Uma Abordagem DevSecOps para Inserção e Automação de Práticas de Segurança em Pipelines CI/CD

Guilherme Henrique de Lima Machado

[1]Instituto de Ciências Exatas e Informática – Pontíficia Universidade Católica de Minas Gerais
Minas Gerais (PUC Minas)
Contagem – MG – Brasil

guilhermeoh.machado@gmail.com

Resumo. *A integração da segurança no ciclo de vida de desenvolvimento de software, ou DevSecOps, tornou-se essencial para aumentar a confiabilidade e reduzir riscos nas entregas contínuas. O objetivo deste trabalho é investigar e implementar uma abordagem prática para a inserção e automação de práticas de segurança em pipelines CI/CD. Foi realizada uma pesquisa experimental na Google Cloud Platform (GCP) com um pipeline de 15 etapas, integrando Semgrep, Trivy, Checkov, OWASP ZAP e um script Python customizado para testes de Força Bruta contra a aplicação Damn Vulnerable Web Application (DVWA). O trabalho conclui que o pipeline alcançou a detecção de 13 das 17 vulnerabilidades mapeadas, atingindo uma cobertura de 76,5%. Os resultados demonstram a eficácia da orquestração automatizada na identificação de falhas técnicas, enquanto identificam lacunas que exigem intervenção humana para análise de lógica de negócio.*

Abstract. *Integrating security into the software development lifecycle, or DevSecOps, has become essential to increase reliability and reduce risks in continuous delivery. This work aims to investigate and implement a practical approach for inserting and automating security practices in CI/CD pipelines. Experimental research was conducted on the Google Cloud Platform (GCP) with a 15-step pipeline, integrating Semgrep, Trivy, Checkov, OWASP ZAP, and a customized Python script for Brute Force testing against the Damn Vulnerable Web Application (DVWA). The study concludes that the pipeline detected 13 out of 17 mapped vulnerabilities, reaching 76.5% coverage. The results demonstrate the effectiveness of automated orchestration in identifying technical flaws while identifying gaps that require human intervention for business logic analysis.*

1. Introdução

A indústria de *software* adotou amplamente metodologias ágeis e a cultura DevOps para acelerar a entrega de valor ao cliente. No centro dessa transformação está o *pipeline* de **Integração Contínua e Entrega Contínua (CI/CD)**, um processo automatizado que gerencia as etapas de *build*, teste e implantação. Essa automação permite que organizações liberem novas versões de *software* em um ritmo sem precedentes, respondendo rapidamente às demandas do mercado.

Contudo, essa velocidade expôs uma falha crítica nos processos legados: as práticas de segurança tradicionais não conseguem acompanhar o rápido Ciclo de Vida

de Desenvolvimento de *Software* (SDLC, do inglês *Software Development Life Cycle*) [Rangnau et al. 2020]. Historicamente, a segurança era tratada como uma fase final, manual e executada por equipes isoladas [Ahmed and Francis 2019]. Isso funciona como um gargalo que contradiz a agilidade do DevOps, fazendo com que a segurança seja percebida como um obstáculo que desacelera os processos [Santos et al. 2024]. Como resultado, muitas equipes negligenciam a segurança ou a aplicam tardiamente, o que gera vulnerabilidades, retrabalho e aumento do risco em produção.

Para resolver esse conflito, surgiu a cultura **DevSecOps** (Desenvolvimento, Segurança e Operações). A sua definição consiste em uma abordagem que integra controles e processos de segurança ao ciclo de vida DevOps, com colaboração entre as equipes de segurança, desenvolvimento e operações [Putra and Kabetta 2022]. O DevSecOps torna a equipe inteiramente responsável pela segurança da aplicação, implementando decisões de segurança na mesma velocidade que as tarefas de desenvolvimento [Ahmed and Francis 2019, Kushwaha et al. 2024]. A principal estratégia adotada é a de **"shift-left"**, que consiste em introduzir práticas e ferramentas de segurança o mais cedo possível no processo [Chen and Suo 2022], de forma automatizada dentro do próprio *pipeline* CI/CD.

Apesar do consenso sobre a importância do *DevSecOps*, o problema de *como* incorporar efetivamente as práticas de segurança de forma contínua e automatizada persiste. A pergunta que este trabalho busca responder é: **Como incorporar práticas de segurança de forma contínua e automatizada em *pipelines* CI/CD?**

A literatura recente apresenta estudos de caso práticos, muitos focando na integração de ferramentas específicas como SAST (Static Application Security Testing) e DAST (Dynamic Application Security Testing) [Putra and Kabetta 2022, Marandi et al. 2023, Kushwaha et al. 2024, Sun et al. 2021], ou aprofundando-se nos desafios técnicos de uma única técnica, como o DAST [Rangnau et al. 2020]. No entanto, ainda há uma lacuna em abordagens práticas que demonstrem a orquestração de um *conjunto* mais amplo de ferramentas (incluindo SAST, DAST, SCA, IaC Scan, *Container Scan* e testes de autenticação) de forma unificada e reprodutível em ambientes de nuvem modernos [Bernardino et al. 2024].

Este trabalho busca preencher essa lacuna, tendo como objetivo geral investigar uma abordagem prática para a inserção e automação de práticas de segurança em *pipelines* CI/CD. Os objetivos específicos deste trabalho são:

- Implementar um *pipeline* CI/CD orquestrado em nuvem integrando seis camadas distintas de análise de segurança: SAST, SCA, IaC Scan, *Container Scan*, DAST e testes de Força Bruta.
- Desenvolver um mecanismo de análise automatizada para mensurar a cobertura de detecção de vulnerabilidades conhecidas em um ambiente controlado.
- Avaliar a eficácia das ferramentas de código aberto na identificação de falhas críticas, distinguindo vulnerabilidades passíveis de automação daquelas que exigem intervenção humana especializada.
- Documentar o processo de provisionamento de infraestrutura como código (IaC) e orquestração de *pipeline* para garantir a reprodutibilidade do experimento.

Para isso, adota-se uma pesquisa experimental estruturada da seguinte forma:

- A Seção 2 apresenta a fundamentação teórica sobre os conceitos de DevSecOps, Teste Contínuo de Segurança e *pipelines* CI/CD.
- A Seção 3 discute os trabalhos relacionados que fundamentam esta pesquisa.
- A Seção 4 detalha os materiais e métodos empregados na pesquisa experimental, incluindo o ferramental e a aplicação-alvo.
- A Seção 5 apresenta e discute os resultados obtidos, com foco na cobertura de detecção alcançada.
- Finalmente, a Seção 6 apresenta as conclusões e sugestões de trabalhos futuros.

2. Fundamentação Teórica

Esta seção aborda os pilares conceituais e os padrões técnicos que sustentam esta pesquisa: a cultura *DevSecOps*, as categorias de teste de segurança e a orquestração via *pipelines* de Integração Contínua e Entrega Contínua.

2.1. A Cultura DevSecOps e o Princípio de Shift-Left

O conceito central deste trabalho é o **DevSecOps**, uma evolução cultural do *DevOps* que integra formalmente a segurança ao ciclo de vida de desenvolvimento de *software*. Sua definição transcende a simples adoção de ferramentas, consistindo em uma mentalidade que torna a equipe inteiramente responsável pela segurança da aplicação [Ahmed and Francis 2019, Chen and Suo 2022]. Isso é alcançado ao implementar atividades e decisões de segurança na mesma medida e velocidade que as tarefas de desenvolvimento e operações, quebrando os silos tradicionais entre as equipes [Santos et al. 2024, Efendi et al. 2021].

