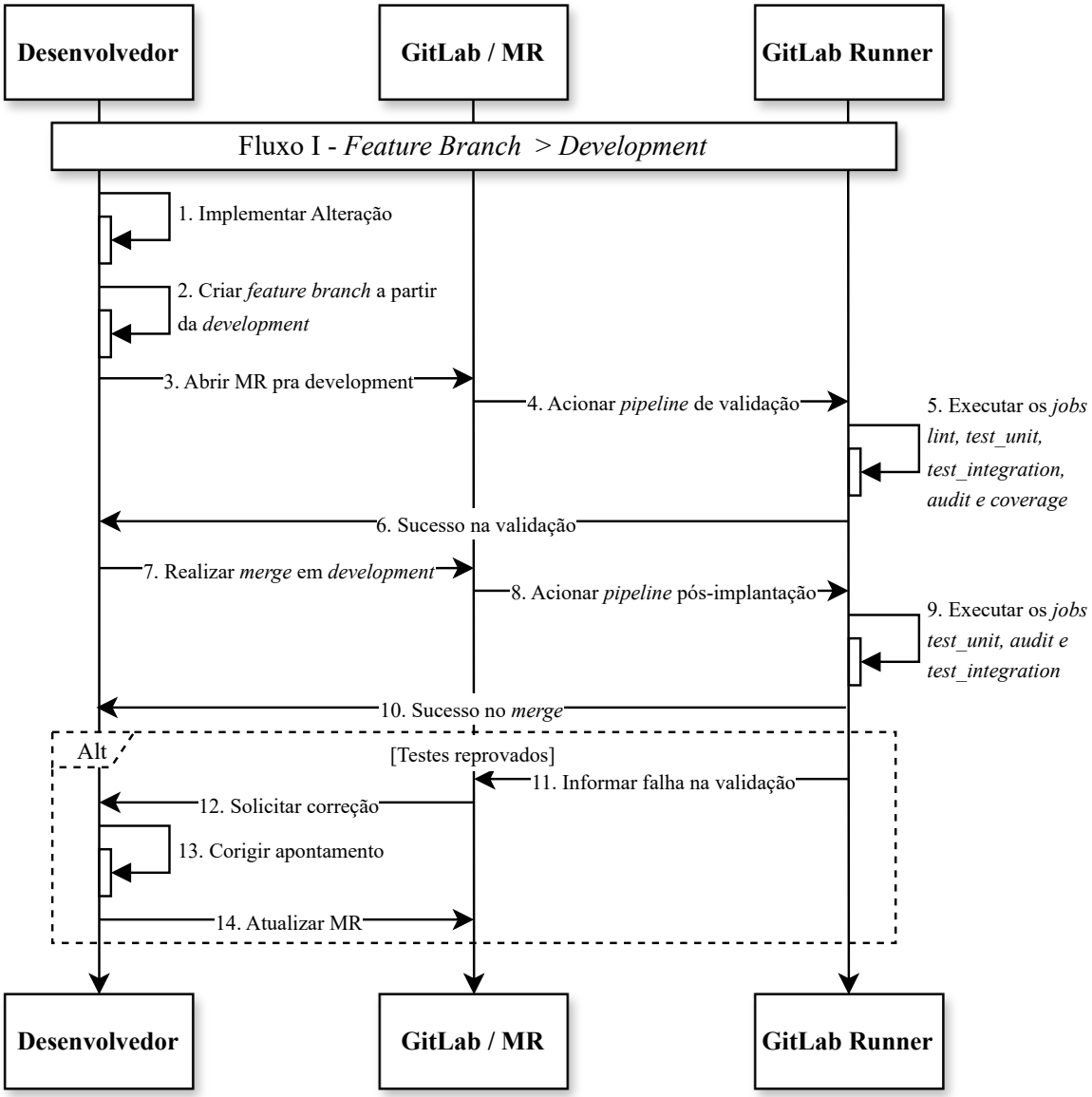

Figura 4.8 – Diagrama de sequência do Fluxo I do pipeline de implantação proposto

Fonte: Elaborado pelo autor (2026).

4.4.1.2 Fluxo II — development → staging

***Gatilho de Pré-integração (MR)*:** Quando um MR é aberto de `development` para `staging`, o *pipeline* de implantação é executado em modo de verificação. As regras atuais incluem os *jobs* `lint`, `test_unit`, `test_integration`, `audit` e `coverage`, além do `validate_config`. Este último valida a presença e o formato das variáveis obrigatórias para o ambiente de *staging*, registrando evidências de diagnóstico quando há falha.

***Gatilho de Pós-integração (push/merge)*:** Após a aprovação do MR, realiza-se um *push* para a *branch* `staging`, o que aciona o *pipeline* de implantação no ambiente-alvo. Antes da implantação propriamente dita, os *jobs* `test_unit`, `test_integration`, `audit` e `validate_config`

são reexecutados. A Figura 4.9 sintetiza o fluxo completo. Posteriormente, apresenta-se uma breve explicação conceitual de cada *job*.

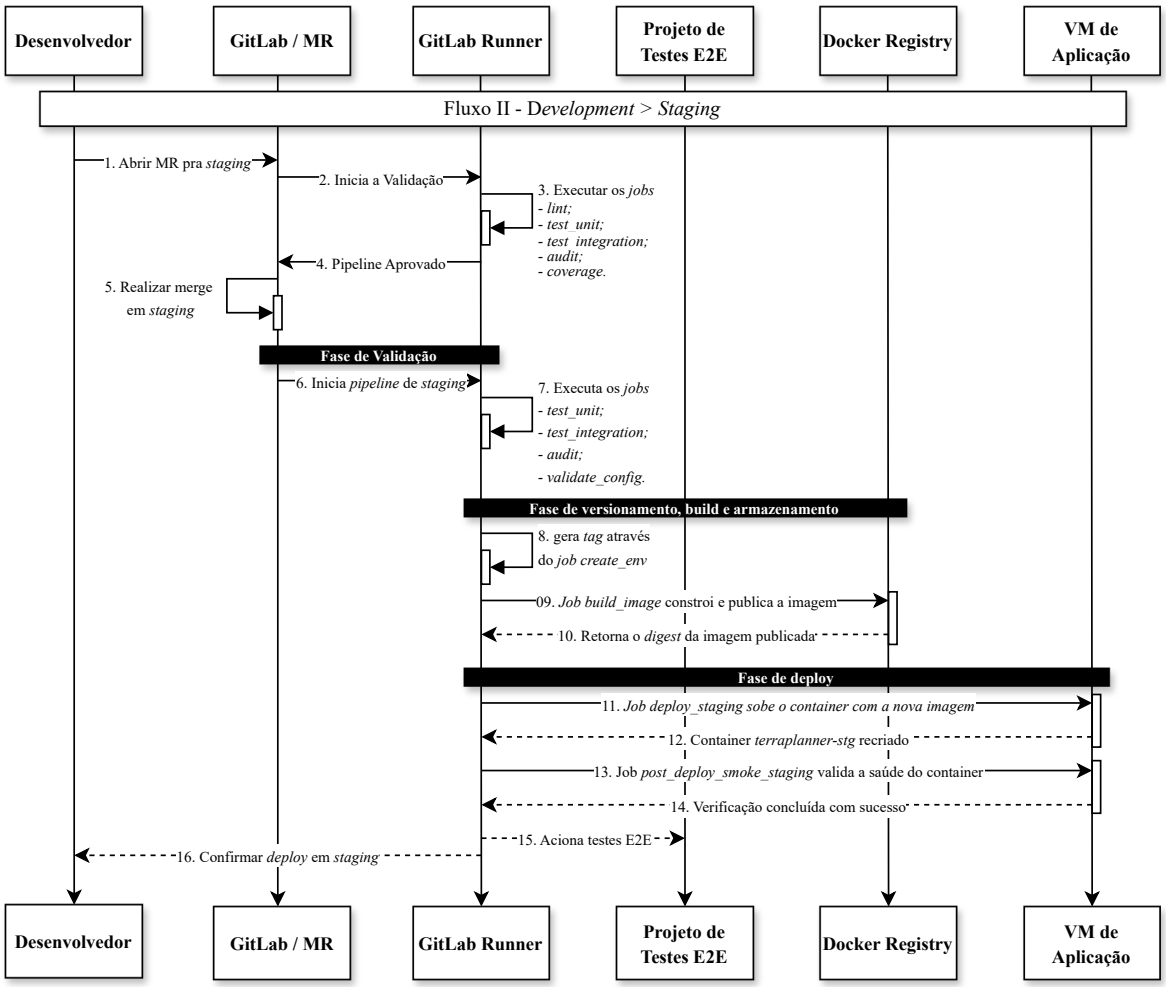

Figura 4.9 – Diagrama de sequência do Fluxo II do pipeline de implantação proposto

Fonte: Elaborado pelo autor (2026).

- `create_env`: construção de variáveis e metadados da *release candidate*. O *job* gera uma etiqueta de *release candidate* com sufixo incremental e exporta variáveis via artefato `dotenv`.

- `build_image`: constrói a imagem Docker, publica-a no registro privado e captura seu *digest*, correspondente a um valor de *hash* SHA-256 calculado com base no conteúdo do manifesto da imagem. Esse valor identifica o manifesto de maneira única e imutável, diferentemente de uma *tag*, que pode ser reatribuída a versões distintas ao longo do tempo[5]. Em seguida, exporta esse identificador para uso por *jobs* subsequentes.

[5] Documentação disponível em: <https://docs.docker.com/reference/cli/docker/image/pull/>. Acesso em: 15 jul. 2026.

- `deploy_staging`: implanta a imagem em *staging* via *Secure Shell* (SSH) na VM alvo, utilizando *docker compose* e referenciando a imagem por *digest*. Também gera evento de implantação para rastreabilidade.

- `post_deploy_smoke_staging`: executa validação pós-implantação com testes de saúde e checagens de SLIs, registrando evidências em relatório estruturado ao final da execução.

- `test_e2e_staging` e `test_e2e_staging_async`: acionam a suíte E2E em projeto de QA externo contra o *staging* atualizado. A escolha entre execução síncrona e assíncrona é controlada por *feature flags*, permitindo aguardar ou apenas disparar o *pipeline* de QA.

4.4.1.3 Fluxo III — staging → main (produção)

***Gatilho de Pré-integração (MR)*:** Ao abrir um MR de `staging` para `main`, o *pipeline* de integração contínua é executado em modo de verificação. As regras atuais reexecutam os *jobs* de qualidade `lint`, `test_unit`, `test_integration`, `audit` e `coverage`. Adicionalmente, o `assert_image_exists_in_registry` valida a existência da imagem no registro privado, e o `assert_staging_e2e_passed` verifica as evidências do *pipeline* E2E de *staging*, operando como bloqueante apenas quando a *feature flag* correspondente está habilitada.

***Gatilho de Pós-integração (*push/merge*):** Uma vez aprovado o MR, ocorre um *push* na *branch* `main`, disparando o *rollout canary* em produção. Antes de recuperar o artefato a ser promovido, o `validate_config` revalida variáveis obrigatórias do ambiente produtivo. Em seguida, o *rollout* é estruturado em quatro fases sequenciais:

- ***Fase 0 — Preparação do Artefato*:** Nesta fase, executa-se o *job* `resolve_digest`. Ele resolve o *digest* criptográfico da imagem previamente validada em *staging*, verifica sua existência no registro privado e exporta o *digest* via artefato `dotenv`. Não há reconstrução da imagem; a promoção ocorre por referência ao *digest*.

- ***Fase 1 — Deploy Canary (0% de Tráfego Público)*:** Nesta fase, executam-se os *jobs* `deploy_canary` e `smoke_canary`. O primeiro inicia um contêiner *canary* em paralelo ao `stable`, ainda sem tráfego público; o segundo valida a saúde mínima da aplicação antes do roteamento progressivo. Falhas nessa etapa acionam o *job* `rollback_canary_auto`.

- ***Fase 2 — Roteamento Progressivo e Análise por SLI*:** Após a validação inicial, o tráfego é direcionado gradualmente ao contêiner *canary* pelos *jobs* `verify_canary_10pct`, `verify_canary_25pct` e `verify_canary_50pct`. Cada *job* atualiza o percentual de exposição, observa o comportamento do *canary* em uma janela configurada e valida os SLIs antes de liberar o próximo incremento.

  - ***Estratégia 1 — Roteamento por Allowlist:*** Adequada ao cenário atual, no qual o produto ainda não possui usuários ativos. O NGINX realiza o roteamento com base

na presença de um *cookie* HTTP específico nas requisições à rota de produção: requisições que incluem o *cookie* são direcionadas ao *canary*, e as demais ao `stable`. Esse mecanismo viabiliza validação controlada por operadores ou pela equipe de QA.

  - ***Estratégia 2 — Roteamento Progressivo por Percentuais:*** Indicada no cenário pós-*go-live*. Aqui o NGINX distribui o tráfego em percentuais crescentes via diretiva `split_clients`, com distribuição determinística por *hash*. A cada incremento, o *pipeline* de implantação observa o comportamento do serviço, valida SLIs contra limites definidos e registra evidências em relatório estruturado. Se aprovado, avança para o próximo percentual; se reprovado, aciona `rollback_canary_auto`.

- ***Fase 3 — Promoção e Validações Pós-promoção*:** Após a aprovação do incremento de 50%, o *job* `promote_canary_to_stable` promove a versão *canary* para `stable`, por acionamento manual ou automático conforme a variável de controle. Em seguida, o `smoke_stable_post_promote` valida a saúde do ambiente estável recém-promovido. Também existem os *jobs* `test_e2e_stable` e `test_e2e_stable_async`, responsáveis por acionar a suíte E2E contra produção em modo síncrono ou assíncrono, de acordo com as *feature flags* do fluxo. A Figura 4.9 sintetiza o fluxo descrito por completo.

