

UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"
FACULDADE DE CIÊNCIAS - CAMPUS BAURU
DEPARTAMENTO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ARTUR KYUNG MIN LEE

**IMPLEMENTAÇÃO DO DEVOPS PARA GERENCIAMENTO DE
RISCOS DE SEGURANÇA NA NUVEM**

BAURU
Novembro/2023

ARTUR KYUNG MIN LEE

IMPLEMENTAÇÃO DO DEVOPS PARA GERENCIAMENTO DE RISCOS DE SEGURANÇA NA NUVEM

Trabalho de Conclusão de Curso do Curso
de Ciência da Computação da Universidade
Estadual Paulista “Júlio de Mesquita Filho”,
Faculdade de Ciências, Campus Bauru.
Orientador: Prof. Dr. Kleber Rocha de Oliveira

BAURU
Novembro/2023

Artur Kyung Min Lee

Implementação do DevOps para Gerenciamento de Riscos de Segurança na Nuvem/ Artur Kyung Min Lee, 2023

46 p. : il. (algumas color.)

Orientador: Prof. Dr. Kleber Rocha de Oliveira

Monografia (Ciências da Computação)–Universidade Estadual Paulista (Unesp). Faculdade de Ciências, Bauru, 2023

1. Leitura (Ensino superior). 2. Livros e leitura. 3. Leitura – Meios auxiliares. I. Universidade Estadual Paulista. Faculdade de Ciências. II. Implementação do DevOps para Gerenciamento de Riscos de Segurança na Nuvem.

Artur Kyung Min Lee

Implementação do DevOps para Gerenciamento de Riscos de Segurança na Nuvem

Trabalho de Conclusão de Curso do Curso de Ciência da Computação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus Bauru.

Banca Examinadora

Prof. Dr. Kleber Rocha de Oliveira

Orientador

Universidade Estadual Paulista "Júlio de Mesquita Filho"

Faculdade de Ciências

Departamento de Ciência da Computação

Professor Simone das Graças Domingues Prado

Universidade Estadual Paulista "Júlio de Mesquita Filho"

Faculdade de Ciências

Departamento de Ciência da Computação

Professor Higor Armario de Souza

Universidade Estadual Paulista "Júlio de Mesquita Filho"

Faculdade de Ciências

Departamento de Ciência da Computação

Bauru, 20 de novembro de 2023.

Espaço destinado à dedicatória do texto.

Agradecimentos

À minha família Joo Hee Park, Do Heon Lee, Celina Seo Jin Lee e Íris Lumi Tateishi por estarem sempre ao meu lado.

Ao Ari Neto de Oliveira por ser meu mentor e ter me ajudado a iniciar minha jornada em *Cloud Security*.

Ao Luca Beraldo Basílio e Fernando Lazar por me auxiliar no desenvolvimento do ShotBot.

Aos meus amigos que fiz durante o curso Pedro, João, Giulia, Larissa, Geovanna e Larissa.

Resumo

A conjuntura do mundo da tecnologia nos últimos anos permitiu com que a computação em nuvem progredisse de forma exponencial por ter diversos benefícios como a velocidade, a escalabilidade, o custo, a disponibilidade e outros. Porém, assim como novas tecnologias de nuvem foram surgindo, ameaças cibernéticas também foram aumentando. Hoje, tornou-se necessário que todo ambiente em *cloud* tenha ferramentas que sejam capazes de manter a integridade, confiabilidade e disponibilidade das aplicações nesse ambiente. Entende-se que uma das principais funções das ferramentas de segurança é de detectar falhas ou ações suspeitas que devem ser bloqueadas em prol da proteção da aplicação. Por conta disso, é vital que as tecnologias de segurança sejam implementadas com um estudo prévio para que possam incorporar-se ao fluxo da aplicação sem anular os principais benefícios da nuvem. Esse trabalho propõe integrar ferramentas de segurança de maneira que, as mesmas não interfiram de negativamente no fluxo da aplicação. Seguindo as melhores práticas do DevOps, será criado uma aplicação/programa na nuvem (AWS) que tenha fluxo de integração e entrega contínuas onde serão integradas as ferramentas responsáveis por trazer segurança e visibilidade ao ambiente. **Palavras-chave:** Nuvem, DevOps, Segurança