A estratégia fundamental para essa integração é o *shift-left*, que consiste em introduzir práticas de segurança o mais cedo possível no processo de desenvolvimento [Kushwaha et al. 2024]. Ao mover a detecção de falhas para as fases iniciais, como o *coding* (codificação) e *build* (construção), as organizações reduzem custos de correção e o risco de vulnerabilidades críticas atingirem o ambiente de produção [Chen and Suo 2022].

2.2. Teste Contínuo de Segurança e Taxonomia de Vulnerabilidades

Para operacionalizar o *DevSecOps*, adota-se o Teste Contínuo de Segurança (*Continuous Security Testing*). Este princípio define a aplicação de testes automatizados de forma ininterrupta ao longo de todo o *pipeline* [Santos et al. 2024, Bernardino et al. 2024]. Para que a análise seja abrangente, utilizam-se diferentes categorias de ferramentas:

- **SAST (*Static Application Security Testing*):** Análise estática que examina o código-fonte sem executá-lo (*white-box*), buscando padrões de código inseguros [Masood and Java 2015].
- **DAST (*Dynamic Application Security Testing*):** Teste dinâmico que interage com a aplicação em tempo de execução (*black-box*), simulando ataques externos para identificar falhas de exposição e runtime [Rangnau et al. 2020].
- **SCA (*Software Composition Analysis*):** Análise de composição de *software* focada em identificar vulnerabilidades conhecidas em bibliotecas e dependências de terceiros.
- **IaC Scan (*Infrastructure as Code Scanning*):** Varredura de arquivos de Infraestrutura como Código para detectar configurações inseguras no provisionamento de recursos de nuvem e orquestradores.

- ***Container Scan*:** Varredura de imagens de contêiner (*Docker*, OCI) que analisa o sistema operacional base e os pacotes instalados em busca de Vulnerabilidades e Exposições Comuns (*Common Vulnerabilities and Exposures* - CVEs) conhecidos. Também identifica imagens baseadas em sistemas operacionais em fim de suporte (*End of Support Life* - EOSL), sinalizando riscos de componentes desatualizados.

A classificação dos achados nestes testes baseia-se em padrões globais de segurança. A Enumeração de Fraquezas Comuns (*Common Weakness Enumeration* - CWE) é uma lista de tipos de fraquezas de *software* que serve como linguagem comum para descrever falhas de arquitetura e código. Quando uma instância específica de vulnerabilidade é identificada em um produto, ela é catalogada como uma Vulnerabilidade e Exposição Comum (*Common Vulnerabilities and Exposures* - CVE), permitindo o rastreio em bases de dados como o Banco de Dados Nacional de Vulnerabilidades (*National Vulnerability Database* - NVD). A severidade dessas falhas é quantificada pelo Sistema de Pontuação de Vulnerabilidade Comum (*Common Vulnerability Scoring System* - CVSS), que fornece uma pontuação numérica de 0 a 10 com base no impacto e na facilidade de exploração.

2.3. Pipeline CI/CD e Infraestrutura como Código

O alicerce tecnológico que permite a automação do teste contínuo é o *pipeline* CI/CD. Este componente orquestra um conjunto de fases consecutivas (como *Plan, Code, Build, Test, Deploy*) que definem atividades e ferramentas para auxiliar na automação do desenvolvimento [Santos et al. 2024, Sun et al. 2021].

Em arquiteturas modernas de nuvem, o *pipeline* opera sobre uma infraestrutura responsiva, frequentemente baseada em contêineres e orquestrada por ferramentas como *Kubernetes* [Díaz et al. 2019]. Para garantir a reprodutibilidade e a consistência do ambiente experimental, utiliza-se a Infraestrutura como Código (IaC, do inglês *Infrastructure as Code*), que permite definir recursos de rede, servidores e serviços de forma declarativa. No contexto *DevSecOps*, qualquer falha de segurança detectada nestas fases deve interromper o fluxo, exigindo correção imediata antes da promoção para o próximo estágio [Marandi et al. 2023, Putra and Kabetta 2022].

3. Trabalhos Relacionados

A literatura sobre *DevSecOps* tem crescido, com foco em *frameworks*, estudos de caso e desafios de implementação. Diversos modelos foram propostos para guiar a adoção do *DevSecOps*. Um *framework* de oito fases aplicado em um estudo de caso real (Projeto GRACE) demonstra a integração de SAST, DAST e SCA [Santos et al. 2024]. De forma complementar, arquiteturas de segurança de múltiplas fases focadas na automação [Chen and Suo 2022] e modelos de transformação cultural [Efendi et al. 2021] têm sido estudados para auxiliar organizações na transição do modelo *Waterfall*. Esta pesquisa baseia-se nesses *frameworks* para propor uma implementação prática focada na orquestração de ferramentas.

Uma vertente significativa da literatura foca em estudos de caso que implementam a combinação de SAST e DAST. Implementações práticas que integram essas técnicas em

pipelines utilizando ferramentas como *GitLab* e *Docker* demonstraram redução significativa no tempo de implantação [Putra and Kabetta 2022]. Similarmente, a automação de SAST e DAST em plataformas como *GitHub Actions* [Marandi et al. 2023] e *Azure DevOps* [Kushwaha et al. 2024] destaca a importância do uso de *dashboards* e a validação da viabilidade de orquestrar essas ferramentas em conjunto.

Enquanto alguns trabalhos combinam técnicas, outros aprofundam-se nos desafios de tipos de testes específicos. Análises fundamentais sobre os desafios técnicos de integrar DAST focam no *overhead* (tempo de execução) e na complexidade de automação de ferramentas como o OWASP ZAP [Rangnau et al. 2020]. No outro extremo, análises teóricas sobre SAST comparam as abordagens *white-box* (caixa-branca) e *black-box* (caixa-preta), justificando a necessidade de ambas [Masood and Java 2015]. A varredura de Infraestrutura como Código (*Infrastructure as Code* - IaC) surge como uma camada complementar e crítica em discussões recentes para mitigar vulnerabilidades de provisionamento [Bernardino et al. 2024].

Outros trabalhos focam na arquitetura de *pipeline* e na infraestrutura de automação. Projetos de *pipeline* de teste de segurança visam integrar-se perfeitamente ao CI/CD e unificar os resultados de múltiplas ferramentas em plataformas de gerenciamento centralizadas [Sun et al. 2021]. Por sua vez, a infraestrutura responsiva necessária para suportar processos *DevSecOps* tem sido defendida com o uso de contêineres e orquestradores como base tecnológica para a automação [Díaz et al. 2019].

Finalmente, a literatura aborda os desafios culturais e as lacunas de ferramentas. O contexto cultural e os desafios de mentalidade ao introduzir a segurança em fluxos *DevOps* existentes são amplamente discutidos [Ahmed and Francis 2019]. Desafios práticos, como a complexidade de integração de ferramentas e a dificuldade de automatizar tarefas manuais, levaram à proposta de desenvolvimento de ferramentas customizadas para preencher lacunas específicas [Bernardino et al. 2024]. Essa necessidade de customização é observada em cenários de testes de autenticação, onde mecanismos de defesa como tokens *Cross-Site Request Forgery* (CSRF) impõem barreiras à automação de ferramentas de mercado, demandando o desenvolvimento de *scripts* específicos para viabilizar os testes de intrusão baseados em força bruta.