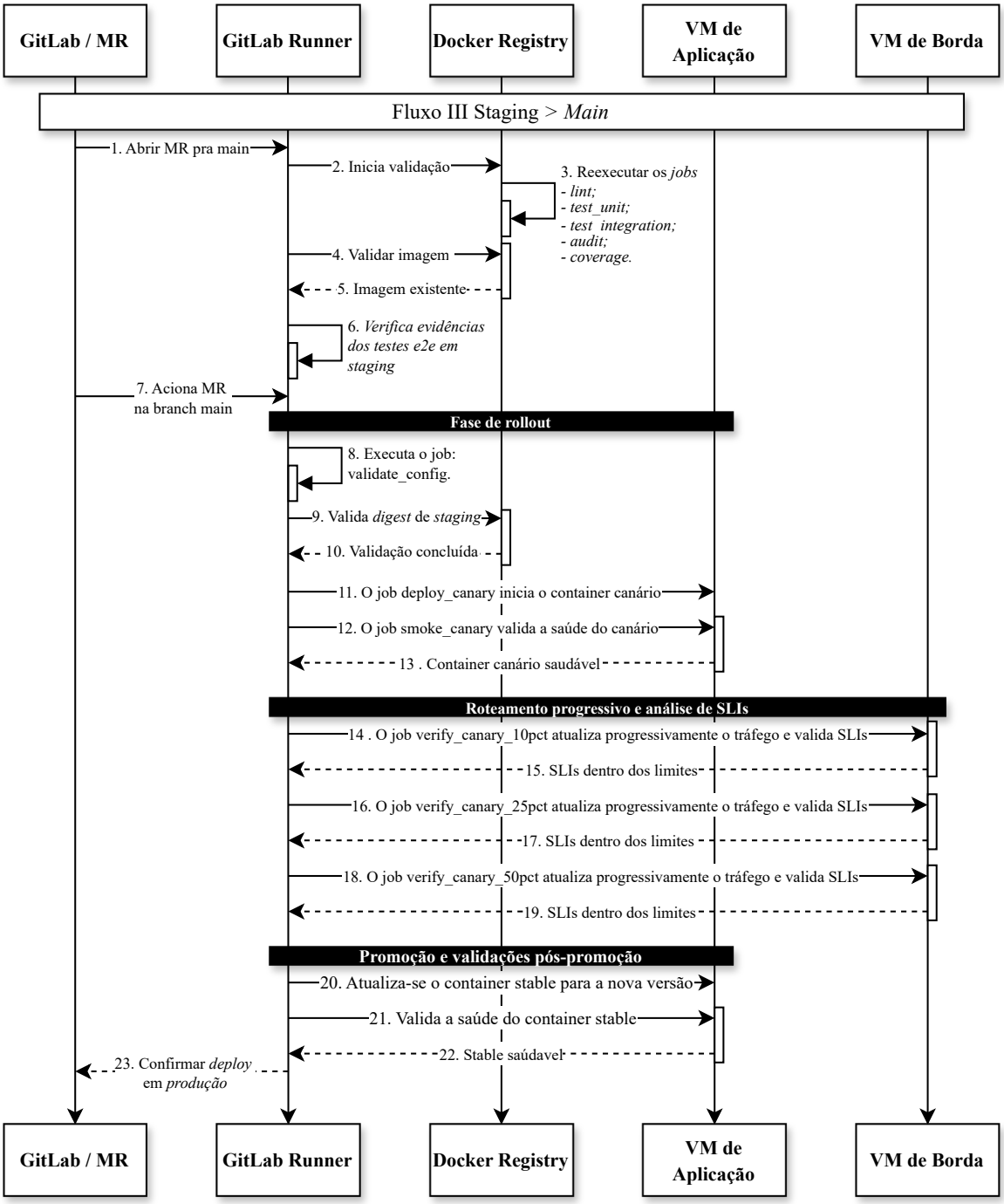

Figura 4.10 – Diagrama de sequência do Fluxo III do pipeline de implantação proposto

Fonte: Elaborado pelo autor (2026).

4.4.1.4 Fluxo IV — *rollback* automatizado

Cenário A — rollback durante o canary: é acionado quando uma falha é detectada durante o *rollout* progressivo ou na fase de análise, antes da promoção para `stable`. O procedimento de reversão é executado pelo *job* `rollback_canary_auto`, acionado por falha no fluxo de produção ou manualmente quando necessário. Sua função é redirecionar o tráfego para o `stable`, remover

o contêiner *canary*, verificar a saúde do ambiente estável e registrar eventos de implantação e incidente para rastreabilidade.

Cenário B — rollback em produção após promoção: é acionado quando uma falha é detectada após a promoção do *canary* para `stable`, com 100% do tráfego direcionado à nova versão. Como o `stable` já foi atualizado para o novo *digest*, a reversão exige identificar e recuperar o *digest* anterior, ou seja, a última versão saudável conhecida. Essa identificação é realizada a partir do arquivo `deployment_history.json`, mantido na VM de produção e atualizado após cada promoção bem-sucedida. O procedimento pode ser executado pelo *job* manual `rollback_stable` ou pelo `rollback_stable_auto`, criado para reagir automaticamente a falhas no `smoke_stable_post_promote`. Ambos registram evidências do incidente em artefatos consumidos posteriormente pela observabilidade das métricas DORA.

4.4.2 Observabilidade e instrumentação para métricas DORA

A evolução do *pipeline* também incorpora uma camada de observabilidade voltada ao cálculo das métricas DORA. A instrumentação não depende apenas dos registros nativos do GitLab; ela combina artefatos gerados durante a própria execução do *pipeline* de implantação através de consultas à API do GitLab, permitindo relacionar eventos de implantação, mudanças de código e incidentes de recuperação.

4.4.2.1 Fontes de dados e modelo de eventos

O modelo de eventos adotado diferencia dois registros operacionais principais. Os eventos de implantação representam atualizações concluídas em *staging* ou produção, incluindo informações como ambiente, *commit*, *digest*, estado da execução e momento de conclusão. Os eventos de incidente representam remediações associadas ao fluxo de entrega, incluindo *rollback* durante o *canary*, *rollback* manual em *stable* e *rollback* automático após falha na validação pós-promoção. Esses eventos são materializados em artefatos em formato JSON, como `deployment_event.json` e `incident_event.json`, produzidos no momento em que a implantação ou a reversão ocorre.

A coleta das métricas é executada de forma assíncrona por um *pipeline* agendado. Em cada execução, o coletor consulta os *pipelines* de implantação recentes da *branch* `main`, recupera artefatos de implantação e de incidentes, enriquece os registros com dados de *commits* obtidos por meio da API do GitLab e gera instruções idempotentes de persistência. A idempotência consiste na propriedade segundo a qual a repetição de uma mesma operação produz o mesmo estado final, evitando a duplicação ou a alteração indevida dos registros. Essa separação impede que a coleta de métricas aumente a duração do caminho crítico da implantação, ao mesmo tempo que preserva a granularidade necessária para auditorias posteriores.

4.4.2.2 Publicação das métricas e visualização em *dashboards*

Os eventos consolidados são persistidos em uma base PostgreSQL dedicada às métricas DORA, organizada em tabelas para implantações e incidentes. A tabela de implantações registra o vínculo entre projeto, ambiente, *commit*, momento de implantação e origem do evento; a tabela de incidentes registra o tipo de remediação, o gatilho, os instantes de detecção e recuperação e a associação com a implantação relacionada. Esse modelo fornece a base analítica necessária para calcular frequência de implantação, tempo de entrega de mudanças, taxa de falha de mudanças e tempo médio de recuperação.

A visualização é realizada no Grafana, que consulta a base analítica e aplica agregações conforme o intervalo selecionado no painel. A frequência de implantação é derivada da contagem de implantações bem-sucedidas em produção; o tempo de entrega de mudanças é calculado pela diferença entre o momento de autoria ou início da mudança e sua efetiva disponibilização em produção; a taxa de falha considera a proporção entre incidentes e eventos de implantação no período analisado; e o MTTR é calculado a partir da diferença entre detecção e recuperação dos incidentes com restauração confirmada. Com isso, a observabilidade deixa de depender de inspeções manuais isoladas e passa a compor um fluxo reprodutível de evidências para a avaliação quantitativa do *pipeline*.

4.5 Implementação do Pipeline Proposto

Esta seção apresenta a implementação do artefato tecnológico proposto na seção 4.4, materializado como um *pipeline* de implantação voltado à validação automatizada, à publicação rastreável de artefatos, à exposição progressiva por *Canary Deployment*, ao *rollback* automatizado e ao registro de eventos para métricas DORA. Para evidenciar como a proposta foi operacionalizada, a discussão organiza-se em seis dimensões complementares. Inicialmente, a subseção 4.5.1 descreve a visão geral da arquitetura de implantação, incluindo a topologia de execução e a separação entre o repositório de *templates* e o projeto alvo. Em seguida, a subseção 4.5.2 detalha a integração contínua e os *quality gates* aplicados antes da construção da imagem, enquanto a subseção 4.5.3 descreve a construção e o versionamento dos artefatos, do versionamento semântico automático à implantação em *staging*. Na sequência, a subseção 4.5.4 apresenta a implementação da estratégia de *Canary Deployment*, detalhando as quatro fases de exposição progressiva de tráfego, ao passo que a subseção 4.5.5 descreve os mecanismos de recuperação acionados em caso de falha, tanto antes quanto depois da promoção. Por fim, a subseção 4.5.6 apresenta a implementação da coleta e da visualização das métricas DORA. Em conjunto, essas dimensões descrevem as decisões técnicas que estruturam o fluxo de entrega e estabelecem os mecanismos necessários para avaliar velocidade, estabilidade e rastreabilidade no contexto investigado.

4.5.1 Visão geral da arquitetura de implantação

O *pipeline* de implantação segue um modelo de repositório de templates centralizado distinto do repositório do projeto alvo, isto é, o repositório do *front-end* do TerraPlanner cujo código de aplicação é efetivamente implantado. O repositório de templates não contém código de aplicação; sua função exclusiva é definir os *jobs* e *stages* do CI/CD. O repositório do projeto alvo aponta para orquestrador raiz (isto é, o arquivo `.gitlab-ci.yml` do repositório de templates) por meio da opção *CI/CD configuration file* do próprio GitLab. Essa separação entre a lógica do *pipeline* de implantação e o código de aplicação facilita a manutenção independente e a reutilização dos templates em outros projetos do laboratório. Uma regra de *workflow* impede que o próprio repositório de templates acione um *pipeline* de implantação, evitando execuções acidentais no contexto errado; essa restrição também abre caminho para a criação futura de um *pipeline* de implantação dedicado ao próprio repositório de templates.

O orquestrador raiz declara treze estágios sequenciais: `quality`, `validate`, `package`, `deploy`, `smoke`, `e2e`, `canary-10`, `canary-25`, `canary-50`, `promote`, `e2e-stable`, `rollback` e `observability`, cada um implementado por um ou mais arquivos YAML. Essa sequência materializa o desenho lógico apresentado na subseção 4.4.1: mudanças consolidadas em `staging` percorrem qualidade, empacotamento, implantação em homologação e validações pós-*deploy*; mudanças promovidas para `main` reutilizam o artefato já publicado, executam o fluxo canário em produção e geram eventos para posterior cálculo das métricas DORA.

A topologia de execução, sintetizada na Figura 4.11, atribui ao GitLab Runner o papel de orquestrador das ações realizadas pelo *pipeline* de implantação. Com base nas definições versionadas de CI/CD, o Runner atua em quatro frentes principais. Na VM de Aplicação, gerencia os contêineres associados aos ambientes `terraplanner-stg`, `terraplanner-canary` e `terraplanner-prod`. Na VM de Borda, que constitui o ponto de entrada público da aplicação e na qual o NGINX opera como *proxy* reverso, atualiza o roteamento de tráfego entre esses ambientes. Além disso, armazena a imagem no Docker Registry privado, registra o respectivo *digest* e inicia a coleta dos eventos de observabilidade, posteriormente armazenados no PostgreSQL e visualizados no Grafana. Essa organização delimita as responsabilidades operacionais relativas à execução da aplicação, ao roteamento de tráfego, ao armazenamento de artefatos e ao registro dos eventos de observabilidade.

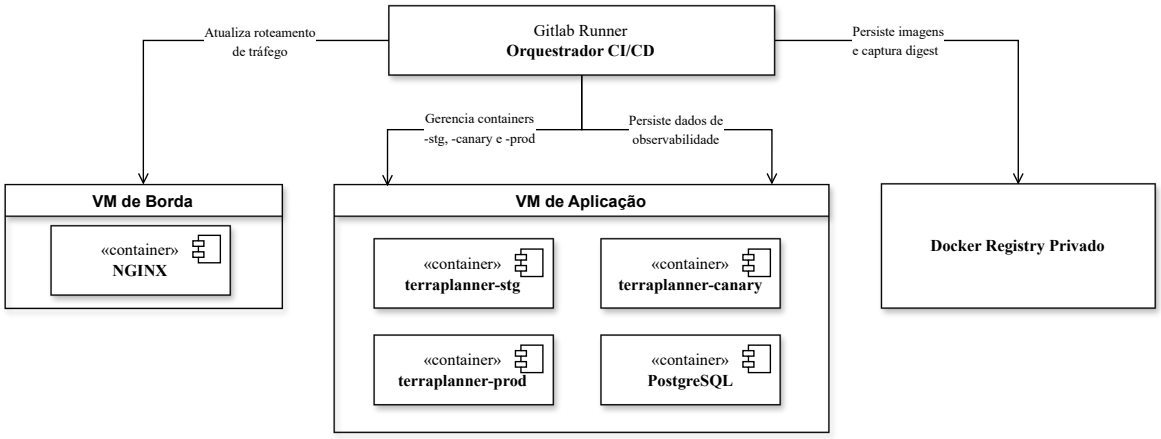

Figura 4.11 – Topologia lógica do *pipeline* de implantação

Fonte: Elaborado pelo autor (2026).

Para viabilizar a validação empírica sem interferir no projeto efetivamente disponibilizado em produção, a implementação foi conduzida em uma VM de Aplicação dedicada aos experimentos, isolada da infraestrutura operacional do projeto alvo. Nessa VM, instanciou-se uma cópia da versão mais recente do projeto alvo disponível em *staging*, preservando as configurações necessárias para manter paridade com o ambiente real. Essa separação reduz o risco de que falhas introduzidas durante a pesquisa afetem o trabalho da equipe de desenvolvimento ou a manutenção evolutiva do produto. Ao mesmo tempo, como a VM de Aplicação experimental opera na mesma rede e com recursos semelhantes aos da infraestrutura original, o arranjo preserva condições suficientes de reprodutibilidade para avaliar o *pipeline* de implantação proposto para o contexto do laboratório.

A implementação combina ferramentas consolidadas com artefatos desenvolvidos especificamente para este trabalho. GitLab CI/CD, Docker, Docker Compose, NGINX, Bun, ESLint, Jest, PostgreSQL e Grafana compõem a base tecnológica reutilizada para orquestração, empacotamento, validação, roteamento e visualização. A contribuição da pesquisa concentra-se na adaptação, na evolução e na articulação desses componentes por meio de modelos reutilizáveis de configuração em YAML, scripts de automação, relatórios estruturados, esquema relacional, *dashboards* e parâmetros experimentais que controlam SLIs, *feature flags*, promoção e *rollback*. Essa distinção delimita o que foi reaproveitado como infraestrutura e o que foi construído como parte do artefato avaliado.

4.5.2 Integração contínua e validação de qualidade

O estágio `quality` concentra os primeiros *quality gates* do fluxo, antes de qualquer construção de imagem ou alteração de ambiente. Ele é composto pelos *jobs* `lint`, `test_unit`, `test_integration`, `audit` e `coverage`, todos executados com Bun para manter a paridade com a *stack* do *front-end*. Essa posição no início do *pipeline* de implantação aplica o princípio de

detecção antecipada de falhas: inconsistências de *code style*, falhas em testes, vulnerabilidades críticas e não atendimento a limiares de cobertura de código são sinalizados antes que o *pipeline* de implantação consuma recursos de *build* e *release* (FOWLER; FOEMMEL, 2006; UGWUEZE; CHUKWUNWEIKE, 2025).

Cada *job* produz artefatos compatíveis com a interface do GitLab quando esse suporte está disponível. O `lint` publica o relatório de qualidade do código, permitindo que violações sejam exibidas na tela de MR; os *jobs* `test_unit` e `test_integration` executam suítes Jest separadas e geram relatórios no formato JUnit, possibilitando a exibição dos resultados de teste; e o `coverage` publica o relatório de cobertura de código produzido pelo Jest, viabilizando a exibição do percentual de cobertura. Como o relatório de auditoria produzido nativamente pelo Bun, um ambiente de execução JavaScript que integra recursos como gerenciamento de pacotes e execução de testes, não segue diretamente o formato esperado pelo GitLab, o *job* `audit` recebeu uma etapa de análise e conversão para garantir a integração com a UI. Além disso, para garantir a compatibilidade retroativa, optou-se por não reutilizar nem alterar *scripts*[6] preexistentes durante a implementação. Em vez disso, foram criados novos *scripts* nomeados de acordo com o padrão *namespaced scripts*, uma convenção amplamente adotada no ecossistema Node.js para favorecer a organização lógica, a hierarquia clara e a padronização da execução de tarefas (CHOJNACKI, 2024). Os *scripts* adicionados foram `lint:check`, destinado ao *job* de *lint*; `test:unit` e `test:integration`, responsáveis pela execução das suítes de testes; e `test:coverage`, utilizado para a geração do relatório de cobertura.