Abstract

The technology landscape in recent years has allowed cloud computing to progress exponentially, thanks to various benefits such as speed, scalability, cost-effectiveness, availability, and more. However, as new cloud technologies have emerged, cyber threats have also increased. Today, it has become necessary for every cloud environment to have tools capable of maintaining the integrity, reliability, and availability of applications in this environment. It is understood that one of the main functions of security tools is to detect faults or suspicious actions that should be blocked in favor of application protection. Therefore, it is vital that security technologies are implemented with prior consideration to seamlessly integrate into the application flow without nullifying the key benefits of the cloud. This work proposes to integrate security tools in a way that they do not negatively impact the application flow. Following DevOps best practices, an application/program will be created in the cloud (AWS) that has a continuous integration and delivery flow where tools responsible for bringing security and visibility to the environment will be integrated. **Keywords:** Cloud, DevOps, Security

Lista de figuras

Figura 1 – Loop do DevOps.	15
Figura 2 – Exemplo de código de um <i>workflow</i>	18
Figura 3 – Exemplo 1 de IaC - Terraform	21
Figura 4 – Exemplo 2 - CloudFormation	22
Figura 5 – Exemplo de um arquivo de estado	23
Figura 6 – Exemplo da console do Terraform Cloud	24
Figura 7 – Desenho dos recursos na <i>Cloud</i> para o funcionamento da auto remediação .	27
Figura 8 – Exemplo de comando do ShotBot no <i>Discord</i>	29
Figura 9 – Trecho do código do ShotBot	29
Figura 10 – Desenho da arquitetura da aplicação	30
Figura 11 – Trecho do terraform para o ShotBot	31
Figura 12 – Pipeline construída para o trabalho	32
Figura 13 – Trecho do código responsável pelo <i>lint</i>	32
Figura 14 – Trecho do código responsável pelo Checkov e pelo Terraform	33
Figura 15 – Trecho do código responsável por fazer o deploy do código	34
Figura 16 – Porcentagem de conformidade da conta hospedando o ShotBot	35
Figura 17 – Tela de alertas detectados pelo Conformity	35
Figura 18 – Diagrama do Pipeline	37
Figura 19 – Exemplo caso existam erros pegos pelo Black	38
Figura 20 – Exemplo na console do Terraform Cloud sobre a criação de recursos na cloud	39
Figura 21 – Configuração no Terraform permitindo acesso irrestrito na porta 22	40
Figura 22 – Checkov bloqueando a configuração que foi submetida	40
Figura 23 – Card criado no Jira mostrando a falha detectada durante o fluxo	41
Figura 24 – Botão de envio para o Jira	41
Figura 25 – Criação da chave KMS	42
Figura 26 – Conformity apontando falha na regra de rotatividade	42
Figura 27 – Após um tempo com a correção feita	43

Lista de abreviaturas e siglas

AWS	<i>Amazon Web Services</i>
CD	<i>Continuous Development</i> (Desenvolvimento Contínuo)
CI	<i>Continuous Integration</i> (Integração Contínua)
CSPM	<i>Cloud Security Posture Management</i>
CVE	<i>Common Vulnerabilities and Exposures</i>
DevOps	abreviação de " <i>Development</i> " e " <i>Operations</i> " (Desenvolvimento e Operações).
DevSecOps	abreviação de " <i>Development</i> ", " <i>Operations</i> " e " <i>Security</i> " (Desenvolvimento, Operações e Segurança).
HCL	<i>HashiCorp Configuration Language</i>
IaC	<i>Infrastructure as Code</i>
IBM	<i>International Business Machines</i>
SSM	<i>Simple Systems Manager</i>