4. Materiais e Métodos

Para atingir os objetivos propostos, foi definida uma metodologia de **pesquisa experimental**. Este tipo de pesquisa é justificado pelo desenvolvimento de um ambiente controlado e instrumentado, onde variáveis podem ser gerenciadas para observar seus efeitos. O experimento consiste em provisionar uma infraestrutura de nuvem, implementar um *pipeline* CI/CD (*Continuous Integration/Continuous Delivery*) instrumentado com seis camadas de segurança e executar este fluxo contra uma aplicação-alvo intencionalmente vulnerável.

4.1. Ambiente e Infraestrutura

O ambiente experimental foi construído inteiramente na ***Google Cloud Platform*** **(GCP)**, utilizando serviços gerenciados para garantir escalabilidade e integração nativa. O ***Cloud Build*** foi o orquestrador central do *pipeline*. Para a execução da aplicação, utilizou-se o ***Google Kubernetes Engine*** **(GKE)**, um serviço gerenciado de orquestração de

contêineres. O *cluster* foi provisionado no canal de lançamento padrão (*regular channel*), que no período do experimento (janeiro-fevereiro de 2026) correspondia à versão 1.30.x do *Kubernetes*. A configuração utilizou um *node pool* com máquinas do tipo `e2-medium` (2 vCPUs, 4GB RAM) para otimizar custos em ambiente de desenvolvimento. O armazenamento de imagens *Docker* foi gerenciado pelo ***Artifact Registry***, enquanto o ***Cloud Storage* (GCS)** foi configurado como repositório centralizado para os artefatos e relatórios de segurança gerados em cada execução.

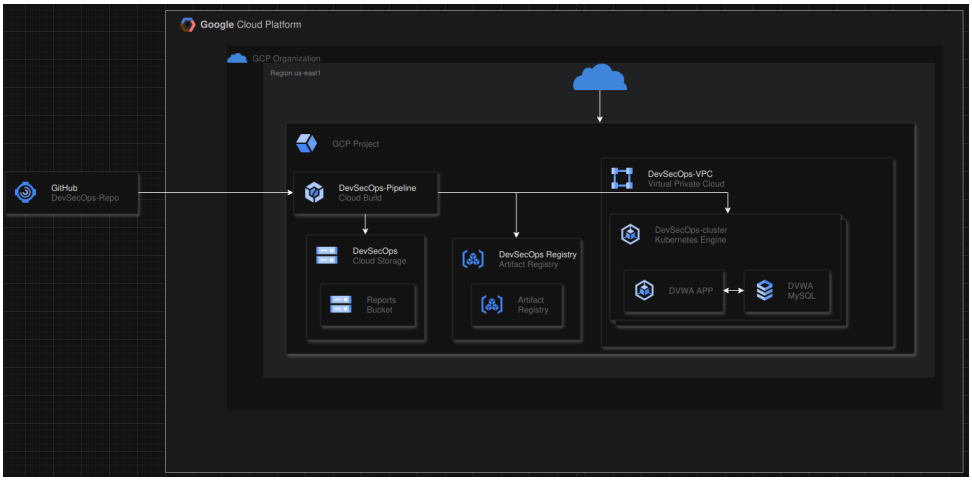

Figura 1. Arquitetura da infraestrutura experimental na GCP, evidenciando a integração entre o Cloud Build, Artifact Registry e o cluster GKE.

A ativação do *pipeline* ocorre de forma automatizada por meio de gatilhos (*triggers*) configurados no *Cloud Build*, vinculados ao repositório de código-fonte. Havendo um *commit* em um ramo (*branch*) específico, a plataforma dispara automaticamente a execução das etapas de segurança. Essa dinâmica simula um ambiente de produção real, no qual o desenvolvimento de novas funcionalidades (*features*) ou atualizações de código é submetido a um ciclo rigoroso de validação antes de ser promovido para a aplicação real.

A **reprodutibilidade** do experimento é garantida pelo uso de Infraestrutura como Código (IaC, do inglês *Infrastructure as Code*). Toda a rede (*VPCs* e sub-redes), o *cluster* GKE e as permissões de acesso (IAM) foram definidos de forma declarativa utilizando o ***Terraform*** (versão $\geq$ 1.5).

4.2. Aplicação-Alvo e Instrumental de Segurança

Como aplicação-alvo, selecionou-se a **DVWA (*Damn Vulnerable Web Application*)**, uma aplicação *web* escrita em PHP/*MySQL* amplamente utilizada na comunidade de segurança por ser intencionalmente vulnerável. O uso desta aplicação é estratégico para a pesquisa experimental, pois ela oferece um catálogo documentado de 17 vulnerabilidades conhecidas, que abrangem categorias críticas do *OWASP Top 10*, como *SQL Injection*, *Cross-Site Scripting* (XSS), *Command Injection* e falhas de autenticação. A existência desse inventário *a priori* permite que a DVWA seja utilizada como um *benchmark* (referencial) para medir a eficácia das ferramentas integradas.

O *pipeline* utiliza a imagem oficial da aplicação (`vulnerables/web-dvwa`), configurada em nível de segurança baixo (*low*), para validar se o instrumental de automação é capaz de identificar as falhas previstas no projeto original da aplicação. O núcleo do experimento é a orquestração das 15 etapas sequenciais no arquivo `cloudbuild.yaml`:

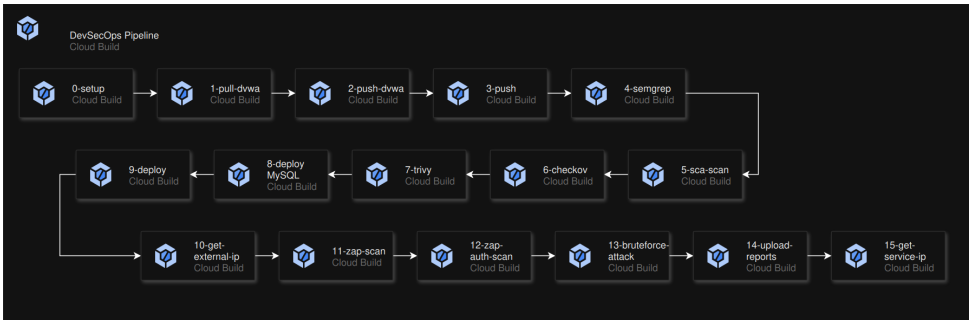

Figura 2. Pipeline de Execução.

- (0) Setup;
- (1) Pull da imagem DVWA pública;
- (2-3) Tag e push da imagem para o Artifact Registry;
- (4) Análise SAST com *Semgrep*;
- (5) Análise SCA com *Trivy*;
- (6) Varredura IaC com *Checkov* nos arquivos *Terraform* e manifestos *Kubernetes*;
- (7) *Container Scan* com *Trivy*;
- (8) *Build* e *deploy MySQL* no *GKE*;
- (9) *Build* e *deploy DVWA* no *GKE*;
- (10) *Get* IP externo para *DAST* + configuração da DVWA em nível *low*;
- (11) *DAST Baseline Scan* com *OWASP ZAP*;
- (12) *DAST Active Scan* com *OWASP ZAP* autenticado;
- (13) *BruteForce Test* com *Script Python*;
- (14) Consolidação dos relatórios no *Cloud Storage*;
- (15) *Get IP* externo da aplicação.

O instrumental de segurança foi selecionado para cobrir múltiplas camadas do modelo de defesa:

- **SAST (*Static Application Security Testing*):** Utilizou-se o ***Semgrep*** para análise estática do código-fonte PHP e *JavaScript* da aplicação.
- **SCA e *Container Scan*:** Empregou-se o ***Trivy*** em duas frentes: primeiro para a Análise de Composição de *Software* (SCA), verificando dependências no código-fonte, e posteriormente para a varredura da imagem *Docker* final, identificando CVEs no sistema operacional e pacotes instalados.
- **IaC Scan:** O ***Checkov*** foi utilizado para analisar os arquivos de configuração do *Terraform* e os manifestos do *Kubernetes*, buscando falhas de segurança na infraestrutura.
- **DAST (*Dynamic Application Security Testing*):** O ***OWASP ZAP*** realizou testes dinâmicos em duas fases: um *Baseline Scan* (análise passiva) e um *Active Scan* autenticado, simulando ataques de injeção em tempo de execução.