O controle de execução dos *jobs* de qualidade é governado por um sistema de *feature flags* versionadas em um arquivo YAML, mantido no repositório de templates e sujeito ao controle de aprovação definido por *code owners*, operando em dois níveis hierárquicos: (i) a *flag* `QUALITY_TESTS_ENABLE` funciona como um interruptor mestre, de modo que, quando definida como `"false"`, todos os *jobs* de qualidade são ignorados, independentemente das demais *flags*; e (ii) *flags* individuais dos tipos `*_ENABLE` e `*_BLOCKING` controlam, respectivamente, se um *job* específico é executado e se sua falha bloqueia o *pipeline* de implantação. Essa separação entre habilitação e obrigatoriedade permite desativar um *job* com problema de infraestrutura sem comprometer os demais controles e converter temporariamente um controle bloqueante em não bloqueante, sem alterar o código do *job*. A decisão reflete uma orientação estratégica dos responsáveis pelo TerraLAB: dado o contexto de alta rotatividade de integrantes e a presença recorrente de estudantes em estágio inicial da graduação, tornar os *gates* obrigatórios desde o primeiro momento poderia introduzir latência excessiva na adaptação dos desenvolvedores às novas validações.

No *job* `coverage`, o relatório de cobertura alimenta a página do MR e amplia a visibilidade da métrica para revisores. A configuração adota uma faixa de referência entre 60% e 90%, evitando tratar 100% de cobertura como meta absoluta. Essa decisão fundamenta-se na

[6] Comandos configurados no gerenciador de pacotes para automatizar tarefas de rotina do projeto.

Lei de Goodhart, formulada originalmente no contexto da política monetária, segundo a qual regularidades estatísticas observadas tendem a perder validade quando passam a ser submetidas a pressão para fins de controle (GOODHART, 1984). No contexto de desenvolvimento de software, autores como Fowler (2012) e Martin (2011) corroboram esse princípio ao alertarem que a imposição de métricas rígidas de cobertura pode gerar um falso senso de segurança, estimulando a criação de testes superficiais apenas para satisfazer a ferramenta. O intervalo adotado também se aproxima de práticas industriais documentadas por Ivanković et al. (2019), que descrevem o uso de limiares de 60%, 75% e 90% para classificar níveis de cobertura de código na infraestrutura de testes do Google.

4.5.3 Construção e versionamento dos artefatos

Após a validação inicial, o estágio `package` consolida a mudança em um artefato versionado. O *job* `create_env` implementa o versionamento semântico automático com base na análise do histórico Git. O algoritmo percorre os *commits* desde a última *tag* estável, inspeciona as mensagens que seguem o padrão *Conventional Commits* e determina o tipo de incremento. Além disso, foi criado um mecanismo por meio do qual o desenvolvedor pode sobrescrever o tipo de incremento com *flags* explícitas na mensagem do *commit*, a saber: `[MAJOR]`, `[MINOR]` e `[PATCH]`. Esse recurso permite controle manual em situações excepcionais sem alterar a convenção padrão.

O *job* `build_image` consome a variável `RC_TAG`, exportada pelo *job* `create_env`, e executa o comando `docker build` a partir do `Dockerfile` do projeto, utilizando uma compilação em múltiplos estágios (*multi-stage build*). No primeiro estágio, as dependências são instaladas e a aplicação é compilada, gerando os artefatos estáticos finais. No segundo estágio, apenas esses artefatos e o arquivo de configuração do NGINX são copiados. Dessa forma, o código-fonte e as dependências de desenvolvimento presentes no primeiro estágio são descartados, preservando-se apenas o *bundle* final e a configuração necessária para servir a aplicação. A imagem é publicada e marcada com a *tag* correspondente ao valor de `CI_COMMIT_SHA`, além de ser anotada com rótulos compatíveis com o padrão Open Container Initiative (OCI)[7]. Os rótulos compatíveis com o padrão registram: a versão da imagem, por meio de `org.opencontainers.image.version`, associada à variável `RC_TAG`; a revisão do código, em `org.opencontainers.image.revision`, correspondente a `CI_COMMIT_SHA`; o momento de criação, em `org.opencontainers.image.created`, representado pelo carimbo temporal Coordinated Universal Time (Tempo Universal Coordenado, UTC) do *build*; e a origem do artefato, em `org.opencontainers.image.source`, definida pela Uniform Resource Locator (Localizador Uniforme de Recursos, URL) do projeto no GitLab.

Após a publicação da imagem no *docker registry*, o *job* captura o *digest* criptográfico por meio do comando `docker inspect` e o exporta como a variável `IMAGE_DIGEST` no artefato `build.env`. Dessa forma, a mesma imagem, identificada de forma imutável por seu *digest*, é validada em *staging* e, em seguida, promovida ao canário e à trilha estável sem qualquer

[7] Disponível em: <https://docs.docker.com/build/metadata/annotations/>. Acesso em: 4 jul. 2026.

reconstrução entre ambientes, o que elimina a possibilidade de divergência entre o artefato testado e o artefato efetivamente implantado em produção (HUMBLE; FARLEY, 2010).

Dando sequência ao fluxo, o *job* `deploy_staging` consome o *digest* para materializar a implantação em *staging* em duas etapas. Primeiro, o arquivo `docker-compose.yml` do repositório é copiado para o diretório de execução na VM de aplicação, sobrescrevendo a cópia ali residente. Essa sincronização garante que a definição de serviços aplicada em tempo de *deployment* seja sempre a versão registrada no repositório. Segundo, o *job* conecta-se à VM de Aplicação por SSH, autentica-se no *docker registry* e executa o comando `docker-compose up -d -pull always terraplanner-stg`. A *flag* `-pull always` força o Docker a buscar novamente a imagem referenciada pelo *digest*, ainda que uma imagem com o mesmo nome já esteja armazenada em *cache* local na VM, reforçando a garantia de imutabilidade descrita no parágrafo anterior. Na mesma chamada, cinco variáveis de ambiente são propagadas, as quais o arquivo `docker-compose.yml` utiliza para compor as *labels* OCI no contêiner em execução, replicando os metadados de rastreabilidade já aplicados à imagem em tempo de *build*.

Após o *deployment*, o *job* `smoke_staging` valida o serviço recém-implantado ao verificar o *endpoint* `/deep/health` por meio de conexão direta à VM de aplicação. Em seguida, a execução dos testes E2E em homologação é controlada por três *feature flags*: `E2E_TESTS_STAGING_ENABLE`, que habilita ou desabilita o fluxo de testes; `WAIT_E2E_TESTS_STAGING`, que define a execução síncrona ou assíncrona do *pipeline* de QA; e `E2E_TESTS_STAGING_BLOCKING`, que determina se falhas devem interromper a implantação ou apenas gerar avisos. Assim, a imagem candidata é validada em homologação antes de se tornar elegível ao fluxo de produção por meio de *canary deployment*.

4.5.4 Implementação da Estratégia de Canary Deployment

A estratégia de *canary deployment* foi implementada em quatro fases, conforme sintetizado na Figura 4.12. A progressão avança somente quando o *job* anterior é concluído com sucesso; a ocorrência de falha relevante no *smoke test*, na verificação das SLIs ou na promoção aciona os mecanismos de recuperação descritos na subseção 4.5.5.

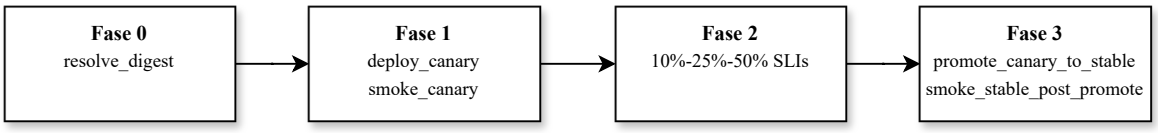

Figura 4.12 – Progressão de fases do *canary deployment*

Fonte: Elaborado pelo autor (2026).

Na Fase 0, o *job* `resolve_digest` recupera, na *branch* `main`, o *digest* da imagem já publicada no Docker Registry. Para isso, consulta-se o *header* Hypertext Transfer Protocol (HTTP) `Docker-Content-Digest`, que é retornado pela API após requisição ao *docker registry*,

evitando reconstruir a imagem em produção. O *job* também trata o caso de *merge commits* sintéticos: quando o *commit* corrente possui dois *commits* pais, utiliza-se o segundo pai como chave de busca, pois ele corresponde ao SHA efetivamente construído em `staging`; quando há apenas um pai, utiliza-se o próprio `CI_COMMIT_SHA`. Esse mecanismo garante que o *job* sempre localize o SHA que efetivamente originou o *build*, independentemente de como o *commit* chegou à *branch* `main`.

Na fase 1, o `deploy_canary` inicia o serviço `terraplanner-canary` na VM de Aplicação, sem direcionar-lhe tráfego público. Antes da criação do contêiner, o *Gitlab runner* aciona o *script* de roteamento na VM de Borda do NGINX com percentual de 0%, removendo alocações residuais de execuções anteriores. Em seguida, o Docker Compose recria apenas o serviço canário, utilizando a *flag* `-no-deps`, preservando o container `terraplanner-prod` no processo. Na sequência, o *job* `smoke_canary` verifica diretamente a porta interna do canário e consulta o *endpoint* `/deep/health`; a aprovação exige código HTTP 2xx e status de saúde válido no corpo da resposta, ficando a verificação de latência reservada à Fase 2.

Para viabilizar essa verificação, foi necessário incluir uma rota dedicada ao *endpoint* `/deep/health` no NGINX interno ao projeto *front-end*. Esse NGINX é responsável por servir os arquivos estáticos no próprio contêiner da aplicação e é distinto daquele que atua como *proxy* reverso na VM de Borda. Diferentemente de uma verificação trivial de disponibilidade, essa rota compõe uma resposta JavaScript Object Notation (JSON) estruturada a partir de duas verificações realizadas na própria camada do servidor *web*: a primeira atesta a disponibilidade do próprio processo do NGINX, condição implicitamente satisfeita sempre que o *endpoint* responde à requisição; a segunda confirma a presença do arquivo de entrada da aplicação no diretório de publicação, atestando que o *bundle* da versão implantada foi corretamente copiado para dentro da imagem e está pronto para ser servido. O *bundle* corresponde ao conjunto de arquivos estáticos, incluindo HyperText Markup Language (HTML), JavaScript, Cascading Style Sheets (CSS) e demais recursos gerados pelo processo de *build* da aplicação. É esse artefato, e não o código-fonte, que é copiado para dentro da imagem Docker e efetivamente servido pelo NGINX em produção. Assim, qualquer falha no processo de *build* ou na cópia dos arquivos para a imagem resultaria em um contêiner que é iniciado e responde na rede, mas que não possui a aplicação disponível para ser servida.

Na Fase 2, os *jobs* `verify_canary_10pct`, `verify_canary_25pct` e `verify_canary_50pct` atualizam progressivamente o roteamento do NGINX e coletam métricas de SLIs. A atualização é realizada por meio de SSH, diretamente na VM de Borda, com a invocação do *script* `nginx-canary-update.sh`. O *script* reescreve a configuração, executa o comando `nginx -t` e aplica `nginx -s reload` somente quando o comando anterior indica que a configuração é válida. Essa sequência reduz o risco de interrupção do *proxy reverso* em razão de erros de sintaxe.

O roteamento implementado no NGINX combina duas estratégias coexistentes, conforme o planejamento apresentado na subseção 4.4.1. A primeira é baseada em *cookie*: requisições

que contêm o *cookie* `tp_canary`, cujo valor é gerado localmente e persistido em uma variável de ambiente, são sempre direcionadas ao contêiner canário, independentemente do percentual configurado. A segunda estratégia é baseada em `split_clients`: o NGINX calcula um *hash* determinístico a partir da combinação entre o endereço Internet Protocol (IP) remoto e o cabeçalho `X-Forwarded-For`, mapeando o resultado um intervalo proporcional ao percentual configurado no espaço de *hash*. Desse modo, a mesma combinação de IP e cabeçalho é sempre alocada na mesma faixa, garantindo que um mesmo usuário permaneça consistentemente no canário ou na versão `stable` durante toda a fase, em vez de alternar a cada requisição. A prioridade do *cookie* sobre o `split_clients` é implementada por uma diretiva `map`, que resolve o *upstream* final: se o *cookie* estiver presente e válido, seu valor prevalece; caso contrário, o resultado do `split_clients` é utilizado. Esse comportamento é ilustrado no algoritmo 4.1.

Algoritmo 4.1: Resolução do *upstream* canário no NGINX

Input: $cookieRecebido$; $cookieEsperado$; $ipRemoto$; $xForwardedFor$; $percentual$
Output: *upstream* de destino: `canary` ou `stable`

```
/* split_clients: hash normalizado no intervalo [0, 100)          */
hash ← Hash(ipRemoto || xForwardedFor);
if hash < percentual then
    destinoPercentual ← canary;
else
    destinoPercentual ← stable;
/* diretiva map: cookie tem prioridade sobre o split_clients      */
if cookieRecebido = cookieEsperado then
    destinoFinal ← canary;
else
    destinoFinal ← destinoPercentual;
return destinoFinal
```

Fonte: Elaborado pelo autor, com base na lógica combinada de `split_clients` e da diretiva `map` (2026).

A validação dos SLIs, realizada pelos *jobs* de *rollout*, foi implementada por meio da lógica de *polling* direto ao *endpoint* `/deep/health` da VM Aplicação. Nesse processo, são validados o código HTTP retornado e o tempo de resposta, sendo uma amostra considerada aprovada apenas quando apresenta código de resposta 2xx e validações de SLIs bem-sucedidas. A configuração de execução desse *polling* pode ser controlada por meio de variáveis versionadas no próprio repositório de *templates*, a saber: `SLI_OBSERVATION_WINDOW_SECONDS`, que define a duração da janela de observação de cada fase; `SLI_SAMPLE_INTERVAL_SECONDS`, que define o intervalo entre as coletas de amostras dentro dessa janela; `SLI_MIN_PASS_RATE`, que define

a taxa mínima de amostras aprovadas para que a fase seja considerada saudável; e `SLI_DEEP_HEALTH_TIMEOUT_MS`, que define a latência máxima aceitável para a resposta do `/deep/health`.

Na Fase 3, o *job* `promote_canary_to_stable` promove, para o serviço `terraplanner-prod`, a imagem aprovada após operar com 50% do tráfego. A promoção é manual por padrão e controlada pela variável `CANARY_PROMOTE_MANUAL: true`, pois a transição para a trilha estável implica maior custo de reversão. Sua execução é acionada manualmente pelo operador, por meio de um botão na interface do GitLab, e torna-se disponível somente após o *job* anterior aprovar os SLIs observados nessa fração de tráfego. Esse intervalo manual permite, ainda, que o operador ou a equipe de QA valide a versão canário por meio do estratégia de *cookies*. Quando se define `CANARY_PROMOTE_MANUAL: false`, a mesma etapa passa a ser disparada automaticamente assim que os SLIs são aprovados, sem alteração no código do *job*. Após a promoção, o roteamento para o canário é atualizado para 0%, o contêiner `terraplanner-canary` é removido e o arquivo `deployment_history.json`, residente na VM de aplicação, é atualizado com o ajuste dos *digests* `current` e `previous`. Em seguida, o *job* `smoke_stable_post_promote` executa uma validação de saúde no ambiente recém-promovido.