Sumário

1	INTRODUÇÃO	12
1.1	Problemática	13
1.1.1	Problema	13
1.1.2	Justificativa	13
1.2	Objetivo	14
1.2.1	Objetivo Geral	14
1.2.2	Objetivos Específicos	14
1.3	Organização da monografia	14
2	REVISÃO TEÓRICA	15
2.1	DevOps	15
2.1.1	Definição	15
2.1.1.1	Planejamento	16
2.1.1.2	Integração	16
2.1.1.3	Teste	16
2.1.1.4	Implementação	16
2.1.1.5	Operação	16
2.1.1.6	Aprendizado	16
2.1.2	CI/CD	17
2.1.2.1	Definição	17
2.1.2.2	Github Actions	17
2.1.3	Cultura DevOps	18
2.1.4	DevSecOps	19
2.1.5	AWS	19
2.1.5.1	EC2	19
2.1.5.2	VPC	19
2.1.5.3	Lambda	19
2.1.5.4	SNS	20
2.1.5.5	Internet Gateway	20
2.1.5.6	Security Group	20
2.2	Infraestrutura como código	20
2.2.1	Definição	20
2.2.2	Terraform	22
2.2.3	Terraform Cloud	23
2.2.4	Checkov	24

2.3	Cloud Security Posture Management	25
2.3.1	Definição	25
2.3.2	<i>Conformity</i>	25
2.3.2.1	Funcionamento	25
2.3.2.2	Integrações	26
2.3.2.3	Auto Remediação	26
3	METODOLOGIA	28
3.1	Aplicativo	28
3.1.1	Discord	28
3.1.2	ShotBot	28
3.2	Arquitetura da aplicação	30
3.2.1	Terraform	31
3.3	GitHub Actions	32
3.3.1	Lint	32
3.3.2	Terraform	33
3.3.3	Deploy	34
3.4	Ferramentas de segurança	34
3.4.1	Conformity	34
3.4.2	Alertas	34
3.4.3	Auto Remediação	35
4	RESULTADOS	37
4.1	Pipeline	37
4.2	Testes no Checkov	39
4.3	Testes no Conformity	40
4.3.1	Fluxo natural	40
4.3.2	Auto-Remediação	42
5	CONCLUSÃO	44
	REFERÊNCIAS	46

1 Introdução

A adoção da computação em nuvem tem ultrapassado grandes obstáculos para as infraestruturas computacionais tradicionais. Dentre as vantagens da computação em nuvem estão a possibilidade de acesso aos dados, a aplicação a distância (desde que haja conexão de qualidade com a internet), o modelo de pagamento pelo uso e a escalabilidade (PEDROSA, 2011). Devido a esses fatores, há uma forte tendência no setor de TI de migrar para a nuvem. Até 2025, mais da metade dos gastos em TI nas categorias de software de aplicação e infraestrutura e serviços de processos de negócios terão migrado para a nuvem (em comparação com 41% em 2022). Além disso, quase dois terços dos gastos em softwares de aplicação serão destinados às tecnologias em nuvem em 2025, em comparação com 57,7% em 2022 (GARTNER, 2022).

O conhecimento sobre quais são os fatores que influenciam no uso da nuvem pode contribuir tanto no âmbito acadêmico, para ampliar os estudos nessa área, quanto no âmbito de negócios auxiliando as empresas, os usuários e os provedores de serviços na nuvem na avaliação das suas necessidades e executarem as ações conforme seus objetivos (VIEIRA; MEIRELLES, 2015). Surge, então, o desafio de como implantar atualizações de serviço de forma rápida e frequente, sem comprometer a confiabilidade e estabilidade do ambiente operacional (CROWLEY, 2018).