- **Teste de Força Bruta:** O intuito inicial para a condução desta etapa era a utilização do ***Hydra***, ferramenta de código aberto amplamente consolidada para ataques de dicionário e força bruta. Contudo, verificou-se que sua cobertura era insuficiente para o cenário específico da aplicação-alvo, uma vez que a DVWA implementa mecanismos de proteção contra *Cross-Site Request Forgery* (CSRF). A dificuldade de ferramentas genéricas em realizar a extração e a propagação de *tokens* dinâmicos para cada tentativa de autenticação exigiu o desenvolvimento de uma solução customizada. Assim, utilizou-se um ***script* em *Python*** capaz de realizar o *parsing* (análise sintática) das respostas HTTP em tempo real, capturando os *tokens* necessários para contornar a proteção e validar a vulnerabilidade de credenciais fracas.

4.3. Coleta de Dados e Método de Análise

A coleta de dados inicia-se ao final de cada execução do *pipeline*, momento em que os relatórios em formato JSON são automaticamente carregados e disponibilizados em um *bucket* do GCS. Para a consolidação e interpretação dos resultados, esses artefatos são recuperados e submetidos ao `analise.py`, um *script Python* robusto desenvolvido especificamente para este experimento. O funcionamento do *script* baseia-se no processamento programático dos diferentes esquemas (*schemas*) de dados gerados por cada ferramenta, implementando funções especializadas para o tratamento de cada formato: `analyze_trivy_container()` para o *Container Scan*, `analyze_semgrep()` para SAST, `analyze_zap_active()` para DAST, `analyze_checkov()` para IaC e `analyze_hydra_bruteforce()` para os testes de força bruta.

Cada função de análise realiza o *parsing* dos objetos JSON e extrai metadados críticos para a avaliação da postura de segurança. Para o *Trivy*, são processados campos como `VulnerabilityID`, `Severity`, `CweIDs` e o *flag* `EOSL` (*End of Support Life*) do sistema operacional. Para o *Semgrep*, extraem-se o `check_id`, a severidade e o *path* do arquivo, além dos metadados de classificação *CWE* e *OWASP*. No caso do *OWASP ZAP*, processam-se os *alerts* com seus respectivos `cweid` e `pluginId`, permitindo validar se os *plugins* de ataque dinâmico foram efetivamente disparados. Por fim, para o *Checkov*, são analisados os `failed_checks` em arquivos de configuração.

A lógica central do *script* reside na função `compare_with_known_vulnerabilities()`, que confronta os achados normalizados com uma matriz de referência (`DVWA_KNOWN_VULNERABILITIES`) contendo as 17 vulnerabilidades conhecidas da DVWA, organizadas nas categorias *web_application* (13 vulnerabilidades) e *infrastructure* (4 vulnerabilidades). Cada item da matriz é mapeado por seu identificador CWE, classificação *OWASP Top 10* e localização prevista no código-fonte.

O *script* implementa ainda a constante `OUT_OF_SCOPE_VULNERABILITIES`, que documenta explicitamente as quatro vulnerabilidades não passíveis de detecção por *pipelines CI/CD* automatizados: *File Inclusion (CWE-98)* , *File Upload (CWE-434)* , *Insecure CAPTCHA (CWE-804)* e *Authorisation Bypass (CWE-639)* . Esta distinção é fundamental para o cálculo de duas métricas distintas: a cobertura geral (vulnerabilidades detectadas sobre o total de 17) e a cobertura ajustada (vulnerabilidades detectadas sobre as 13 automatizáveis), proporcionando uma avaliação técnica precisa da eficácia do *pipeline*.

O resultado final é um relatório em formato *Markdown* que consolida o sumário executivo, a matriz de cobertura e recomendações de melhoria.

5. Resultados e Discussões

Nesta seção, são apresentados e discutidos os dados obtidos a partir da execução do experimento. A análise contempla desde a eficácia da orquestração do fluxo de trabalho até a profundidade da detecção de vulnerabilidades em diferentes camadas, fundamentada nos relatórios gerados pelo *pipeline* e processados pelo *script* de análise.

5.1. Orquestração e Desempenho do Pipeline DevSecOps

O *pipeline* configurado no *Cloud Build* demonstrou estabilidade ao longo das execuções experimentais, completando as 15 etapas sequenciais em um tempo médio de aproximadamente 9 minutos e 30 segundos, condizente com a complexidade das análises integradas. A Tabela 1 detalha o tempo de execução de cada etapa do *pipeline*. O tempo total de execução foi de 9 minutos e 31 segundos, demonstrando a viabilidade da abordagem para integração em ciclos de CI/CD de produção.

Tabela 1. Tempo de execução por etapa do *pipeline*

Etapa	Descrição	Duração	% Total
0. setup	Criar diretórios	00:00:02	0,4%
1. pull-dvwa	*Pull* imagem DVWA	00:00:20	3,5%
2-3. push	*Tag* e *push*	00:00:02	0,4%
4. semgrep	SAST	00:01:30	15,8%
5. sca-scan	SCA (*Trivy FS*)	00:00:39	6,8%
6. checkov	IaC Scan	00:01:06	11,6%
7. trivy	*Container Scan*	00:01:12	12,6%
8. deploy-mysql	*Deploy* MySQL	00:00:40	7,0%
9. deploy	*Deploy* DVWA	00:00:10	1,8%
10. get-ip	Obter IP externo	00:00:06	1,1%
11. setup-dvwa	Configurar DVWA	00:00:10	1,8%
12. zap-scan	DAST *Baseline*	00:02:51	29,9%
13. zap-active	DAST *Active*	00:02:09	22,6%
14. bruteforce	Força bruta	00:00:36	6,3%
15. upload	*Upload* para GCS	00:01:39	17,3%
Total	**-**	**00:09:31**	**100%**

As etapas de análise dinâmica (DAST) representam a maior parcela do tempo total: o *Baseline Scan* consumiu 2 minutos e 51 segundos (29,9%) e o *Active Scan* autenticado 2 minutos e 9 segundos (22,6%), totalizando 52,6% do tempo em testes DAST. Isso é esperado, pois essas ferramentas precisam interagir com a aplicação em execução e aguardar respostas HTTP para cada requisição de teste.

Em contrapartida, as análises estáticas (*Shift-Left*) foram significativamente mais rápidas: *Semgrep* (SAST) executou em 1 minuto e 30 segundos, *Checkov* (IaC) em 1 minuto e 6 segundos, *Trivy Container* em 1 minuto e 12 segundos e *Trivy* SCA em 39

segundos. Juntas, essas etapas representam aproximadamente 47% do tempo total, evidenciando o baixo *overhead* da estratégia *Shift-Left* em relação aos benefícios de detecção precoce.

A automação via *triggers* de *commit* permitiu que, a cada atualização no código ou na infraestrutura, todo o ciclo de segurança fosse reiniciado, garantindo um *feedback* contínuo e imediato ao desenvolvedor. A conformidade dessa orquestração pode ser verificada na Figura 3, que apresenta o histórico de sucesso de todas as etapas do fluxo.