4.5.5 Implementação do Rollback Automatizado

O *pipeline* de implantação implementa dois mecanismos de recuperação: *rollback* do canário antes da promoção e *rollback* da trilha estável após a promoção. O primeiro reduz a exposição a falhas detectadas durante o *rollout*; o segundo permite recuperar o ambiente produtivo quando a regressão se manifesta apenas após a promoção do canário.

O *job* `rollback_canary_auto` é acionado pela diretiva `when: on_failure` quando qualquer *job* do *rollout* falha, incluindo o *smoke test* do canário, os três *jobs* de verificação de SLIs e a própria promoção do canário para o ambiente produtivo. Sua sequência é curta e deliberadamente conservadora: a VM de Borda recebe uma atualização para encaminhar 0% do tráfego ao *container* canário; em seguida, a VM de Aplicação remove o `terraplanner-canary`, caso exista. Essa operação é idempotente, pois verifica a existência do *container* antes de removê-lo, permitindo repetir o *rollback* manualmente sem falha caso ele já tenha sido removido. Concluída a remoção, o *job* verifica o estado de saúde do `terraplanner-prod` em até três tentativas. Esse fluxo é sintetizado na Figura 4.13.

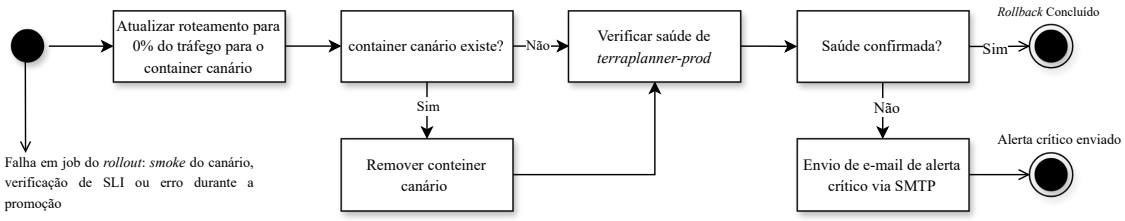

Figura 4.13 – Fluxograma de *rollback* do ambiente canário

Fonte: Elaborado pelo autor (2026).

O *rollback* da trilha estável é tratado por dois *jobs*: `rollback_stable`, que permanece manual e pode ser acionado após uma promoção já concluída; e `rollback_stable_auto`, que executa a mesma lógica quando `smoke_stable_post_promote` falha imediatamente após a promoção. A separação entre os dois *jobs* decorre do modelo de regras do GitLab CI, que exige definições distintas para combinar disponibilidade manual e acionamento automático por falha.

A sequência de execução de ambos compreende seis etapas: (i) leitura do campo `previous` do arquivo `deployment_history.json` na VM de Aplicação, a fim de recuperar o *digest* da versão anterior; (ii) validação da existência desse *digest* no *Docker Registry*; (iii) substituição do `terraplanner-prod` utilizando o *digest* anterior; (iv) reconciliação de *tags* no *Docker Registry*, em que o *digest* anterior recebe a *tag* `:stable`, e o *digest* da versão substituída recebe `:stable-failed`, preservando a rastreabilidade sem ambiguidade; (v) atualização do `deployment_history.json` com o novo estado; e (vi) nova verificação de saúde do `terraplanner-prod`, realizada em até três tentativas. Essa sequência é sintetizada na Figura 4.14.

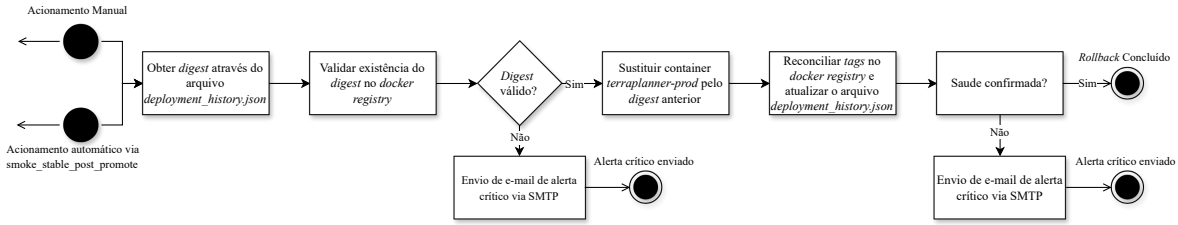

Figura 4.14 – Fluxograma de *rollback* do ambiente de produção

Fonte: Elaborado pelo autor (2026).

Em caso de falha no próprio *job* de *rollback*, por exemplo, por perda de conectividade com o *host* de acesso ao NGINX ou com a VM de Aplicação, o *handler* `_on_error` do *runner* envia um e-mail de alerta crítico por Simple Mail Transfer Protocol (SMTP) ao endereço configurado por meio da variável `ALERT_EMAIL`, contendo as instruções necessárias para intervenção manual. Essa notificação por e-mail corresponde à primeira implementação do mecanismo de alerta, adotada por sua simplicidade de configuração e por não introduzir dependências externas ao *pipeline* de implantação. Como direção de evolução, considera-se desejável a definição de um canal oficial de comunicação do TerraLAB, por exemplo, Slack, WhatsApp ou outra ferramenta de mensagens, destinado a centralizar e persistir esse alerta e outros que venham a ser criados.

4.5.6 Implementação da Coleta e Visualização das Métricas DORA

A partir da arquitetura de observabilidade apresentada na subseção 4.4.2, esta subseção detalha sua materialização técnica no *pipeline* implementado. A solução combina o *job* agendado `collect_dora_metrics`, uma base PostgreSQL dedicada e painéis no Grafana. Assim, a ênfase deixa de recair sobre o modelo conceitual dos eventos e passa a se concentrar em como os artefatos

gerados pela implantação e pelos *rollbacks* são persistidos, relacionados e disponibilizados para consulta.

4.5.6.1 Persistência dos eventos

A persistência foi implementada por meio de duas tabelas principais, mantidas em um *container* PostgreSQL dedicado na VM de Aplicação. A tabela `dora_deployments` armazena eventos de implantação, enquanto a `dora_incidents` armazena eventos de remediação associados a *rollbacks*. A Figura 4.15 sintetiza esse esquema relacional e explicita os campos utilizados para reconstruir a origem de cada evento, como projeto, ambiente, *commit*, *pipeline*, *job* de origem e instantes relevantes. Cada registro utiliza a combinação entre `pipeline_id` e `source_job` como chave de unicidade, o que torna a ingestão idempotente: reprocessar os mesmos artefatos em execuções futuras do coletor não duplica eventos na base analítica.

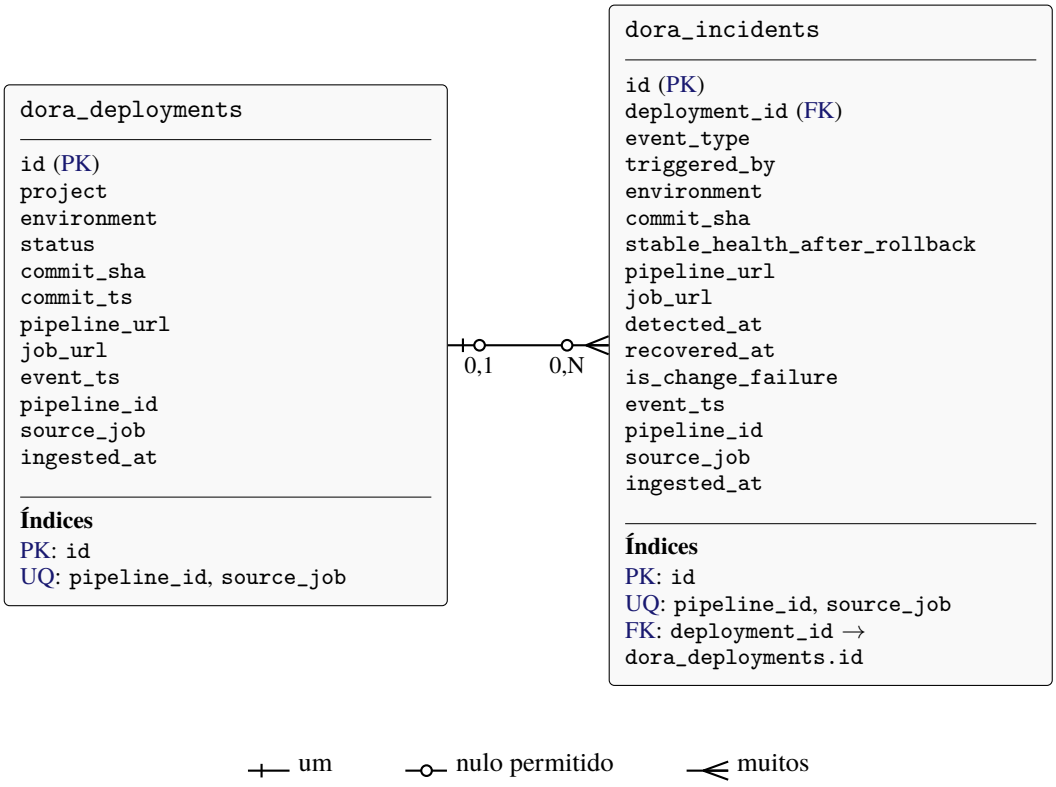

Figura 4.15 – Esquema relacional da base analítica utilizada para métricas DORA

Fonte: Elaborada pelo autor (2026).

4.5.6.2 Coleta, cálculo e visualização das quatro métricas

A coleta é executada por um *job* agendado, dissociado do fluxo de implantação, e ocorre automaticamente em dois momentos do dia: às 6h e às 18h. Em vez de calcular métricas a cada implantação, um processo periódico percorre os *pipelines* de implantação mais recentes da *branch main*, localiza os artefatos de evento gerados pelos *jobs* de implantação e de *rollback* e os insere no banco de dados. Essa dissociação evita acoplar o tempo de resposta da implantação à

latência adicional de uma etapa de agregação de métricas. Como o *runner* não possui acesso direto, via rede, ao banco de dados, o lote de inserções gerado localmente é transferido para a VM de aplicação e executado dentro do próprio *container* do banco de dados. A partir dos eventos persistidos nas duas tabelas descritas na Figura 4.15, as quatro métricas canônicas do programa DORA são calculadas de forma automatizada, sendo cada uma delas obtida por meio da leitura direta dos campos já registrados no banco de dados.

A DF resulta de uma contagem simples: quantas linhas da tabela `dora_deployments` têm o campo `environment` igual a `produção` e o campo `status` igual a `sucesso` dentro do período selecionado. Essa contagem é exibida diariamente no painel de tendência e, no painel de valor agregado, é dividida pelo número de dias do período, resultando em uma média de implantações por dia.

O LTFC parte de dois instantes já registrados para cada implantação bem-sucedida: o momento em que o *commit* implantado foi criado, representado pelo campo `commit_ts`, e o momento em que essa mudança chegou, de fato, ao ambiente produtivo, registrado no campo `event_ts`. A diferença entre esses dois instantes, convertida em horas, corresponde ao tempo de entrega daquela implantação específica; o painel de tendência exibe esse valor para cada implantação, e a linha de valor agregado calcula a média de todos os intervalos observados no período.

A CFR depende da relação entre as duas tabelas: para cada implantação em ambiente produtivo, verifica-se se ela está associada, pelo campo `deployment_id`, a algum registro da tabela `dora_incidents` marcado como falha de mudança, registrada por meio do campo `is_change_failure`, definido como `true`. A métrica corresponde ao percentual de implantações nessa condição, apurado para a janela de tempo selecionada.

O MTTR utiliza exclusivamente a tabela `dora_incidents` e considera apenas os incidentes classificados como falha de mudança que chegaram, de fato, a ser recuperados, ou seja, aqueles em que tanto o instante de detecção, representado pelo campo `detected_at`, quanto o instante de recuperação, persistido no campo `recovered_at`, estão preenchidos. A diferença entre essas duas marcações de tempo, convertida em minutos, indica por quanto tempo o ambiente permaneceu degradado; o painel de tendência apresenta esse valor para cada incidente, e a linha de valor agregado calcula a média dos intervalos válidos no período, excluindo os casos em que o próprio *rollback* não teve sucesso.

Essas quatro métricas compõem um *dashboard* interativo, consultável para diferentes janelas de tempo, que também classifica o período corrente segundo as faixas de desempenho definidas por DeBellis et al. (2024), isto é, *Elite*, *High*, *Medium* e *Low*.

4.6 Metodologia da Avaliação Experimental

Esta seção descreve a metodologia experimental adotada para avaliar, de forma controlada, a capacidade do *pipeline* de implantação proposto (Seção 4.5) de detectar falhas injetadas em diferentes estágios do *canary rollout* e de acionar automaticamente o mecanismo de *rollback* correspondente. A avaliação compreende cinco cenários experimentais: um cenário de controle, no qual nenhuma falha é injetada e a promoção completa é esperada (Cenário 2); três cenários de injeção de falha, cada um visando um ponto de detecção distinto do *pipeline* (Cenários 1a, 1b e 1c); e um cenário de reversão pós-promoção, acionado manualmente (Cenário 3). Os resultados obtidos a partir da execução deste protocolo são apresentados na Seção 4.7.

4.6.1 Configuração dos testes

Os experimentos foram executados no ambiente de testes utilizado também para a implementação do *pipeline* de implantação proposto. Esse ambiente é isolado do ambiente real de produção do TerraPlanner, o que representa uma delimitação relevante para a interpretação dos resultados, conforme discutido na subseção 4.6.3. Os três cenários de injeção de falha descritos a seguir exploram o mesmo mecanismo de verificação de SLIs, implementado no *job* `verify_canary_*pct`, apresentado na subseção 4.5.4. Os cenários diferem exclusivamente quanto ao tipo de falha injetada, permitindo isolar o comportamento do mecanismo de detecção em diferentes pontos do fluxo de implantação. A Tabela 4.2 descreve as configurações utilizadas para a execução dos testes, correspondentes às quatro variáveis de ambiente introduzidas na subseção 4.5.4: a duração da janela de observação, o intervalo entre coletas de amostras dentro dessa janela, a taxa mínima de amostras aprovadas exigida para que a fase seja considerada saudável e a latência máxima aceitável por amostra individual. Os valores adotados nesta bateria de testes foram deliberadamente reduzidos em relação às configurações típicas propostas para o ambiente produtivo, definidas na subseção 3.5.2 a partir das Equações 3.6 e 3.7, de modo a tornar cada cenário experimental mais ágil e viabilizar a execução completa dos cinco cenários dentro do tempo disponível para a coleta, preservando amostragem suficiente para capturar as falhas injetadas nos Cenários 1a, 1b e 1c.

Tabela 4.2 – Parâmetros globais de avaliação de SLIs nos cenários experimentais

Parâmetro	**Valor**
Limiares de avaliação de SLIs	
Janela de observação (`SLI_OBSERVATION_WINDOW_SECONDS`)	60 s
Intervalo entre amostras (`SLI_SAMPLE_INTERVAL_SECONDS`)	10 s
Taxa mínima de aprovação (`SLI_MIN_PASS_RATE`)	80 %
Limiar de latência por amostra (`SLI_DEEP_HEALTH_TIMEOUT_MS`)	1500 ms

Fonte: Elaborado pelo autor (2026).