DevOps é a combinação de filosofias, práticas e ferramentas que aumentam a capacidade de uma operação de distribuir aplicativos e serviços em alta velocidade. Otimizando e aperfeiçoando produtos em um ritmo mais rápido do que o das empresas que usam processos tradicionais de desenvolvimento e gerenciamento de infraestrutura (AWS, 2023). Com a implementação de um modelo de DevOps, as equipes de desenvolvimento e operações não trabalham mais separadamente. Os engenheiros trabalham durante o ciclo de vida inteiro do aplicativo: da fase de desenvolvimento e testes à fase de implantação e operações. (AMAZON, s.d.).

Um conceito importante do DevOps é a pipeline. Uma pipeline de DevOps é um conjunto de processos e ferramentas automatizados que permite que desenvolvedores e profissionais de operações trabalhem de maneira coesa na criação e implementação de código em um ambiente de produção. Um dos principais fundamentos do DevOps é criar um fluxo para a implantação em produção com qualidade, feedback rápido e experimentação contínua (MUNIZ, 2019). Em alguns modelos de DevOps, as equipes de controle de qualidade e segurança também podem aumentar sua integração com todo o ciclo de vida de um aplicativo. Quando se incorpora a segurança, pode-se denominar como DevSecOps. (AMAZON, s.d.).

1.1 Problemática

1.1.1 Problema

A segurança é o desafio mais visível a ser enfrentado no ambiente na nuvem. A informação que antes era armazenada localmente irá localizar-se na nuvem em local físico que não se tem precisão onde é e nem que tipos de dados estão sendo armazenados junto a ela. A privacidade e integridade das informações são então itens de suma importância, pois especialmente em nuvens públicas existe uma grande exposição a ataques (PEDROSA et al, 2011).

A IBM conduziu uma pesquisa que revelou que 45% das violações de dados ocorreram na nuvem. Além disso, em 2022, foi necessário, em média, 277 dias - cerca de 9 meses - para identificar e conter uma violação. Outro dado relevante é que, das empresas entrevistadas, 79% admitiram que a segurança na nuvem era um desafio para elas (IBM, 2022).

Quando utilizamos os serviços dos maiores provedores de nuvem atuais (AWS, Azure, etc), existe o conceito de responsabilidade compartilhada. A AWS descreve esse conceito como segurança e conformidade são uma responsabilidade compartilhada entre o provedor e o cliente. Esse modelo compartilhado pode auxiliar a reduzir os encargos operacionais do cliente à medida que o fornecedor é responsável por proteger a infraestrutura que executa todos os serviços oferecidos na nuvem. Já o cliente, tem a responsabilidade determinada pelos serviços de nuvem selecionados por ele (AMAZON, s.d.).

Para proteger o ambiente na nuvem, o usuário tem à disposição diversas ferramentas de segurança. Essas ferramentas podem ser fornecidas pela provedora ou por terceiros. Elas permitem monitorar, revisar e detectar possíveis configurações incorretas no ambiente. No entanto, um dos papéis fundamentais das ferramentas de segurança é a capacidade bloquear componentes suspeitos ou maliciosos, que podem impactar diretamente o funcionamento do ambiente, especialmente se o serviço afetado for considerado crítico. Esse bloqueio pode interromper o ciclo de vida da operação por tempo indeterminado, até que a ameaça seja neutralizada.

1.1.2 Justificativa

DevOps é um conjunto de ferramentas, técnicas e cultura que buscam integrar e automatizar os processos de desenvolvimento e operação. É fundamental escolher ferramentas que sejam compatíveis com as tecnologias do ambiente para evitar interrupções ou lentidão no ciclo de vida da operação. Para isso, é importante ter um bom entendimento das capacidades de cada ferramenta e integrá-las corretamente na esteira, especialmente quando se trata de ferramentas de segurança, uma vez que entende-se o grande impacto que as mesmas podem ter caso mal gerenciadas.

1.2 Objetivo

1.2.1 Objetivo Geral

Arquitetar e implementar uma *pipeline* que integre ferramentas de segurança focadas em gerenciamento de risco na nuvem respeitando o conceito do DevOps.