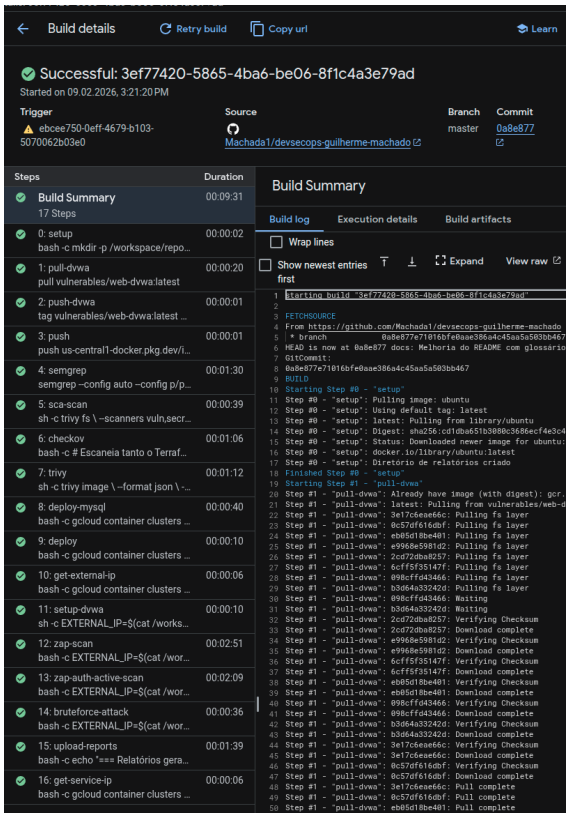

Figura 3. Histórico de execução do pipeline no Cloud Build.

A orquestração obteve sucesso na centralização de artefatos por meio de um *bucket* dedicado no *Cloud Storage*. Para garantir a rastreabilidade e a integridade histórica das auditorias, o *pipeline* foi configurado para organizar os resultados em diretórios nomeados com o *hash* de cada *commit*. Esse volume unificado de dados permitiu que o *script* `analise.py` identificasse um total de **1.748 ocorrências de vulnerabilidades**, majoritariamente relacionadas a CVEs do sistema operacional base no ambiente experimental.

A Tabela 2 apresenta o sumário executivo desses achados, detalhando a produtividade de cada ferramenta integrada no ecossistema de testes.

Tabela 2. Sumário Executivo de achados por ferramenta

Ferramenta	Tipo de Teste	Achados	Status
Trivy	Container Scan	1575	Executado
Semgrep	SAST	77	Executado
Trivy FS	SCA	0	Executado
OWASP ZAP	DAST (Baseline)	19	Executado
OWASP ZAP	DAST (Active Scan)	13	Executado
Checkov	IaC Scan	63	Executado
Script Python	Brute Force	Vulnerável	Executado
Total	**-**	**1.748**	**-**

5.2. Segurança de Infraestrutura e Imagem Base

A análise da camada de infraestrutura revelou o ponto de maior vulnerabilidade técnica do projeto. O *Trivy*, ao realizar o *Container Scan*, identificou que a imagem alvo (`dvwa-app:0a8e877`) utiliza o sistema operacional *Debian 9.5*, categorizado como *End of Support Life* (EOSL). Esta condição expõe a aplicação a 1.575 vulnerabilidades de sistema, das quais 254 são críticas, conforme detalhado na Tabela 3.

Tabela 3. Distribuição de severidade das vulnerabilidades (Trivy)

Severidade	Quantidade	Percentual
CRITICAL	254	16,1%
HIGH	551	35,0%
MEDIUM	642	40,8%
LOW	116	7,4%
UNKNOWN*	12	0,8%
Total	**1.575**	**100,0%**

*Advisories do Debian (DLA/DSA) sem classificação CVSS, incluindo 6 atualizações de `tzdata` e patches de sistema (`libgnutls30`, `libssl1.0.2`).

A classificação de severidade apresentada na Tabela 3 é atribuída pelo próprio *Trivy*, que utiliza uma lógica proprietária baseada em múltiplos fatores além do CVSS, como idade da CVE e existência de *exploits* conhecidos. Para uma análise mais padronizada, as vulnerabilidades foram estratificadas também pela pontuação CVSS v3[1] (*Common Vulnerability Scoring System* versão 3), o sistema numérico mantido pelo NVD que quantifica a severidade de vulnerabilidades em uma escala de 0 a 10, considerando vetores como complexidade de ataque, privilégios necessários e impacto na confidencialidade, integridade e disponibilidade.

Das 1.575 CVEs identificadas, 1.563 (99,2%) possuem pontuação CVSS v3 atribuída pelo NVD, resultando em uma média de **6,98** (limiar entre *Medium* e *High*). A Tabela 4 apresenta a distribuição por faixas de severidade CVSS, evidenciando que

[1]O CVSS v4.0, lançado em novembro de 2023, introduz melhorias como novas métricas de *Attack Requirements* e foco em sistemas OT/ICS. Contudo, sua adoção pelo NVD e ferramentas como *Trivy* ainda está em transição, sendo o CVSS v3.1 o padrão vigente nas bases de dados consultadas neste experimento.

53,2% das vulnerabilidades possuem pontuação igual ou superior a 7.0 (*High* ou *Critical*), corroborando a gravidade da exposição em sistemas com suporte descontinuado.

Tabela 4. Distribuição de vulnerabilidades por faixa CVSS v3

Faixa CVSS	Classificação	Quantidade	Percentual
9.0 – 10.0	Critical	272	17,3%
7.0 – 8.9	High	566	35,9%
4.0 – 6.9	Medium	683	43,4%
0.1 – 3.9	Low	42	2,7%
Sem CVSS	N/A	12	0,8%

A análise por CWE (*Common Weakness Enumeration*) identificou 89 tipos distintos de fraquezas catalogadas nas 1.575 CVEs, totalizando 1.460 associações CWE-CVE (algumas CVEs possuem múltiplos CWEs ou nenhum). A Tabela 5 apresenta as categorias mais frequentes, revelando predominância de vulnerabilidades de memória típicas de código nativo em C/C++ (como *Out-of-bounds Read/Write* e *Buffer Overflow*), características de pacotes de sistema operacional legados como os presentes no Debian 9.5.

Tabela 5. CWEs mais frequentes nas vulnerabilidades detectadas

CWE	Descrição	Ocorrências
CWE-125	Out-of-bounds Read	343
CWE-787	Out-of-bounds Write	148
CWE-190	Integer Overflow	114
CWE-476	NULL Pointer Dereference	92
CWE-20	Improper Input Validation	65
CWE-416	Use After Free	63

A análise aprofundada revelou que pacotes como `libapache2-mod-php7.0` e o próprio servidor `apache2` concentram a maioria das CVEs críticas, como a `CVE-2021-44790` (vulnerabilidade de *buffer overflow* no *mod_lua*). Em contraste, a varredura de **SCA (Software Composition Analysis)** via *Trivy FS* não detectou vulnerabilidades nas dependências declaradas (`composer.lock`, etc.), indicando que o risco reside primariamente na infraestrutura hospedeira e não nas bibliotecas de terceiros.

Complementarmente, o *Checkov* identificou 63 falhas de configuração nos arquivos de Infraestrutura como Código. Como evidenciado na Tabela 6, as não conformidades mais graves referem-se à ausência de políticas de segurança de rede e de registro de *logs* (*Stackdriver Logging*) nos recursos do GKE.