Além dos limiares de SLI, adotou-se como critério operacional que os *rollbacks* automá-

ticos dos Cenários 1a, 1b e 1c restaurassem o ambiente estável em tempo inferior a dois minutos após a detecção da falha, prazo compatível com a prática relatada por Basiri et al. (2019), cujos mecanismos de segurança automatizados priorizam a reação mais rápida possível para limitar o raio de impacto de uma falha em produção. Esse critério não se aplica ao Cenário 3, pois nele a reversão pós-promoção é acionada manualmente e avalia a disponibilidade do mecanismo de contingência, não a reação automática do *pipeline* de implantação.

4.6.2 Descrição dos cenários experimentais

Esta etapa do método experimental tem como objetivo descrever os cenários utilizados para avaliar o comportamento do *pipeline* de implantação proposto diante de diferentes condições de execução. Em todos os cenários, observa-se o comportamento dos mecanismos de verificação descritos na subseção 4.5.4 e dos mecanismos de reversão descritos na subseção 4.5.5.

4.6.2.1 Cenário 1a: Falha de taxa de erro

O Cenário avalia a capacidade do *pipeline* de implantação de impedir que uma versão com falha manifesta receba qualquer parcela do tráfego real. A falha injetada consiste em substituir a resposta do *endpoint* `/deep/health` por um código de status HTTP 500, por meio de uma *branch* de falha dedicada no repositório da aplicação *front-end*. Essa alteração sobrescreve o bloco correspondente na configuração do NGINX incorporado ao contêiner. Como a falha já é observável na primeira verificação de saúde do contêiner canário, o ponto de detecção esperado é o *job* `smoke_canary`, executado quando ainda há 0% do tráfego real direcionado ao canário. Espera-se que o *smoke test* falhe imediatamente, impedindo a execução de qualquer *job* `verify_canary_*pct`, e que o *job* `rollback_canary_auto` seja acionado automaticamente, restaurando 100% do tráfego para o ambiente estável e removendo o contêiner canário em aproximadamente 2 minutos. O procedimento consiste em mesclar a *branch* de falha à *branch* `main` e registrar o instante de início do *pipeline*, o instante em que o *smoke test* reporta a falha e o instante de conclusão do *rollback*. As evidências esperadas são o artefato `smoke-canary-error.json`, que reflete a falha injetada, e o artefato `incident_event.json`, gerado pelo *rollback*.

4.6.2.2 Cenário 1b: Falha de latência

Este cenário avalia uma degradação projetada para não ser detectada pela verificação inicial de saúde, tornando-se observável apenas quando uma parcela real de tráfego passa a ser roteada para o canário. A falha injetada consiste na aplicação da diretiva `limit_rate 200` do NGINX no mesmo *endpoint* `/deep/health`, por meio de uma *branch* dedicada à falha, restringindo a 200 bytes/s a transferência da resposta de aproximadamente 500 bytes, o que eleva a latência observada a aproximadamente 2.500 ms. A distinção em relação ao Cenário 1a é intencional. O *job* `smoke_canary` opera com uma tolerância de tempo maior e apenas confirma a entrega do corpo da resposta; por isso, espera-se que ele aprove a versão nesse cenário. Já o

job `verify_canary_10pct` mede explicitamente o tempo total da requisição e o compara ao limiar definido por `SLI_DEEP_HEALTH_TIMEOUT_MS`; por esse motivo, espera-se que a falha seja detectada apenas nesse estágio, quando o canário já recebe 10% do tráfego real. O resultado esperado é a reprovação da etapa de 10% por violação do limiar de latência, seguida da execução automática do *rollback*. As evidências esperadas incluem o relatório `sli_report_10pct.json` e o arquivo `incident_event.json`.

4.6.2.3 Cenário 1c: Falha de disponibilidade

O cenário avalia a detecção de uma falha total de disponibilidade do contêiner canário, em contraste com as falhas de conteúdo e de desempenho analisadas nos cenários anteriores. Essa falha é introduzida por meio de uma intervenção operacional direta: o contêiner `terraplanner-canary` é interrompido e reiniciado durante a janela de observação referente ao incremento de 10%, de modo a produzir ao menos duas amostras que indiquem falha de conexão dentro da janela de 60 s. Prevê-se que essas amostras sejam registradas sem código de resposta HTTP válido, reduzindo a taxa de aprovação do incremento a um valor inferior ao limiar mínimo de 80%. Com isso, o mecanismo de *rollback* automático deve restaurar integralmente o tráfego para o ambiente estável. As evidências esperadas consistem no relatório `sli_report_10pct.json`, contendo amostras com falha de conexão e taxa de aprovação abaixo do limiar mínimo, e no arquivo `incident_event.json`, referente ao *rollback* correspondente.

4.6.2.4 Cenário 2: Promoção bem-sucedida

Este cenário funciona como controle experimental. Nele, uma versão íntegra da aplicação, sem qualquer falha injetada, é submetida ao fluxo completo de *canary deployment*. O objetivo é verificar se o mecanismo de validação dos SLIs aprova corretamente uma versão íntegra em todos os incrementos de tráfego e permite a conclusão da promoção para o ambiente estável. O procedimento consiste em integrar um *commit* à *branch* `staging` e, posteriormente, mesclá-la à *branch* `main`, registrando-se o instante de início do *pipeline* de implantação. Na sequência, acompanham-se os *jobs* de qualidade, empacotamento, implantação em *staging*, resolução de *digest*, implantação canária e *smoke test*. Posteriormente, observam-se as três janelas de verificação dos SLIs e, uma vez aprovadas, a promoção é acionada manualmente e seu instante é registrado. As evidências esperadas compreendem os relatórios `sli_report_{10,25,50}pct.json` e o artefato `deployment_event.json`, gerado pela promoção.

4.6.2.5 Cenário 3: Rollback pós-promoção

O Cenário 3 avalia o mecanismo de reversão do ambiente estável, representado pelo *job* `rollback_stable`. Diferentemente dos cenários anteriores, esse mecanismo opera após a conclusão da promoção e é acionado exclusivamente de forma manual. Sua execução pressupõe a conclusão bem-sucedida do Cenário 2, de modo que o arquivo `deployment_history.json` já

contenha o registro do *digest* correspondente à versão anterior à promoção mais recente. Espera-se que a execução manual do *job* `rollback_stable` restaure corretamente a versão anterior no ambiente `terraplanner-prod`, verificando a integridade do *endpoint* `/deep/health` após a reversão. O procedimento consiste em capturar os valores dos campos `current` e `previous` do arquivo `deployment_history.json`, bem como a imagem em execução antes do acionamento. Em seguida, o *job* é executado manualmente, com o preenchimento do campo `ROLLBACK_REASON` para registrar a justificativa da reversão. Após a conclusão da execução, verifica-se que o campo `current` do histórico passa a corresponder ao valor anteriormente registrado em `previous`, confirmando a restauração da versão anterior e a preservação da integridade do ambiente estável. Como evidência, espera-se a geração do arquivo `incident_event.json` pelo *job*, contendo os campos que identificam o *digest* restaurado e o motivo informado para o acionamento.

4.6.3 Ameaças à Validade

interpretação dos resultados apresentados na Seção 4.7 deve considerar um conjunto de limitações inerentes ao delineamento dos experimentos controlados de injeção de falhas. Em primeiro lugar, os cinco cenários foram executados em um projeto de teste dedicado, e não no ambiente de produção do TerraPlanner; portanto, conclusões sobre o comportamento do *pipeline* de implantação sob condições reais de tráfego e concorrência não podem ser diretamente extrapoladas dos resultados aqui obtidos, o que constitui uma ameaça à validade externa.

Em segundo lugar, os contêineres `terraplanner-canary`, `terraplanner-prod` e `terraplanner-stg` são executados lado a lado na mesma VM de aplicação, compartilhando os mesmos recursos de CPU, RAM, armazenamento e rede do hospedeiro. Essa coabitação introduz um efeito de *vizinho ruidoso*[8]: uma falha injetada que aumente o consumo de recursos ou o tempo de resposta do contêiner canário, como a latência artificial do Cenário 1b, pode degradar, ainda que marginalmente, o desempenho do contêiner estável observado simultaneamente, e reciprocamente. As medições de latência e disponibilidade coletadas pelos *jobs* de verificação de SLIs podem, assim, refletir em parte a carga momentânea do hospedeiro compartilhado, e não exclusivamente o comportamento isolado da versão candidata, o que representa uma ameaça à validade interna das medições de desempenho.

Em terceiro lugar, os três tipos de falha injetadas constituem um subconjunto controlado e deliberadamente simples dos possíveis modos de falha em produção. Degradações mais sutis ou compostas, como vazamento progressivo de memória, esgotamento de conexões com dependências externas, degradação gradual de desempenho sob carga real ou falhas intermitentes, não foram avaliadas e podem exigir comportamento distinto do mecanismo de verificação de SLIs, atualmente ajustado para reagir a sintomas relativamente abruptos e sustentados durante toda a janela de observação. Essa cobertura limitada representa uma ameaça à validade de construto

[8] Antipadrão em que a atividade de um inquilino degrada o desempenho de outro em um sistema compartilhado (DOWNS, 2025).

quanto à alegação mais ampla de que o *pipeline* de implantação detecta falhas de maneira geral, e não apenas os três tipos efetivamente exercitados nos experimentos.

4.7 Avaliação Experimental e Resultados

Esta seção apresenta os resultados dos cinco cenários experimentais descritos na Seção 4.6, evidenciando o comportamento do mecanismo de verificação de SLIs e do *rollback* automatizado do *pipeline* de implantação proposto diante de falhas injetadas em diferentes pontos do fluxo de *canary deployment*, bem como o comportamento do *rollback* manual acionado após a promoção. A subseção 4.7.1 apresenta o Cenário 1a, referente à falha de taxa de erro detectada na verificação inicial de saúde do canário; a subseção 4.7.2 descreve o Cenário 1b, relativo à falha de latência detectada no primeiro incremento de tráfego; a subseção 4.7.3 trata do Cenário 1c, referente à falha de disponibilidade do contêiner canário; a subseção 4.7.4 apresenta o Cenário 2, cenário de controle sem falha injetada; e a subseção 4.7.5 encerra a seção com o Cenário 3, referente ao *rollback* manual pós-promoção.

4.7.1 Cenário 1a — Falha de Taxa de Erro

Este cenário submeteu ao *pipeline* de implantação proposto uma versão do *front-end* com o *endpoint* `/deep/health` sobrescrito para reportar falha, conforme o procedimento descrito na subseção 4.6.2. A Tabela 4.3 apresenta os instantes registrados durante a execução.

Tabela 4.3 – Cenário 1a — Tempos e evento de *rollback* registrados

Evento	**Valor**
Início do *pipeline*	23:09:24 (UTC)
Detecção da falha	23:13:01 (UTC)
Conclusão do *rollback*	23:14:15 (UTC)
MTTR ($T_{recuperacao} - T_{deteccao}$)	1 min 14 s

Fonte: Elaborado pelo autor (2026).

O *job* `smoke_canary` reprovou a versão candidata na primeira verificação de saúde do canário, ainda sem roteamento de tráfego real. Como consequência, os *jobs* `verify_canary_*pct` não foram executados, conforme previsto. O MTTR, de 1 minuto e 14 segundos, medido entre os campos `detected_at` e `recovered_at` do artefato `incident_event.json`, permaneceu muito abaixo do limiar de dois minutos definido no protocolo experimental, evidenciando a celeridade do *rollback* automático quando a falha é detectada no estágio inicial do fluxo de canário.

4.7.2 Cenário 1b — Falha de Latência

O Cenário 1b avaliou a detecção de uma degradação de latência imperceptível à verificação inicial de saúde, tornando-se observável apenas no primeiro incremento de tráfego real, conforme

descrito na subseção 4.6.2. A Tabela 4.4 apresenta os instantes registrados durante a execução.

Tabela 4.4 – Cenário 1b — Tempos e evento de *rollback* registrados

Evento	**Valor**
Início do *pipeline*	23:51:13 (UTC)
Detecção da falha	23:56:55 (UTC)
Conclusão do *rollback*	23:58:11 (UTC)
MTTR ($T_{recuperacao} - T_{deteccao}$)	1 min 16 s

Fonte: Elaborado pelo autor (2026).

Diferentemente do cenário anterior, o *job* `smoke_canary` aprovou a versão candidata, confirmando que a degradação de latência não é perceptível nessa verificação inicial. A reprovação ocorreu apenas no primeiro incremento de tráfego, conforme detalhado na Tabela 4.5. A taxa de aprovação de 0% e a latência média de 3.003 ms, muito superior ao limiar de 1.500 ms definido na Tabela 4.2, evidenciam que a degradação artificial de aproximadamente 2.500 ms na transferência da resposta, descrita na subseção 4.6.2, foi corretamente capturada pelo *job* `verify_canary_10pct`, responsável por medir o tempo total da requisição. O MTTR de 1 minuto e 16 segundos permanece na mesma ordem de grandeza observada no Cenário 1a, indicando que a celeridade do mecanismo de *rollback* não depende do estágio de detecção nos cenários avaliados.

Tabela 4.5 – Cenário 1b — Relatório de SLI do incremento de 10 %

Campo	**Valor observado**
Amostras totais / aprovadas	4 / 0
Taxa de aprovação (limiar mínimo: 80 %)	0 %
Latência média observada (limiar: 1.500 ms)	3.003 ms
`canary_pct_at_failure`	10 %

Fonte: Elaborado pelo autor (2026).

4.7.3 Cenário 1c — Falha de Disponibilidade

O Cenário 1c avaliou a detecção de uma falha total de disponibilidade do contêiner canário, provocada pela sua interrupção manual durante a janela de observação do incremento de 10 %, conforme descrito na subseção 4.6.2. A Tabela 4.6 apresenta os instantes registrados durante a execução.

Tabela 4.6 – Cenário 1c — Tempos e evento de *rollback* registrados

Evento	**Valor**
Início do *pipeline*	22:25:23 (UTC)
Detecção da falha	22:31:06 (UTC)
Conclusão do *rollback*	22:32:23 (UTC)
MTTR ($T_{recuperacao} - T_{deteccao}$)	1 min 17 s

Fonte: Elaborado pelo autor (2026).

Assim como na subseção 4.7.2, o *job* `smoke_canary` aprovou a versão candidata, uma vez que a falha foi introduzida somente após a implantação canária, durante a coleta de amostras referente ao incremento de 10%. Os resultados da verificação de SLI encontram-se na Tabela 4.7. A interrupção do contêiner `terraplanner-canary` durante a janela de 60 segundos produziu duas das quatro amostras coletadas sem resposta válida, reduzindo a taxa de aprovação para 50%, valor inferior ao limiar mínimo de 80% estabelecido na Tabela 4.2. O resultado confirma que o mecanismo de verificação de SLI reage tanto a falhas de conteúdo e de desempenho, como demonstrado nos Cenários 1a e 1b, quanto a falhas de indisponibilidade total do contêiner canário. O MTTR de 1 minuto e 17 segundos mantém-se na mesma ordem de grandeza observada nos dois cenários anteriores, reforçando a consistência do tempo de reação do *rollback* automático independentemente da natureza da falha injetada.

Tabela 4.7 – Cenário 1c — Relatório de SLI do incremento de 10 %

Campo	**Valor observado**
Amostras totais / aprovadas	4 / 2
Taxa de aprovação (limiar mínimo: 80 %)	50 %
`canary_pct_at_failure`	10 %

Fonte: Elaborado pelo autor (2026).

4.7.4 Cenário 2 — Promoção Bem-Sucedida

O Cenário 2 foi executado com versão saudável do *front-end* TerraPlanner submetida ao *canary deployment* completo. Os *timestamps* registrados durante a execução são apresentados na Tabela 4.8.