1.2.2 Objetivos Específicos

Para atingir o objetivo final do trabalho, as próximas etapas foram concluídas:

- Criar uma aplicação que rode na nuvem
- Arquitetar e implementar uma *pipeline* básica
- Integrar ferramentas de gerenciamento de risco
- Gerar evidências do fluxo da aplicação

1.3 Organização da monografia

O trabalho está organizado na seguinte forma:

- Capítulo 2: Revisão teórica de conceitos que são importantes de saber para facilitar compreensão do projeto
- Capítulo 3: Aplicação desenvolvida para o trabalho
- Capítulo 4: Arquitetura *cloud* desenvolvida para a aplicação
- Capítulo 5: *Pipeline* desenvolvida para o trabalho
- Capítulo 6: Ferramentas de segurança integradas à pipeline
- Capítulo 7: Resultados
- Capítulo 8: Conclusão

Os capítulos 3 ao 6 abordam o desenvolvimento do trabalho

2 Revisão Teórica

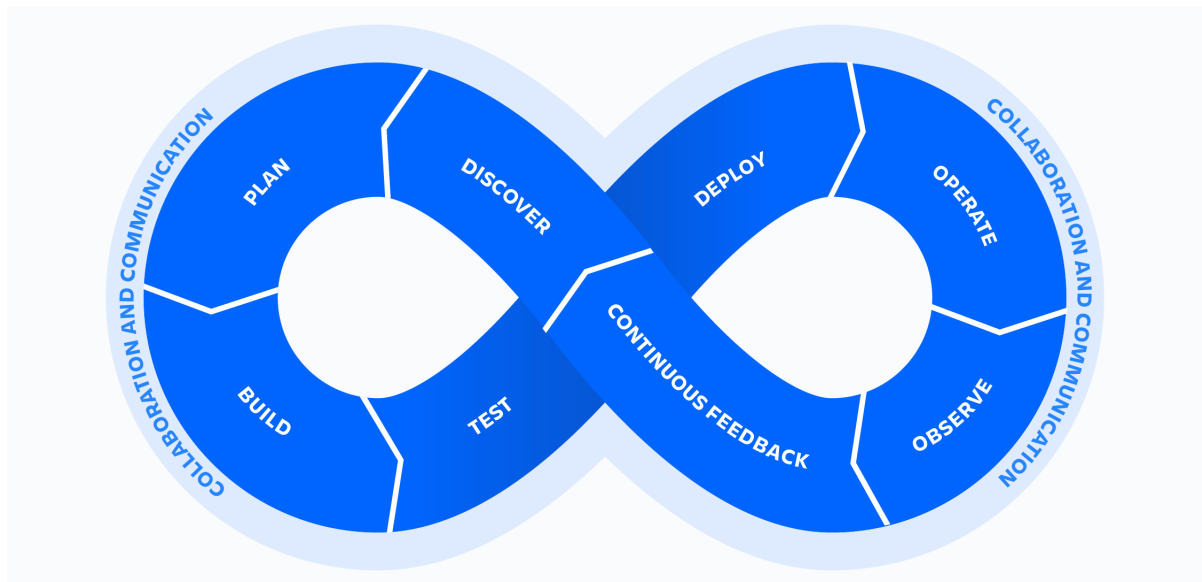
2.1 DevOps

2.1.1 Definição

DevOps é um conjunto de práticas, ferramentas e uma filosofia cultural que automatizam e integram os processos entre o desenvolvimento de software e as equipes de TI. Ele enfatiza o empoderamento da equipe, a comunicação e a colaboração entre equipes e a automação da tecnologia. (ATLASSIAN, 2022)

A metodologia DevOps descreve abordagens que ajudam a acelerar os processos necessários para levar uma ideia do desenvolvimento à implantação em um ambiente de produção no qual ela seja capaz de gerar valor para o usuário. Essas ideias podem ser um novo recurso de software, uma solicitação de aprimoramento ou uma correção de bug, entre outros. Essas abordagens exigem comunicação frequente entre as equipes de desenvolvimento e operações, trabalho colaborativo e empatia com os demais membros das equipes. Também são necessários provisionamento flexível e escalabilidade. (REDHAT, 2022a)

Figura 1 – Loop do DevOps.