Tabela 6. Principais vulnerabilidades de IaC detectadas pelo Checkov

Check ID	Recurso	Arquivo
CKV_GCP_21	google_container_cluster.primary	gke.tf
CKV_GCP_62	google_storage_bucket.reports	storage.tf
CKV_GCP_49	google_project_iam_member.cloudbuild_builder	iam.tf

5.3. Análise de Segurança da Aplicação (SAST e DAST)

Na camada de aplicação, o *Semgrep* obteve 77 achados, sendo 51 erros e 26 avisos. O mapeamento para o *OWASP Top 10* revelou uma predominância da categoria **A03:2021 - Injection**, com 50 ocorrências relacionadas a injeções de código e SQL. Conclui-se que o uso de funções perigosas como `exec()` em `HealthController.php` e a ausência de sanitização em `low.php` são os principais vetores dos ataques estáticos identificados.

Tabela 7. Distribuição de CWEs via SAST (Semgrep)

CWE	Descrição	Ocorrências
CWE-94	Code Injection	17
CWE-89	SQL Injection	14
CWE-78	OS Command Injection	11
CWE-918	SSRF	11
CWE-697	MD5 Comparison Error	10

A análise dinâmica via *OWASP ZAP* estendeu a visibilidade para o ambiente de execução. O *Baseline Scan* identificou 19 alertas de configuração de segurança, destacando a ausência do cabeçalho *Content Security Policy* (CSP) e a exposição de banners de versão do servidor (CWE-497). O *Active Scan* autenticado confirmou a viabilidade de exploração com um achado de **SQL Injection Crítico** (Figura 5), provando que as falhas apontadas pelo SAST são efetivamente exploráveis em *runtime*.

Alerts

Name	Risk Level	Number of Instances
Content Security Policy (CSP) Header Not Set	Medium	2
Directory Browsing	Medium	3
HTTP Only Site	Medium	1
Missing Anti-clickjacking Header	Medium	1
Relative Path Confusion	Medium	1
Cookie No HttpOnly Flag	Low	4
Cookie without SameSite Attribute	Low	4
In Page Banner Information Leak	Low	1
Insufficient Site Isolation Against Spectre Vulnerability	Low	7
Permissions Policy Header Not Set	Low	2
Server Leaks Version Information via "Server" HTTP Response Header Field	Low	Systemic
X-Content-Type-Options Header Missing	Low	5
Authentication Request Identified	Informational	1
Cookie Slack Detector	Informational	Systemic
Non-Storable Content	Informational	3
Session Management Response Identified	Informational	3
Storable and Cacheable Content	Informational	5
Storable but Non-Cacheable Content	Informational	1
User Agent Fuzzer	Informational	Systemic

Figura 4. Detecções do ZAP Baseline.

Alerts

Name	Risk Level	Number of Instances
SQL Injection	High	1
Application Error Disclosure	Medium	2
Content Security Policy (CSP) Header Not Set	Medium	Systemic
Directory Browsing	Medium	Systemic
HTTP Only Site	Medium	1
Missing Anti-clickjacking Header	Medium	Systemic
In Page Banner Information Leak	Low	2
Information Disclosure - Debug Error Messages	Low	2
Server Leaks Version Information via "Server" HTTP Response Header Field	Low	Systemic
X-Content-Type-Options Header Missing	Low	Systemic
Authentication Request Identified	Informational	1
Information Disclosure - Suspicious Comments	Informational	1
User Agent Fuzzer	Informational	Systemic

Figura 5. Detecção de SQL Injection via Active Scan autenticado no ZAP.

Por fim, o teste de força bruta customizado validou a vulnerabilidade de credenciais fracas ao extrair 102 combinações de *login/senha* bem-sucedidas em um universo de apenas 200 tentativas. Essa taxa de sucesso de 51% é tecnicamente alarmante, pois denota que a aplicação não apenas carece de políticas de complexidade de senhas, mas ignora completamente a implementação de mecanismos de *rate limiting* ou bloqueios de conta após sucessivas falhas (CWE-307).

A Tabela 8 apresenta uma amostra das credenciais obtidas, evidenciando a presença de pares triviais e previsíveis, o que facilita a exploração por ataques de dicionário automatizados.

Tabela 8. Amostra de credenciais vulneráveis identificadas via força bruta

Usuário	Senha
admin	dvwa
admin	123456
admin	guest
root	admin
root	root
root	admin123
user	password

Além da fragilidade das credenciais, o sucesso desta etapa destaca o diferencial da abordagem programática adotada no experimento. Enquanto ferramentas genéricas de mercado falharam ao encontrar obstáculos em proteções de integridade, o *script* em *Python* desenvolvido foi capaz de realizar o *parsing* e a propagação dinâmica de *tokens* CSRF para cada requisição. Isso prova que mecanismos isolados de proteção de formulário são insuficientes se a lógica de autenticação e a gestão de tentativas de acesso não forem devidamente protegidas contra automações direcionadas.

5.4. Eficácia do Pipeline e Matriz de Cobertura

A eficácia final do experimento foi mensurada pelo confronto entre os achados e as 17 vulnerabilidades conhecidas da DVWA. O *pipeline* alcançou a detecção de 13 falhas, atingindo 76,5% de cobertura geral. Entretanto, essa métrica isolada não reflete adequadamente a capacidade do instrumental automatizado. Das 17 vulnerabilidades mapeadas, quatro foram explicitamente classificadas como fora do escopo de automação (`OUT_OF_SCOPE_VULNERABILITIES`): *File Inclusion* (CWE-98), *File Upload* (CWE-434), *Insecure CAPTCHA* (CWE-804) e *Authorisation Bypass* (CWE-639). Essas falhas exigem, respectivamente, a construção manual de *payloads* de inclusão de arquivos, o envio e análise de arquivos maliciosos, a interação com elementos visuais de verificação humana e a compreensão contextual de regras de negócio — tarefas que transcendem as capacidades de varreduras automatizadas convencionais.

Considerando apenas as 13 vulnerabilidades automatizáveis, o *pipeline* atingiu **100% de cobertura ajustada**, demonstrando que o instrumental selecionado é eficaz para a totalidade das classes de vulnerabilidades dentro de seu escopo metodológico. É importante notar que, embora existam ferramentas de automação avançadas como o IAST

(*Interactive Application Security Testing*) que poderiam detectar falhas como o *File Inclusion* (CWE-98), tais soluções não foram integradas neste estágio devido à maior complexidade de configuração e necessidade de agentes no *runtime*. Portanto, o *pipeline* atingiu cobertura integral dentro do escopo metodológico delimitado para este estudo (SAST, DAST convencional, SCA e IaC).

É importante destacar que a metodologia de mapeamento adotada considera dois tipos de detecção: **direta** e **indireta**. A detecção direta ocorre quando a ferramenta identifica a vulnerabilidade específica no código ou comportamento da aplicação (por exemplo, o *Semgrep* encontrando padrões de *SQL Injection* no código PHP). A detecção indireta ocorre quando o *Trivy Container Scan* reporta CVEs em componentes da infraestrutura (sistema operacional, bibliotecas, servidores) cujos identificadores CWE coincidem com vulnerabilidades mapeadas da DVWA. Por exemplo, a CVE-2019-10098 no módulo `mod_rewrite` do Apache possui CWE-601 (*Open Redirect*), indicando que a *stack* que hospeda a aplicação é suscetível a esse tipo de ataque. Essa abordagem é válida do ponto de vista de defesa em profundidade, pois sinaliza riscos na camada de infraestrutura que podem ser explorados em conjunto com vulnerabilidades da aplicação. As detecções indiretas estão marcadas com asterisco (*) na tabela.