Tabela 4.8 – Cenário 2 — Timestamps registrados durante a execução

Evento	**Valor**
Início do *pipeline* de implantação	23:01:44 (UTC)
Acionamento manual da promoção	23:07:26 (UTC)
Conclusão do *pipeline* de implantação	23:08:52 (UTC)
Duração até promoção manual	5 min 42 s
Duração do *job* de promoção	1 min 26 s
Duração total do *pipeline* de implantação	7 min 8 s

Fonte: Elaborado pelo autor (2026).

O intervalo de 5 minutos e 42 segundos até a promoção manual abrange os estágios de resolução de *digest*, inicialização do contêiner canário, *smoke tests* no contêiner canário e as três janelas de verificação de SLIs. O *job* `promote_canary_to_stable` foi concluído em 86 segundos, incluindo a atualização do `terraplanner-prod`, a zeragem do roteamento no NGINX, a remoção do contêiner canário, a atualização do `deployment_history.json` e a publicação do artefato `deployment_event.json`.

4.7.5 Cenário 3 — Rollback Pós-Promoção

O Cenário 3 avaliou o mecanismo de *rollback* do ambiente estável, acionado manualmente após a conclusão do Cenário 2, conforme descrito na subseção 4.6.2. Os instantes registrados durante a execução do *job* `rollback_stable` podem ser observados na Tabela 4.9.

Tabela 4.9 – Cenário 3 — Tempos registrados durante o *rollback* manual

Evento	**Valor**
Início do *rollback* em *stable*	23:44:42 (UTC)
Conclusão do *rollback* em *stable*	23:45:27 (UTC)
Duração do *rollback* em *stable*	45 s

Fonte: Elaborado pelo autor (2026).

A reversão do ambiente estável envolveu a substituição da imagem em execução pelo *digest* imediatamente anterior, registrado no arquivo `deployment_history.json`. O *digest* restaurado pelo *rollback* corresponde exatamente ao *digest* promovido ao ambiente estável ao término do Cenário 2, confirmando que o mecanismo `rollback_stable` reverteu o ambiente para a versão imediatamente anterior à promoção mais recente. A verificação do *endpoint* `/deep/health` após a reversão retornou HTTP 200, atestando a integridade do ambiente restaurado. A duração de 45 segundos entre o acionamento e a conclusão do *job*, ainda que não diretamente comparável ao MTTR dos *rollbacks* automáticos dos Cenários 1a a 1c por decorrer de acionamento manual, evidencia que a operação de reversão do ambiente estável também é célere quando o *digest* anterior já está disponível no registro privado de imagens. Ressalva-se, contudo, que esse tempo foi obtido

com o procedimento acionado por quem já conhecia o *job* `rollback_stable`; um operador sem essa familiaridade tenderia a levar mais tempo para localizá-lo e preencher corretamente o campo `ROLLBACK_REASON`, o que relativiza a generalização direta dessa duração para cenários reais de incidente.

4.8 Comparação entre a Linha de Base e o Pipeline Proposto

Esta seção compara quantitativamente o desempenho do *pipeline* de implantação proposto ao da linha de base apurada na seção 4.3, evidenciando a evolução das quatro métricas DORA após a implementação do *Canary Deployment* com *rollback* automatizado descrito na seção 4.5. A subseção 4.8.1 descreve o procedimento de coleta adotado na janela piloto de observação. Em seguida, a subseção 4.8.2 apresenta a evolução da DF, a subseção 4.8.3 analisa a CFR, a subseção 4.8.4 caracteriza o LTFC e a subseção 4.8.5 quantifica o MTTR, cada uma delas acompanhada do respectivo painel do *dashboard* Grafana implementado na subseção 4.5.6, utilizado como evidência visual do cálculo apresentado.

4.8.1 Procedimento de Coleta na Janela Piloto de Observação

Diferentemente da linha de base, apurada por extração retrospectiva via `python-gitlab` sobre o histórico de 2024, descrito na seção 4.3, as métricas do *pipeline* de implantação proposto foram coletadas de forma automatizada pela própria camada de observabilidade implementada, descrita na subseção 4.5.6: o *job* agendado `collect_dora_metrics` persiste os eventos de implantação e de remediação nas tabelas `dora_deployments` e `dora_incidents`, e os painéis do Grafana aplicam sobre esses registros as mesmas definições operacionais estabelecidas nas Equações 3.1 a 3.5. A janela piloto considerada compreende 14 dias corridos, entre 27 de junho e 10 de julho de 2026, período no qual foram registradas 14 implantações no ambiente produtivo do *pipeline* de implantação proposto, mantido no mesmo projeto de teste dedicado já utilizado nos cenários experimentais da seção 4.6.

As implantações computadas na janela correspondem a mudanças reais aplicadas ao projeto de teste ao longo do período, cada uma delas atendendo a um requisito técnico concreto já discutido neste trabalho, e não a *commits* triviais criados apenas para gerar volume de dados. Os refactors e os ajustes de tipagem responderam às exigências de qualidade impostas pelos *jobs* `lint` e `test_unit` do estágio `quality`, corrigindo inconsistências de estilo e de tipos sinalizadas por essas verificações; a criação de *scripts* de apoio seguiu a convenção de *namespaced scripts* adotada no projeto, suprindo automações exigidas pelo próprio fluxo de qualidade; as melhorias no NGINX interno do contêiner decorreram da exigência operacional do *endpoint* `/deep/health` introduzida pelo mecanismo de *canary deployment* (CHOJNACKI, 2024). Cada mudança computada respondeu, portanto, a uma necessidade técnica preexistente, identificada durante a própria implementação e evolução do *pipeline* de implantação proposto. Essa delimitação

foi classificada segundo o padrão *Conventional Commits* já adotado no versionamento semântico do projeto, atribuindo-se cada *commit* a uma das categorias historicamente reconhecidas na literatura de manutenção de *software*: manutenção corretiva, voltada à correção de defeitos; manutenção adaptativa, decorrente de mudanças no ambiente ou nos requisitos operacionais; e manutenção perfectiva, direcionada à melhoria de estrutura, desempenho ou manutenibilidade sem alteração de comportamento externo (SWANSON, 1976).

Das 14 implantações, apenas uma foi associada a um evento de incidente nas tabelas de observabilidade: o desligamento deliberado do contêiner `terraplanner-prod`, que produziu uma violação real do SLI esperado no *job* `smoke_stable_post_promote` e acionou automaticamente o *job* `rollback_stable_auto`, sem intervenção manual. O incidente foi induzido com o propósito de popular os painéis de CFR e MTTR com um registro não nulo para fins de demonstração nesta monografia; essa implantação específica não pode ser tratada como isenta de falha real: a verificação automática de saúde pós-promoção efetivamente reprovou o ambiente, ainda que a causa raiz tenha sido artificialmente provocada. Sob o critério de falha orgânica adotado na linha de base descrita em subseção 4.3.2, as demais 13 implantações foram concluídas sem qualquer falha real de funcionamento; desconsiderando o evento deliberadamente induzido, a CFR orgânica do período seria, portanto, de 0%. Os valores de CFR e MTTR apresentados nas Seções 4.8.3 e 4.8.5 decorrem integralmente desse evento controlado e são, por isso, interpretados como uma estimativa conservadora da evolução dessas métricas de estabilidade em relação à linha de base.

4.8.2 Deployment Frequency (Frequência de Implantação, DF)

Aplicando-se a definição operacional da Equação 3.1 sobre as implantações persistidas na janela piloto ($D = 14$ implantações e $t = 14$ dias), obtém-se DF $= 14/14 = 1{,}00$ implantação por dia, conforme exibido no painel de valor agregado da Figura 4.16. Segundo os *benchmarks* DORA 2024 reproduzidos na Tabela 2.1, essa evolução deslocaria a organização do nível *Baixo* (entre uma implantação por mês e uma por semestre) para o limite superior do nível *Alto* (entre uma implantação por dia e uma por semana), a um passo do limiar *Elite* de implantação sob demanda. A Figura 4.16 evidencia, ainda, a distribuição diária das implantações ao longo da janela, com uma concentração maior nos dias finais do período, quando o ritmo de mudanças no projeto de teste se intensificou.

Figura 4.16 – Evolução diária da DF no pipeline proposto (27 jun.–10 jul. 2026)

Fonte: Elaborado pelo autor (2026).

4.8.3 Taxa de Falha nas Mudanças (*Change Failure Rate*, CFR)

Aplicando-se a definição operacional da Equação 3.4 sobre o único evento de incidente descrito na subseção 4.8.1 ($N_{failed} = 1$ e $N_{total} = 14$), obtém-se CFR $= \left(\frac{1}{14}\right) \times 100\% \approx 7{,}14\%$, exibido como 7,1% no painel de valor agregado da Figura 4.17. Além desse resultado global, a série temporal apresentada na figura registra um pico de 17,5% na data do incidente, correspondente à proporção de implantações em produção que resultaram em falha naquele recorte temporal. Em comparação com os 45,45% apurados na linha de base, apresentada na subseção 4.3.2, o valor observado representa uma redução absoluta de $45{,}45\% - 7{,}14\% \approx 38{,}31$ pontos percentuais e, tomando a linha de base como referência, uma redução relativa de $\left(\frac{45{,}45\% - 7{,}14\%}{45{,}45\%}\right) \times 100\% \approx 84{,}29\%$ na taxa de falha de mudanças. Ainda que os *benchmarks* de DeBellis et al. (2024) reproduzidos na Tabela 2.2 não definam faixas contínuas para os níveis intermediários tratados na subseção 2.1.5, o valor observado situa-se abaixo da referência do nível *Médio* (10%) e próximo da referência do nível *Elite* (5%), o que já representaria uma evolução expressiva frente ao nível *Baixo* da linha de base. Como discutido na subseção 4.8.1, esse resultado decorre de uma falha deliberadamente induzida pelo autor para fins de demonstração, e não de uma falha orgânica do código implantado; desconsiderando esse evento, a CFR das demais 13 implantações da janela piloto seria de 0%, o que reforça, em vez de enfraquecer, a evidência de melhoria na estabilidade das entregas.

Figura 4.17 – Evolução da CFR no pipeline proposto (27 jun.–10 jul. 2026)

Fonte: Elaborado pelo autor (2026).

4.8.4 Lead Time for Changes (Tempo de Entrega para Mudanças, LTFC)

O LTFC do *pipeline* de implantação proposto é apurado conforme as Equações 3.2 e 3.3. Na janela piloto, a média resultante foi de 4,2 horas, conforme exibido no painel de valor agregado da Figura 4.18. Confrontado com a média de 12,58 dias (isto é, 301,82 horas) apurada na linha de base após a exclusão do *outlier* de março discutido na subseção 4.3.3, o *pipeline* de implantação proposto entrega mudanças $\left(\frac{301{,}82}{4{,}2}\right) \approx 71,9$ vezes mais rápido que a linha de base. Segundo os *benchmarks* da Tabela 2.1, essa evolução deslocaria a organização do nível *Médio* (entre 7 e 30 dias) diretamente para o nível *Elite* (menos de um dia), transpondo o nível *Alto* intermediário.

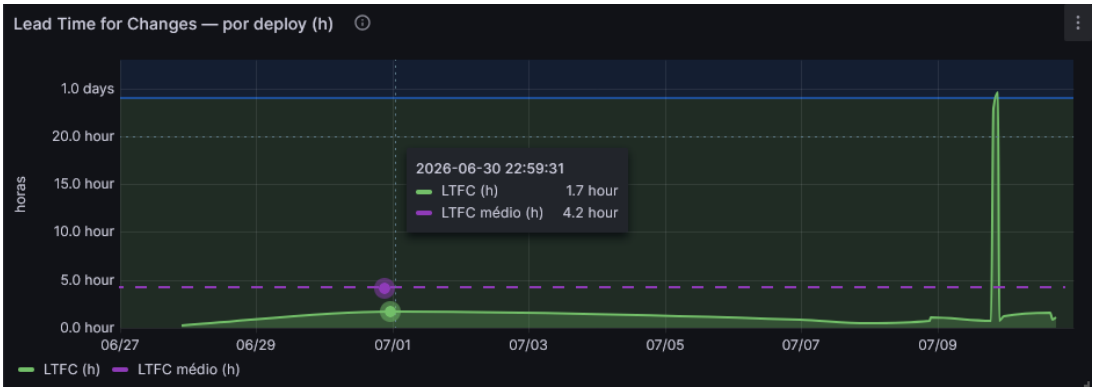

Figura 4.18 – LTFC por implantação no pipeline proposto (27 jun.–10 jul. 2026)

Fonte: Elaborado pelo autor (2026).

4.8.5 Tempo para Restaurar o Serviço (*Mean Time To Restore*, MTTR)

Para o único incidente efetivamente recuperado na janela piloto ($I = 1$), a Equação 3.5 resulta em MTTR $= \frac{T_1^{recovery} - T_1^{detect}}{1} \approx 3,4$ minutos ≈ 204 segundos, conforme o painel de valor agregado da Figura 4.19. Esse valor contrasta com o MTTR médio de 261,29 horas (isto é, aproximadamente 10,89 dias, ou 940.644 segundos) apurado na linha de base, representando uma redução de $\left(\frac{940.644-204}{940.644}\right) \times 100\% \approx 99,98\%$ no tempo de recuperação do serviço. Segundo

os *benchmarks* da Tabela 2.2, o resultado deslocaria a organização do nível *Baixo* (entre uma semana e um mês) para o nível *Elite* (menos de uma hora). Cabe ressaltar que o valor mede o intervalo entre a detecção da falha pelo *job* `smoke_stable_post_promote` e a conclusão da restauração pelo *job* `rollback_stable_auto`, acionado automaticamente pela diretiva `when: on_failure`, sem qualquer intervenção manual, diferentemente do ciclo de diagnóstico humano refletido na linha de base, em que a recuperação dependia da identificação manual do problema e da coordenação entre pessoas responsáveis antes da correção.

Figura 4.19 – MTTR por incidente no pipeline proposto (27 jun.–10 jul. 2026)

Fonte: Elaborado pelo autor (2026).

Em conjunto, as quatro métricas indicam evolução consistente nas duas dimensões de desempenho definidas por DeBellis et al. (2024): a velocidade da entrega, expressa pela DF e pelo LTFC, avançou do nível *Baixo*/*Médio* para os níveis *Alto*/*Elite*; a estabilidade, expressa pela CFR e pelo MTTR, avançou do nível *Baixo* para valores próximos ou compatíveis com o nível *Elite*. Cabe destacar que os níveis *Alto* e *Elite* aos quais o *pipeline* de implantação proposto se equipara não são categorias arbitrárias: DeBellis et al. (2024) os deriva de um levantamento global com mais de 39 mil profissionais de engenharia, o que confere às faixas de referência das Tabelas 2.1 e 2.2 respaldo estatístico em organizações reais de grande escala, e não apenas um referencial teórico. Assim, ao aproximar-se ou atingir esses patamares, o ambiente de pequena escala analisado neste trabalho evidencia desempenho comparável ao de organizações que compõem o estrato superior desse levantamento. Ressalvadas as limitações já discutidas, como a janela piloto de 14 dias, muito inferior aos 12 meses da linha de base, e execução no projeto de teste dedicado em vez do ambiente real de produção do TerraPlanner, esse conjunto de evidências sustenta quantitativamente a resposta à pergunta de pesquisa deste trabalho: a adoção de *Canary Deployment* com *rollback* automatizado esteve associada a uma melhoria mensurável no desempenho da entrega de *software*, tanto em velocidade quanto, sobretudo, em estabilidade, no ambiente de pequena escala analisado. A Figura 4.20 sintetiza essa evolução conjunta, posicionando a linha de base e o *pipeline* de implantação proposto sobre os níveis ordinais de desempenho DORA nos quatro eixos.