Fonte: (ATLASSIAN, 2022)

A Figura 1 mostra um conceito importante que é o *Loop* do *DevOps*. A principal ideia desse conceito é representar o ciclo de desenvolvimento de uma aplicação. A IBM define cada passo desse ciclo da seguinte maneira:

2.1.1.1 Planejamento

Utilizando o feedback do usuário e estudos de caso, o objetivo no estágio de planejamento é maximizar o valor de negócio do produto, produzindo uma lista não processada de recursos que, quando entregues, produzem um resultado desejado que tenha valor.

2.1.1.2 Integração

Atividades de automação comuns incluem mesclar mudanças de código em uma cópia "mestre", verificar esse código de um repositório de código-fonte e automatizar a compilação, teste de unidade e empacotamento em um executável.

2.1.1.3 Teste

Testes realizados para testar todas as etapas da aplicação como: desenvolvimento orientado por comportamento, testes de unidade, varreduras de CVE, linting, teste de fumaça, teste de configuração, teste de conformidade e etc. O teste é uma forma poderosa para a identificação de risco e vulnerabilidade e fornece uma oportunidade para a TI aceitar, minimizar ou corrigir riscos.

2.1.1.4 Implementação

O resultado da integração é implementado a um ambiente de desenvolvimento em diversos testes serão realizados. Se erros ou defeitos forem encontrados, os desenvolvedores terão a chance de interceptar e corrigir qualquer problema antes que qualquer usuário final os veja. Normalmente, existem ambientes para desenvolvimento, teste e produção, com cada ambiente exigindo portões de qualidade progressivamente mais "rígidos". Caso a aplicação passe por todos os testes e seja considerada estável, ela é entregue ao cliente final.

2.1.1.5 Operação

As operações garantem que os recursos estejam funcionando sem problemas e que não haja interrupções no serviço, garantindo assim que a rede, o armazenamento, a plataforma, a capacidade de computação e a postura de segurança estejam todos funcionando corretamente. Se algo der errado, as operações garantem que os incidentes sejam identificados, o pessoal adequado seja alertado, os problemas sejam determinados e as correções sejam aplicadas. Essa etapa corresponde as fases *Operate* e *Observe* da Figura 1

2.1.1.6 Aprendizado

Coleta de feedback de usuários finais e clientes sobre o desempenho e valor da aplicação. Isso também incluiria qualquer aprendizado das fases de operação e monitoramento. Esses

feedbacks são levados para fase de planejamento, reiniciando o ciclo. Essa etapa corresponde à *continuous feedback* e *discover* da Figura 1.

2.1.2 CI/CD

2.1.2.1 Definição

CI/CD é a abreviação de *Continuous Integration/Continuous Delivery*, traduzindo para o português: integração e entrega contínuas. Trata-se de uma prática de desenvolvimento de *software* que visa tornar a integração de código mais eficiente por meio de builds e testes automatizados. Com a abordagem CI/CD é possível entregar aplicações com mais frequência aos clientes.(REDHAT, 2022b)