A Tabela 9 consolida a matriz de rastreabilidade do experimento, mapeando cada vulnerabilidade ao seu respectivo identificador CWE, à ferramenta responsável pela detecção e ao seu *status* final.

Tabela 9. Matriz de Cobertura: DVWA vs. Pipeline DevSecOps

Vulnerabilidade (DVWA)	CWE	Ferramenta	Origem do Achado	Status
SQL Injection	CWE-89	OWASP ZAP (Active)	*Alert*: SQL Injection em /vulnerabilities/sqli/	Detectado
Cross-Site Scripting (XSS)	CWE-79	Semgrep	Regra: `php.lang.security.xss` em low.php	Detectado
Command Injection	CWE-78	Semgrep	Regra: `php.lang.security.exec-use` em low.php	Detectado
CSRF	CWE-352	Trivy (Container)*	CVE-2018-1000858 em gpgv (CWE-352)	Detectado
Weak Session IDs	CWE-330	Trivy (Container)*	CVE-2019-1543 em OpenSSL (CWE-330)	Detectado
Brute Force	CWE-307	Script Python	102 credenciais válidas em 200 tentativas	Detectado
Open HTTP Redirect	CWE-601	Trivy (Container)*	CVE-2019-10098 em Apache mod_rewrite (CWE-601)	Detectado
JavaScript Attacks	CWE-749	Semgrep	Regra: `javascript.browser.security` em /javascript/	Detectado
Content Security Policy Bypass	CWE-693	OWASP ZAP (Baseline)	*Alert*: CSP Header Not Set	Detectado
Outdated OS	CWE-1104	Trivy (EOSL)	Debian 9.5 - End of Support Life	Detectado
Outdated Packages	CWE-1104	Trivy (Container)	254 CVEs críticas em pacotes do SO	Detectado
Default Credentials	CWE-798	Script Python	admin/password, admin/dvwa confirmados	Detectado
Exposed MySQL	CWE-284	Trivy (Container)*	CVE-2021-21703 em PHP-FPM (CWE-284)	Detectado
File Inclusion (LFI/RFI)	CWE-98	-	Requer *payload* manual específico	Não Detectado
File Upload	CWE-434	-	Requer upload e análise de arquivo	Não Detectado
Insecure CAPTCHA	CWE-804	-	Requer interação humana	Não Detectado
Authorisation Bypass	CWE-639	-	Requer análise de lógica de negócio	Não Detectado

** Detecção indireta: O Trivy identificou CVEs em componentes da infraestrutura (sistema operacional, bibliotecas) cujos CWEs coincidem com as vulnerabilidades mapeadas da DVWA. Embora não sejam a mesma instância de vulnerabilidade no código da aplicação, indicam que a stack tecnológica é suscetível a esse tipo de ataque.*

A análise das quatro vulnerabilidades não detectadas revela uma limitação inerente e documentada da automação em segurança. Conforme detalhado nos relatórios de análise, falhas como *Authorisation Bypass* (CWE-639) e *File Inclusion* (CWE-98) exigem ou o entendimento profundo da lógica de negócio para identificar privilégios inadequados, ou a construção manual de *payloads* específicos que ferramentas de varredura genérica falham em emular de forma autônoma. Portanto, os resultados ratificam que a integração de testes contínuos deve ser encarada como uma camada de proteção escalável de pri-

meira linha, mas que não substitui a necessidade de revisões humanas e testes de intrusão (*pentests*) manuais para vetores de ataque que dependem de contexto.

5.5. Ameaças à Validade

A principal ameaça à validade externa refere-se à generalização dos resultados: a *DVWA* é uma aplicação intencionalmente vulnerável, e a taxa de detecção pode diferir em aplicações de produção com código mais complexo. Quanto à validade interna, o experimento foi conduzido em um único provedor de nuvem (GCP), e variações de configuração em outras plataformas podem influenciar os resultados.

Adicionalmente, uma limitação de construto (*construct validity*) reside na estratégia de mapeamento por identificadores *CWE*. O *script* de análise associa vulnerabilidades detectadas às falhas conhecidas da *DVWA* através de seus identificadores *CWE*. Contudo, ferramentas como o *Trivy Container* reportam *CVEs* em pacotes do sistema operacional (PHP, Apache, bibliotecas) que possuem *CWEs* associados. Esses identificadores podem coincidir com vulnerabilidades de aplicação *web* da *DVWA*, gerando mapeamentos que, embora tecnicamente corretos do ponto de vista taxonômico, referem-se a contextos distintos. Por exemplo, uma *CVE* em um pacote PHP com CWE-352 (CSRF) não corresponde à mesma vulnerabilidade *CSRF* implementada intencionalmente no código da aplicação. Essa característica foi tratada no presente trabalho através da distinção entre detecções diretas e indiretas na matriz de cobertura, permitindo transparência sobre a natureza de cada achado. Reconhece-se que detecções indiretas representam indicadores de risco na *stack* tecnológica, e não a exploração confirmada da vulnerabilidade específica da aplicação.

6. Conclusão

Este trabalho apresentou a implementação e a análise de um ecossistema *DevSecOps* automatizado, demonstrando como a integração de testes de segurança contínuos em *pipelines* CI/CD podem elevar a maturidade de segurança de aplicações *web*. A utilização da DVWA como aplicação-alvo permitiu uma validação quantitativa e qualitativa rigorosa do instrumental selecionado.

6.1. Síntese dos Resultados e Contribuições

Os resultados demonstram que a automação é uma aliada indispensável na detecção precoce de falhas. O *pipeline* foi capaz de processar 15 etapas críticas em um tempo de aproximadamente 9 minutos e 30 segundos. A identificação de 1.748 ocorrências de vulnerabilidades evidencia a profundidade das varreduras, com destaque para a infraestrutura legada e a detecção de vulnerabilidades de injeção em tempo de execução.

Atingir uma cobertura de 76,5% das vulnerabilidades conhecidas da DVWA valida a robustez do instrumental selecionado. O experimento demonstrou que a estratégia de *Shift Left* é altamente eficaz para filtrar vulnerabilidades técnicas e de configuração de forma automatizada. Contudo, reconhece-se que este índice de cobertura poderia ser ampliado com a inclusão de ferramentas interativas (IAST), que foram desconsideradas no presente escopo para priorizar ferramentas de menor fricção e mais ampla adoção em *pipelines* de mercado. Assim, o aproveitamento total obtido refere-se à eficácia máxima das categorias de testes implementadas, e não à totalidade das possibilidades de automação existentes na literatura.

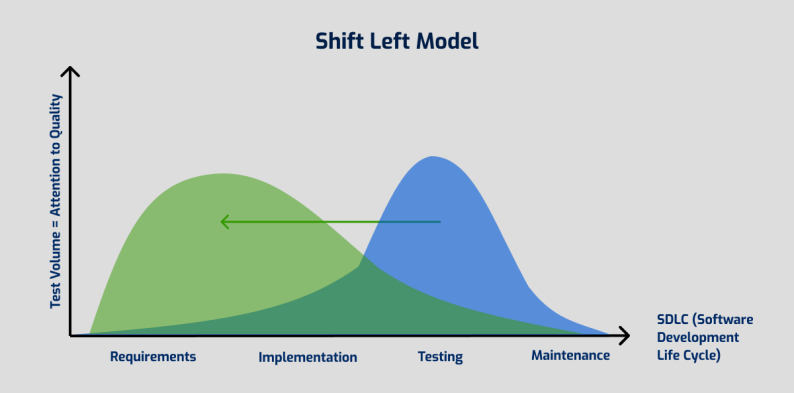

Figura 6. Representação do modelo Shift Left aplicado ao ciclo de desenvolvimento.