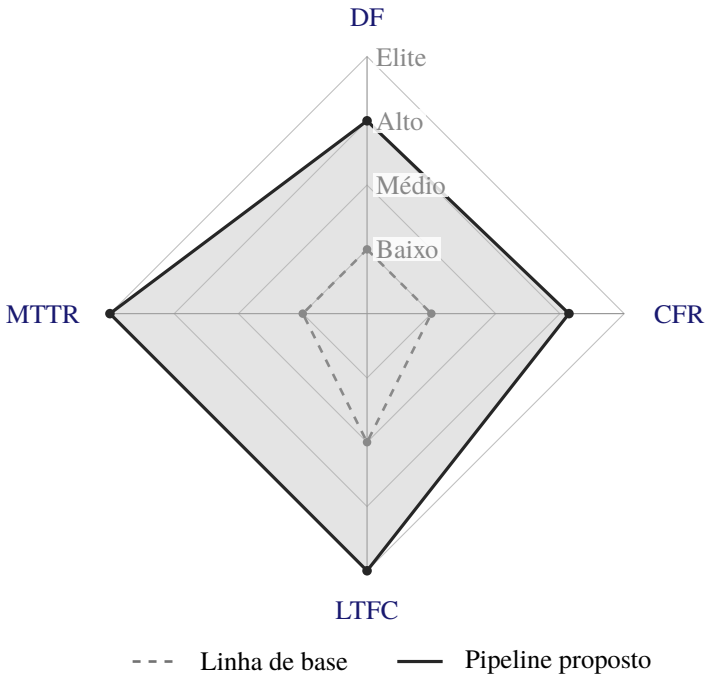

Figura 4.20 – Síntese da evolução das quatro métricas DORA: linha de base versus *pipeline* de implantação proposto

Fonte: Elaborado pelo autor (2026).

5 Considerações Finais

Este trabalho investigou o problema da mitigação de riscos operacionais em ambientes de pequena escala de desenvolvimento de software, nos quais restrições orçamentárias inviabilizam a duplicação de infraestrutura característica de estratégias como *Blue-Green Deployment*, ao mesmo tempo em que a ausência de automação no processo de reversão expõe o sistema a períodos de indisponibilidade. A pergunta central que orientou a pesquisa foi: *qual é o impacto quantitativo da implementação de Canary Deployment com rollback automatizado no desempenho da entrega de software, medido pelas métricas DORA, em um ambiente de pequena escala?*

Para responder a essa questão, estabeleceram-se quatro objetivos específicos: (i) estabelecer uma linha de base de desempenho por meio do mapeamento do *pipeline* de implantação vigente; (ii) desenvolver uma arquitetura experimental de implantação integrando *Canary Deployment* com observabilidade e *rollback* automatizado; (iii) validar a eficácia dos mecanismos de recuperação por meio de experimentos controlados; e (iv) comparar quantitativamente os cenários antes e após a automação.

5.1 Objetivos Atendidos

Conforme estabelecido na seção 1.2, este trabalho teve como objetivo geral analisar o impacto da adoção de *Canary Deployment* com *rollback* automatizado sobre a eficiência do ciclo de entrega de *software*, utilizando as métricas DORA como indicadores de velocidade e estabilidade. A execução integral da pesquisa abrangeu o diagnóstico do *pipeline* de implantação vigente, a concepção e a implementação da arquitetura proposta e a validação experimental dos mecanismos de recuperação, permitindo avaliar o atendimento aos quatro objetivos específicos nos seguintes termos:

- *Objetivo 1 (Estabelecer uma linha de base de desempenho): atingido*. O mapeamento sistemático do *pipeline* de implantação vigente, apresentado na seção 4.1, caracterizou seus componentes, fluxos operacionais e limitações, evidenciando gargalos como a adoção de *Big Bang Deployment*. Essa caracterização qualitativa foi complementada pela extração das métricas DORA referentes ao período de janeiro a dezembro de 2024 discutida na seção 4.3, que consolidou a linha de base quantitativa da organização: DF de aproximadamente 0,92 implantações por mês, CFR de 45,45%, LTFC mediano de 7,34 dias e MTTR médio de 261,29 horas, classificando a organização predominantemente no nível *Baixo* de desempenho segundo os arquétipos de DeBellis et al. (2024).
- *Objetivo 2 (Desenvolver uma arquitetura experimental de implantação): atingido*. A arquitetura lógica do *pipeline* de implantação com *Canary Deployment*, *rollback* automatizado

e observabilidade das métricas DORA, formalizada na seção 4.4, foi efetivamente implementada, conforme descrito na seção 4.5. A implementação materializou os treze estágios do *pipeline* de implantação proposto, as estratégias de roteamento canário por *allowlist* e por percentuais progressivos, os dois cenários de *rollback* automatizado e a camada de observabilidade baseada em PostgreSQL e Grafana.

- *Objetivo 3 (Validar a eficácia dos mecanismos de recuperação): atingido*. A validação foi conduzida por meio de experimentos controlados de injeção de falhas apresentados na seção 4.6 e seção 4.7, contemplando três categorias de falha (sendo elas: taxa de erro, latência e disponibilidade) e um cenário de controle sem falha injetada. Em todos os cenários com falha, o *rollback* automático restaurou o serviço em menos de dois minutos, com tempos de recuperação entre 1 minuto e 14 segundos e 1 minuto e 17 segundos; o *rollback* manual pós-promoção restaurou o ambiente estável em 45 segundos. Em conjunto, os cinco cenários confirmaram o funcionamento dos mecanismos de detecção e reversão propostos, sem falsos positivos na promoção de versões íntegras.

- *Objetivo 4 (Comparar quantitativamente os cenários): atingido*. A comparação entre a linha de base e o *pipeline* de implantação proposto, consolidada na seção 4.8 a partir de uma janela piloto de 14 dias em produção, evidencia evolução consistente nas quatro métricas DORA: o MTTR caiu de 261,29 horas na linha de base para 3,4 minutos, uma redução de aproximadamente 99,98%, e a CFR recuou de 45,45% para 7,1%, deslocando a estabilidade da entrega do nível *Baixo* para valores próximos ou compatíveis com o nível *Elite*; o LTFC passou de 12,58 dias para 4,2 horas, cerca de 71,9 vezes mais rápido, e a DF avançou para 1,00 implantação por dia, deslocando a velocidade da entrega dos níveis *Baixo*/*Médio* para os níveis *Alto*/*Elite*. Esse conjunto de evidências situa o ganho observado na recuperação de falhas no panorama mais amplo dos indicadores de velocidade e estabilidade.

Assim, os resultados apresentados respondem de forma articulada à pergunta de pesquisa que orientou este trabalho. O diagnóstico do *pipeline* de implantação vigente evidenciou os gargalos que fundamentaram a proposta; a arquitetura implementada materializou os mecanismos de *Canary Deployment*, *rollback* automatizado e observabilidade DORA; e a validação experimental confirmou a eficácia desses mecanismos sob condições controladas de falha. Em conjunto, essas evidências permitem avaliar, de forma fundamentada, o impacto da automação sobre a velocidade e, sobretudo, a estabilidade da entrega de *software* em um ambiente de pequena escala.

5.2 Implicações Práticas

A solução implementada e validada neste trabalho fornece evidências quantitativas sobre a viabilidade de práticas de alta confiabilidade em ambientes de recursos limitados. A instrumentação das métricas DORA, detalhada na subseção 4.5.6, permite mensurar objetivamente o

impacto da automação do *rollback* sobre a velocidade e a estabilidade da entrega, oferecendo um modelo de referência replicável para organizações similares. Essa abordagem tem potencial para democratizar práticas de SRE, ao evidenciar que PMEs e *startups* podem operar com níveis de disponibilidade competitivos por meio de automação inteligente, sem depender de orçamentos de infraestrutura característicos de grandes corporações tecnológicas. Adicionalmente, a redução observada no tempo de recuperação contribui para superar o dilema entre velocidade e garantia identificado por Hilton et al. (2017), ao indicar que a automação robusta do ciclo de *feedback* pode manter a estabilidade sem sacrificar a agilidade da entrega, corroborando as descobertas de Forsgren, Humble e Kim (2018).

No âmbito operacional, a estratégia de promoção por *digest* imutável, detalhada na subseção 4.5.3, elimina a divergência entre o artefato validado em homologação e o artefato efetivamente implantado em produção, mitigando um risco recorrente em organizações que reconstroem a imagem em cada ambiente. Da mesma forma, o desenho incremental e não bloqueante dos *quality gates*, descrito na subseção 4.5.2, configura-se como implicação prática relevante para laboratórios e equipes com alta rotatividade de integrantes, como o TerraLAB: a possibilidade de habilitar um controle sem torná-lo imediatamente bloqueante permite introduzir práticas avançadas de DevOps sem impor latência excessiva à adaptação de novos desenvolvedores. Por fim, a redução do trabalho manual repetitivo, evidenciada pela automação completa da detecção e da reversão nos cenários testados, sugere potencial de contribuição para a sustentabilidade cognitiva e econômica de equipes enxutas, mitigando o estresse operacional e permitindo que desenvolvedores concentrem-se em atividades de engenharia de maior valor agregado (BEYER et al., 2016).

5.3 Limitações

Este trabalho apresenta limitações que devem ser consideradas na interpretação dos resultados. Os experimentos de injeção de falhas foram conduzidos em um projeto de teste dedicado, isolado do ambiente real de produção do TerraPlanner, o que limita a extrapolação direta dos resultados para condições reais de tráfego e concorrência. A coexistência dos contêineres *canary*, *stable* e *staging* na mesma VM de aplicação introduz, adicionalmente, o efeito de vizinho ruidoso discutido na subseção 4.6.3, que pode influenciar marginalmente as medições de latência e disponibilidade coletadas durante os experimentos. As três categorias de falha injetadas representam, por sua vez, um subconjunto controlado e deliberadamente simples dos modos de falha possíveis em produção; degradações mais sutis ou compostas, como vazamento progressivo de memória, esgotamento de conexões com dependências externas ou falhas intermitentes, não foram avaliadas.

Outras limitações decorrem da maturidade ainda parcial de determinados componentes do *pipeline* de implantação implementado. A cobertura de testes automatizados permanece reduzida

mesmo após a evolução proposta, uma vez que os *quality gates* de teste e de cobertura foram mantidos predominantemente não bloqueantes, como discutido em subseção 4.5.2. Adicionalmente, a comparação quantitativa relativa à DF, ao LTFC e à CFR do *pipeline* de implantação proposto na seção 4.8, fundamenta-se em uma janela piloto de observação de 14 dias, duração muito inferior aos doze meses considerados na linha de base, o que limita, para essas três métricas, a robustez estatística da comparação direta entre os cenários. Soma-se a isso o caráter predominantemente sintético das alterações introduzidas no projeto de testes, ainda que elas representem melhorias efetivas de usabilidade e de engenharia no *front-end* do TerraPlanner.

5.4 Trabalhos Futuros

Os trabalhos futuros compreendem, em primeiro lugar, a ampliação do protocolo de injeção de falhas para modos de degradação mais sutis ou compostos, como vazamento progressivo de memória, esgotamento de conexões com dependências externas e falhas intermitentes, não contemplados no escopo experimental atual. Em segundo lugar, planeja-se estender a avaliação experimental ao ambiente real de produção após o *go-live* do TerraPlanner, quando a presença de tráfego real permitirá validar a estratégia de roteamento progressivo por percentuais sob condições de carga não sintéticas. Em terceiro lugar, a ampliação da janela de observação piloto utilizada na comparação da DF, do LTFC e da CFR permitirá consolidar, com maior robustez estatística, o impacto da automação sobre essas métricas ao longo do tempo. Adicionalmente, planeja-se avaliar a escalabilidade da solução e sua aplicabilidade a outros componentes do sistema, como o *back-end* do TerraPlanner, ampliando o escopo de validação e fortalecendo a generalização dos achados.

Como direção de evolução de mais longo prazo, sugere-se investigar a incorporação de práticas de MLOps (*Machine Learning Operations*, MLOps) ao *pipeline* de implantação implementado. Conforme discutido por Subramanya, Sierla e Vyatkin (2022), o MLOps estende os princípios do DevOps a modelos de aprendizado de máquina por meio de monitoramento contínuo, detecção de *drift* de dados e reciclagem automatizada de modelos a partir de novas evidências operacionais. A base analítica já implementada para as métricas DORA, que registra o histórico estruturado de implantações e incidentes nas tabelas `dora_deployments` e `dora_incidents`, constitui uma fonte de dados adequada ao treinamento de um agente preditivo de risco: um modelo continuamente realimentado pelo próprio histórico do *pipeline* de implantação, capaz de estimar, antes da promoção de uma mudança, a probabilidade de que ela produza um incidente, a partir de padrões observados em implantações e *rollbacks* anteriores. Essa camada preditiva poderia complementar os limiares de SLIs atualmente utilizados nas fases de *rollout*, deslocando parte da decisão de promoção de um critério reativo, fundamentado na degradação já observada, para um critério antecipatório, fundamentado no risco histórico da mudança, o que representa uma direção promissora para aprofundar a automação e a inteligência do processo de entrega em ambientes de pequena escala.

Declaração de Uso de Inteligência Artificial Generativa

Durante a preparação deste documento, eu, Josué Vila Real de Almeida, estudante de graduação em Ciência da Computação da Universidade Federal de Ouro Preto, declaro o uso da IAGen ChatGPT (OpenAI, versão do modelo GPT-5) para revisão textual e geração de códigos LaTeX.

Após o uso deste modelo, eu revisei e editei o conteúdo em conformidade com os princípios éticos para uso de IAGen (Resolução CONPEP nº 144) e com os acordos estabelecidos com o orientador desta pesquisa. Dessa forma, assumo total responsabilidade pelo conteúdo da publicação.

Referências

ABBASS, M. K. A.; OSMAN, R. I. E.; MOHAMMED, A. M. H.; ALSHAIKH, M. W. A. Adopting continuous integeration and continuous delivery for small teams. In: *2019 International Conference on Computer, Control, Electrical, and Electronics Engineering (ICCCEEE)*. IEEE, 2019. Disponível em: <http://dx.doi.org/10.1109/ICCCEEE46830.2019.9070849>.

ALSOLAI, H.; ROPER, M. A systematic literature review of machine learning techniques for software maintainability prediction. *Information and Software Technology*, Elsevier BV, v. 119, p. 106214, mar. 2019. ISSN 0950-5849. Disponível em: <http://dx.doi.org/10.1016/j.infsof.2019.106214>.

Atlassian. *DevOps*. 2025. Online. Acesso em: 27 dez. 2025. Disponível em: <https://www.atlassian.com/devops>.

BASIRI, A.; HOCHSTEIN, L.; JONES, N.; TUCKER, H. Automating chaos experiments in production. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019. p. 31–40. Disponível em: <https://doi.org/10.1109/ICSE-SEIP.2019.00012>.

BEYER, B.; JONES, C.; PETOFF, J.; MURPHY, N. R. *Site Reliability Engineering: How Google Runs Production Systems*. Sebastopol, CA: "O'Reilly Media, Inc.", 2016. ISBN 9781491929124.

BEYER, B.; MURPHY, N. R.; RENSIN, D.; KAWAHARA, K.; THORNE, S. (Ed.). *The Site Reliability Workbook: Practical Ways to Implement SRE*. O'Reilly Media, 2018. Acessado em: 2025-12-30. ISBN 978-1-4920-2906-6. Disponível em: <https://sre.google/workbook/table-of-contents/>.