6.2. Diretrizes para Mitigação das Vulnerabilidades Identificadas

Com base no volume unificado de dados processados, as seguintes estratégias de remediação são propostas para as camadas analisadas:

- **Segurança de Contêiner (Trivy):** A correção definitiva para as 1.575 vulnerabilidades de sistema requer a migração da imagem base *Debian 9.5* (EOSL) para uma distribuição com suporte ativo, como *Debian 12* ou *Alpine Linux*. Isso eliminaria vetores críticos como o `CVE-2021-44790` no Apache.
- **Infraestrutura como Código (Checkov):** Para as 63 falhas de IaC, recomenda-se o endurecimento (*hardening*) via *Terraform*, habilitando explicitamente *Network Policies* (CKV_GCP_21) para isolamento de rede e ativando o *Stackdriver Logging* para auditoria. Além disso, deve-se implementar o *SecurityContext* nos manifestos para restringir privilégios de execução.
- **Segurança da Aplicação (Semgrep e ZAP):** As 77 falhas de SAST e as detecções de *SQL Injection* no DAST exigem a substituição de funções perigosas como `exec()` e a adoção de *prepared statements* para consultas ao banco de dados. Complementarmente, a configuração correta de cabeçalhos *Content Security Policy* (CSP) mitigaria os riscos de injeção detectados no *Baseline Scan*.
- **Autenticação e Força Bruta:** A vulnerabilidade de força bruta deve ser mitigada pela implementação de mecanismos de *rate limiting*, bloqueio temporário de contas após sucessivas falhas (*account lockout policies*) e a obrigatoriedade de políticas de senhas fortes para neutralizar os 102 acessos bem-sucedidos identificados.

Para ambientes de produção, recomenda-se ainda a configuração de *thresholds* de severidade no *pipeline*, definindo critérios de falha automática do *build*. Por exemplo, o fluxo pode ser configurado para interromper a execução apenas quando forem detectadas CVEs com classificação *CRITICAL* e pontuação CVSS ≥ 9.0, enquanto vulnerabilidades de menor severidade são registradas para tratamento posterior. Adicionalmente, o uso de *allowlists* permite documentar CVEs conhecidas e aceitas pelo time de segurança, evitando bloqueios recorrentes por falsos positivos ou vulnerabilidades já mitigadas por outros controles. O *pipeline* implementado neste experimento opera em modo de auditoria (não bloqueia o *deploy*), porém a arquitetura permite a configuração de *exit codes* condicionais para ambientes com requisitos de conformidade mais rigorosos.

6.3. Trabalhos Futuros e Considerações Finais

As limitações encontradas — especificamente nas 4 vulnerabilidades que exigiam análise de lógica de negócio — definem as fronteiras da automação. Conclui-se que o *DevSecOps* atua como um filtro de alta eficiência, mas não elimina a necessidade de *pentests* manuais para ameaças complexas.

Como contribuição técnica, o desenvolvimento do *script* `analise.py` provou ser fundamental para a consolidação de métricas heterogêneas, transformando dados brutos em uma ferramenta de *insights* completos. Para trabalhos futuros, esta pesquisa abre caminho para a integração de ferramentas de *IAST (Interactive Application Security Testing)*, que operam com agentes no *runtime* da aplicação e poderiam elevar a cobertura para vulnerabilidades como *File Inclusion* (CWE-98). Adicionalmente, pretende-se investigar: (i) a geração automatizada de sugestões de correção (*fix suggestions*) integradas ao fluxo de revisão de código; (ii) a extensão do pipeline para arquiteturas de microsserviços, avaliando a comunicação inter-serviços e APIs; e (iii) a integração com plataformas de gerenciamento de vulnerabilidades para priorização e acompanhamento do ciclo de vida das correções.

O código-fonte completo do experimento está disponível em repositório público para garantir a reprodutibilidade[2].

Agradecimentos

À Thaís Peixoto, minha noiva, pelo apoio incondicional e pela compreensão durante as inúmeras horas de dedicação a esta pesquisa. Ao meu amigo Bryan Damasceno, por despertar minha curiosidade e interesse pela área de segurança da informação; seu incentivo foi o catalisador necessário para que eu buscasse o conhecimento técnico aprofundado apresentado neste estudo.

Aos professores Lesandro Ponciano e Fabio Cordeiro, pela oportunidade acadêmica e pela orientação fundamental que permitiram a viabilização deste trabalho.

À empresa *SantoDigital*, pela infraestrutura cedida para a realização dos experimentos no ambiente GCP da organização. Agradeço, primordialmente, pela vivência profissional diária, que foi o alicerce para a obtenção da experiência técnica e maturidade necessárias para projetar e executar um fluxo de automação desta complexidade.

Por fim, aos meus amigos e familiares, pelo incentivo e paciência ao longo da jornada para a obtenção deste diploma.

Referências

[Ahmed and Francis 2019] Ahmed, Z. and Francis, S. C. (2019). Integrating security with devsecops: Techniques and challenges. In *2019 International Conference on Digitization (ICD)*, pages 178–182.

[Bernardino et al. 2024] Bernardino, N. A., Sequeira, B., Piza, E., Henriques, F., Neves, F., and Reis, C. I. (2024). Enhancing devsecops: Three custom tools for continuous security. In *2024 IEEE 11th International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 53–58.

[2] `https://github.com/Machada1/TCC-DevSecOps/tree/main`

[Chen and Suo 2022] Chen, T. and Suo, H. (2022). Design and practice of security architecture via devsecops technology. In *2022 IEEE 13th International Conference on Software Engineering and Service Science (ICSESS)*, pages 310–313.

[Díaz et al. 2019] Díaz, O., Muñoz, M., and Mejía, J. (2019). Responsive infrastructure with cybersecurity for automated high availability devsecops processes. In *2019 8th International Conference On Software Process Improvement (CIMPS)*, pages 1–9.

[Efendi et al. 2021] Efendi, M., Raharjo, T., and Suhanto, A. (2021). Devsecops approach in software development case study: Public company logistic agency. In *2021 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, pages 96–101.

[Kushwaha et al. 2024] Kushwaha, M. K., David, P., and Suseela, G. (2024). Automation and devsecops: Streamlining security measures in financial system. In *2024 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–6.

[Marandi et al. 2023] Marandi, M., Bertia, A., and Silas, S. (2023). Implementing and automating security scanning to a devsecops ci/cd pipeline. In *2023 World Conference on Communication & Computing (WCONF)*, pages 1–6.

[Masood and Java 2015] Masood, A. and Java, J. (2015). Static analysis for web service security - tools & techniques for a secure development life cycle. In *2015 IEEE International Symposium on Technologies for Homeland Security (HST)*, pages 1–6.

[Putra and Kabetta 2022] Putra, A. M. and Kabetta, H. (2022). Implementation of devsecops by integrating static and dynamic security testing in ci/cd pipelines. In *2022 IEEE International Conference of Computer Science and Information Technology (ICOSNIKOM)*, pages 1–6.

[Rangnau et al. 2020] Rangnau, T., v. Buijtenen, R., Fransen, F., and Turkmen, F. (2020). 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)*, pages 145–154.

[Santos et al. 2024] Santos, N., Escravana, N., Pacheco, B., and Feio, C. (2024). An empirical study of devsecops focused on continuous security testing. In *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 610–617.

[Sun et al. 2021] Sun, X., Cheng, Y., Qu, X., and Li, H. (2021). Design and implementation of security test pipeline based on devsecops. In *2021 IEEE 4th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, volume 4, pages 532–535.