BUZACHIS, A.; GALLETTA, A.; CELESTI, A.; CARNEVALE, L.; VILLARI, M. Towards osmotic computing: a blue-green strategy for the fast re-deployment of microservices. In: *2019 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2019. p. 1–6. Disponível em: <http://dx.doi.org/10.1109/ISCC47284.2019.8969621>.

CHEN, L. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, Institute of Electrical and Electronics Engineers (IEEE), v. 32, n. 2, p. 50–54, mar. 2015. ISSN 1937-4194. Disponível em: <http://dx.doi.org/10.1109/MS.2015.27>.

CHOJNACKI, T. *Mastering npm scripts: Best practices in sustainable naming and organizing of your scripts*. 2024. Blog. Disponível em: <https://literat.dev/blog/2024-12-14/mastering-npm-scripts-best-practices-in-sustainable-naming-and-organizing-of-your-scripts/>. Acesso em: 4 jul. 2026.

COHN, M. *Succeeding with Agile: Software Development Using Scrum*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2009. ISBN 0321579364.

CrowdStrike. *External Technical Root Cause Analysis — Channel File 291*. [S.l.], 2024. Disponível em: <https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf>. Acesso em: 2026-03-03.

DEBELLIS, D.; STORER, K.; HARVEY, N.; BEANE, M.; EDWARDS, R.; FRASER, E.; GOOD, B.; KALLIAMVAKOU, E.; KIM, G.; MAXWELL, E.; D'ANGELO, S.; INMAN, S.; MURILLO, A.; VILLALBA, D. *DORA 2025 State of AI-assisted Software Development Report*. [S.l.], 2025. Disponível em: <https://cloud.google.com/resources/content/2025-dora-ai-assisted-software-development-report?e=48754805>.

DEBELLIS, D.; STORER, K.; LEWIS, A.; GOOD, B.; VILLALBA, D.; MAXWELL, E.; CASTILLO, K.; IRVINE, M.; HARVEY, N. *DORA Accelerate State of DevOps 2024 Report*. Estados Unidos, 2024. Disponível em: <https://cloud.google.com/devops/state-of-devops>.

DEBROY, V.; MILLER, S. Overcoming challenges with continuous integration and deployment pipelines: An experience report from a small company. *IEEE Software*, Institute of Electrical and Electronics Engineers (IEEE), v. 37, n. 3, p. 21–29, maio 2020. ISSN 1937-4194. Disponível em: <http://dx.doi.org/10.1109/MS.2019.2947004>.

DevOps Research and Assessment (DORA). *DORA's Software Delivery Metrics: The Four Keys*. 2025. Online. Acesso em: 10 dez. 2025. Disponível em: <https://dora.dev/guides/dora-metrics-four-keys/>.

DOWNS, J. *Noisy Neighbor antipattern*. 2025. Azure Architecture Center, Microsoft Learn. Disponível em: <https://learn.microsoft.com/en-us/azure/architecture/antipatterns/noisy-neighbor/noisy-neighbor>. Acesso em: 2026-07-07.

DUVALL, P. M.; MATYAS, S.; GLOVER, A. *Continuous Integration: Improving Software Quality and Reducing Risk*. Upper Saddle River, NJ: Pearson Education, 2007. ISBN 978-0321630148.

FORSGREN, N.; HUMBLE, J.; KIM, G. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. Portland, OR: IT Revolution Press, 2018. ISBN 9781942788331.

FOWLER, M. *Strangler Fig*. 2004. <https://martinfowler.com/bliki/StranglerFigApplication.html>. Acessado em: 4 de março de 2026. Disponível em: <https://martinfowler.com/bliki/StranglerFigApplication.html>. Acesso em: 4 mar. 2026.

FOWLER, M. *Test Coverage*. 2012. Disponível em: <https://martinfowler.com/bliki/TestCoverage.html>. Acesso em: 4 jul. 2026.

FOWLER, M. *Patterns for Managing Source Code Branches*. 2020. Online. Disponível em: <https://martinfowler.com/articles/branching-patterns.html>.

FOWLER, M.; FOEMMEL, M. *Continuous Integration*. 2006. Online. Disponível em: <https://martinfowler.com/articles/continuousIntegration.html>.

GHALEB, T.; ABDULJALIL, O.; HASSAN, S. Ci/cd configuration practices in open-source android apps: An empirical study. *ACM Transactions on Software Engineering and Methodology*, Association for Computing Machinery (ACM), maio 2025. ISSN 1557-7392. Disponível em: <http://dx.doi.org/10.1145/3736758>.

GIL, A. C. *Como elaborar projetos de pesquisa*. 4. ed. São Paulo: Atlas, 2002. ISBN 85-224-3169-8.

GOODHART, C. A. E. Problems of monetary management: The uk experience. In: ____. *Monetary Theory and Practice: The UK Experience*. London: Macmillan Education UK, 1984. p. 91–121. ISBN 978-1-349-17295-5. Disponível em: <https://doi.org/10.1007/978-1-349-17295-5_4>.

GOYAL, A. Optimising cloud-based ci/cd pipelines: Techniques for rapid software deployment. *Technix International Journal for Engineering Research*, v. 11, p. a896–a904, 11 2024.

HEVNER, A. R.; MARCH, S. T.; PARK, J.; RAM, S. Design science in information systems research1. *MIS Quarterly*, MIS Quarterly, v. 28, n. 1, p. 75–106, mar. 2004. ISSN 2162-9730. Disponível em: <http://dx.doi.org/10.2307/25148625>.

HILTON, M.; NELSON, N.; TUNNELL, T.; MARINOV, D.; DIG, D. Trade-offs in continuous integration: assurance, security, and flexibility. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017. (ESEC/FSE'17), p. 197–207. Disponível em: <http://dx.doi.org/10.1145/3106237.3106270>.

HOEK, A. van der; HALL, R. S.; HEIMBIGNER, D.; WOLF, A. L. Software release management. *ACM SIGSOFT Software Engineering Notes*, Association for Computing Machinery (ACM), v. 22, n. 6, p. 159–175, nov. 1997. ISSN 0163-5948. Disponível em: <http://dx.doi.org/10.1145/267896.267909>.

HUMBLE, J.; FARLEY, D. *Continuous delivery reliable software releases through build, test, and deployment automation*. Boston: Pearson Education, 2010. ISBN 9780321601919.

IVANKOVIĆ, M.; PETROVIĆ, G.; JUST, R.; FRASER, G. Code coverage at Google. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 2019. (ESEC/FSE '19), p. 955–963. Disponível em: <https://doi.org/10.1145/3338906.3340459>.

JHA, A. V.; TERI, R.; VERMA, S.; TARAFDER, S.; BHOWMIK, W.; MISHRA, S. K.; APPASANI, B.; SRINIVASULU, A.; PHILIBERT, N. From theory to practice: Understanding devops culture and mindset. *Cogent Engineering*, Informa UK Limited, v. 10, n. 1, set. 2023. ISSN 2331-1916. Disponível em: <http://dx.doi.org/10.1080/23311916.2023.2251758>.

KHAN, M. S.; KHAN, A. W.; KHAN, F.; KHAN, M. A.; WHANGBO, T. K. Critical challenges to adopt devops culture in software organizations: A systematic review. *IEEE Access*, Institute of Electrical and Electronics Engineers (IEEE), v. 10, p. 14339–14349, 2022. ISSN 2169-3536. Disponível em: <http://dx.doi.org/10.1109/ACCESS.2022.3145970>.

KIM, G. *The Three Ways: The Principles Underpinning DevOps*. 2012. Online. Acesso em: 17 dez. 2025. Disponível em: <https://itrevolution.com/articles/the-three-ways-principles-underpinning-devops/>.

KIM, G.; BEHR, K.; SPAFFORD, G. *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win*. 1st. ed. [S.l.]: IT Revolution Press, 2013. ISBN 0988262592.

KIM, G.; DEBOIS, P.; WILLIS, J.; HUMBLE, J. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. [S.l.]: IT Revolution Press, 2016. ISBN 1942788002.

KINSMAN, T.; WESSEL, M.; GEROSA, M. A.; TREUDE, C. How do software developers use github actions to automate their workflows? In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021. p. 420–431. Disponível em: <http://dx.doi.org/10.1109/MSR52588.2021.00054>.

LEITE, L.; ROCHA, C.; KON, F.; MILOJICIC, D.; MEIRELLES, P. A survey of devops concepts and challenges. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 52, n. 6, nov. 2019. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3359981>.

LISPECTOR, C. *A hora da estrela*. Rio de Janeiro: Rocco, 1998. 88 p. ISBN 978-85-325-0812-6.

MALHOTRA, A.; ELSAYED, A.; TORRES, R.; VENKATRAMAN, S. Evaluate canary deployment techniques using kubernetes, istio, and liquibase for cloud native enterprise applications to achieve zero downtime for continuous deployments. *IEEE Access*, Institute of Electrical and Electronics Engineers (IEEE), v. 12, p. 87883–87899, 2024. ISSN 2169-3536. Disponível em: <http://dx.doi.org/10.1109/ACCESS.2024.3416087>.

MARTIN, R. C. *The Clean Coder: A Code of Conduct for Professional Programmers*. Philadelphia, PA: Prentice Hall, 2011. ISBN 978-0-13-708107-3.

PRODANOV, C. C.; FREITAS, E. C. d. *Metodologia do Trabalho Científico: Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico*. 2. ed. Novo Hamburgo: Editora Feevale, 2013. ISBN 8577171582.

RAHMAN, M. T.; QUEREL, L.-P.; RIGBY, P. C.; ADAMS, B. Feature toggles: practitioner practices and a case study. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, 2016. (MSR '16), p. 201–211. ISBN 9781450341868. Disponível em: <https://doi.org/10.1145/2901739.2901745>.

RAMASWAMY, Y. Zero downtime deployments in devops: Blue-green, canary, and feature flag techniques. *International Journal of Communication Networks and Information Security*, Kohat University of Science and Technology (KUST), v. 16, n. 5, p. 969–977, 2024.

RANGNAU, T.; BUIJTENEN, R. v.; FRANSEN, F.; TURKMEN, F. Continuous security testing: A case study on integrating dynamic security testing tools in ci/cd pipelines. In: *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, 2020. p. 145–154. Disponível em: <http://dx.doi.org/10.1109/EDOC49727.2020.00026>.

RUDRABHATLA, C. K. Comparison of zero downtime based deployment techniques in public cloud infrastructure. In: *2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*. IEEE, 2020. p. 1082–1086. Disponível em: <http://dx.doi.org/10.1109/I-SMAC49090.2020.9243605>.

RüEGGER, J.; KROPP, M.; GRAF, S.; ANSLOW, C. Fully automated dora metrics measurement for continuous improvement. In: *Proceedings of the 2024 International Conference on Software and Systems Processes*. ACM, 2024. (ICSSP '24), p. 36–45. Disponível em: <http://dx.doi.org/10.1145/3666015.3666020>.

SALLIN, M.; KROPP, M.; ANSLOW, C.; QUILTY, J. W.; MEIER, A. Measuring software delivery performance using the four key metrics of devops. In: ____. *Agile Processes in Software Engineering and Extreme Programming*. Springer International Publishing, 2021. p. 103–119. ISBN 9783030780982. Disponível em: <http://dx.doi.org/10.1007/978-3-030-78098-2_7>.

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

SINGH, C.; GABA, N. S.; KAUR, M.; KAUR, B. Comparison of different ci/cd tools integrated with cloud platform. In: *2019 9th International Conference on Cloud Computing, Data Science amp; Engineering (Confluence)*. IEEE, 2019. Disponível em: <http://dx.doi.org/10.1109/confluence.2019.8776985>.

SOLANKI, S.; THASON, J. R. M. Automated canary deployments in continuous delivery: Balancing speed and reliability. *International Journal of Enhanced Research In Science Technology Engineering*, v. 14, p. 2319–7463, 04 2025.

SUBRAMANYA, R.; SIERLA, S.; VYATKIN, V. From devops to mlops: Overview and application to electricity market forecasting. *Applied Sciences*, MDPI AG, v. 12, n. 19, p. 9851, 2022. ISSN 2076-3417. Disponível em: <https://doi.org/10.3390/app12199851>.

SWANSON, E. B. The dimensions of maintenance. In: *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*. San Francisco, CA: IEEE Computer Society Press, 1976. p. 492–497.

TerraLAB. *O TerraLAB e as parcerias em inovação e extensão tecnológica*. 2020. Website. Disponível em: <https://www2.decom.ufop.br/terralab/o-terralab-e-as-parcerias-em-inovacao-e-extensao-tecnologica/>.

TerraLAB. *Explicando as áreas de atuação do TerraLAB*. 2025. Website. Disponível em: <https://www2.decom.ufop.br/terralab/explicando-as-areas-de-atuacao-do-terralab-2/>.

TUAPE, M.; HASHEELA-MUFETI, V.; KAYANDA, A.; KASURINEN, J. Software development in small software companies: Exploring the usage of procedures, techniques, methods and models in practice. In: *2021 2nd European Symposium on Software Engineering*. ACM, 2021. (ESSE 2021), p. 29–38. Disponível em: <http://dx.doi.org/10.1145/3501774.3501779>.

UGWUEZE, V. U.; CHUKWUNWEIKE, J. N. Continuous integration and deployment strategies for streamlined devops in software engineering and application delivery. *International Journal of Computer Applications Technology and Research*, Association of Technology and Science, v. 14, n. 1, p. 1–24, jan. 2025. ISSN 2319-8656. Disponível em: <http://dx.doi.org/10.7753/IJCATR1401.1001>.

Universidade Federal de Ouro Preto. *Ferramenta criada na UFOP é utilizada por importantes centros de pesquisa do país*. 2017. Notícia institucional. Disponível em: <https://ufop.br/noticias/pesquisa-e-inovacao/ferramenta-criada-na-ufop-e-utilizada-por-importantes-centros-de>.

VANGALA, V. Blue-green and canary deployments in devops: A comparative study. *International Journal of Engineering Sciences Research Technology*, v. 15, p. 1047–1063, 04 2020.

VOCKE, H. *The Practical Test Pyramid*. 2018. Online. Publicado em MartinFowler.com. Disponível em: <https://martinfowler.com/articles/practical-test-pyramid.html>.

WESTON, D. *Helping our customers through the CrowdStrike outage*. 2024. Official Microsoft Blog. Disponível em: <https://blogs.microsoft.com/blog/2024/07/20/helping-our-customers-through-the-crowdstrike-outage/>. Acesso em: 2026-03-03.

WILKES, B.; MILANI, A. M. P.; STOREY, M.-A. A framework for automating the measurement of devops research and assessment (dora) metrics. In: *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023. p. 62–72. Disponível em: <http://dx.doi.org/10.1109/ICSME58846.2023.00018>.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. Empirical research. In: *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 9–26. ISBN 978-3-662-69305-6.

YASAR, H.; KONTOSTATHIS, K. Where to integrate security practices on devops platform. *International Journal of Secure Software Engineering*, IGI Global, v. 7, n. 4, p. 39–50, out. 2016. ISSN 1947-3044. Disponível em: <http://dx.doi.org/10.4018/IJSSE.2016100103>.

ZOLKIFLI, N. N.; NGAH, A.; DERAMAN, A. Version control system: A review. *Procedia Computer Science*, Elsevier BV, v. 135, p. 408–415, 2018. ISSN 1877-0509. Disponível em: <http://dx.doi.org/10.1016/j.procs.2018.08.191>.