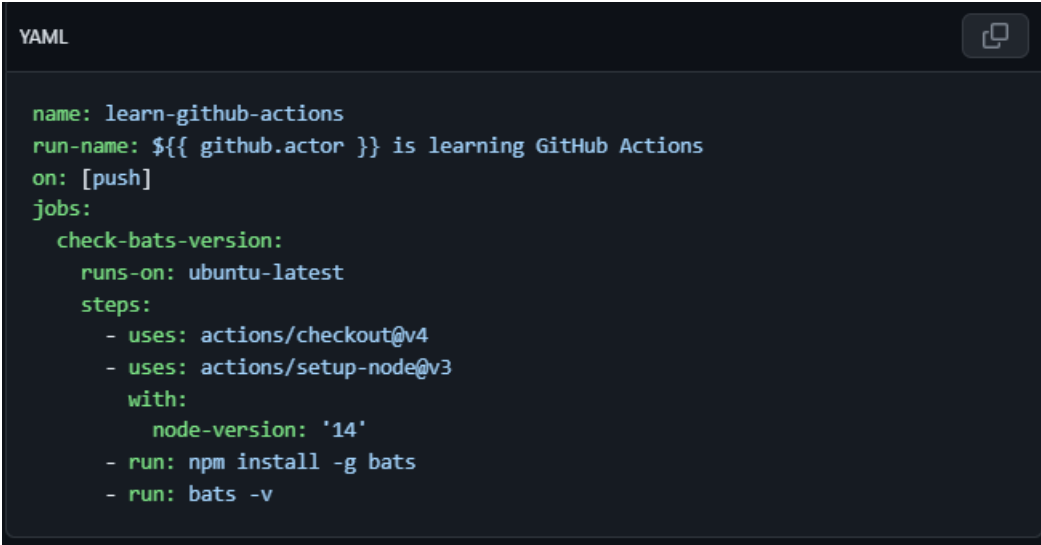
A integração contínua é a ideia de realizar mudanças contínuas no código, sejam novas funcionalidades, correção de *bugs*, ajustes e etc. Também são realizados testes automatizados para verificar se as mudanças realizadas no código não vão ser responsáveis por quebrar a aplicação.

Já a integração contínua aconteceria após as fases da integração contínua onde o objetivo principal é entregar o que foi realizado na etapa anterior. A ideia é de que a base de código esteja sempre pronta para implantação para o ambiente de produção

2.1.2.2 Github Actions

O GitHub Actions é uma plataforma de CI/CD. Nela, é possível criar a *pipeline* que será responsável por automatizar todos os processos de integração e entrega. Ela também permite a criação de fluxos de trabalho que podem rodar fluxos de trabalho diferentes (exemplo: fluxo de produção, desenvolvimento e homologação).

Para criar e executar um fluxo de trabalho é necessário criar um arquivo de configuração onde serão especificados todas as etapas que a sua *pipeline* consistirá. Esses arquivos devem estar no repositório do GitHub, na pasta *.github/workflows*.

Figura 2 – Exemplo de código de um *workflow*

```
name: learn-github-actions
run-name: ${ github.actor } is learning GitHub Actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

Fonte: Elaborada pelo autor.

No arquivo de configuração, é possível definir quais serão os gatilhos de cada fluxo de trabalho. Alguns exemplos de gatilhos são:

- Detectado um *push* em uma *branch*
- Detectado um *pull request* em uma *branch*
- Detectado um *merge* em uma *branch*
- Detectado a criação de uma nova *branch*

Quando o fluxo de trabalho é acionado, ele vai iniciar os *jobs*, que são um conjunto de *actions* que a *pipeline* deve executar. Na Figura 2 o *job* é chamado "*learn-github-actions*" e as ações que ele vai executar são "*actions/checkout@v4*" e "*actions/setup-node@v3*".

2.1.3 Cultura DevOps

O DevOps não se trata somente das tecnologias. As ferramentas serão a base para que a metodologia seja implementada com sucesso. Entendendo os conceitos de DevOps, é possível inferir o que é esperado das equipes. É necessário que as equipes estejam alinhadas com as boas práticas do DevOps para que possam aproveitar o máximo possível do que as ferramentas podem oferecer. Entre as boas práticas é possível destacar:

- Comunicação
- Colaboração
- Responsabilidade compartilhada

2.1.4 DevSecOps

Uma outra definição ou até metodologia (dependendo do autor) é o DevSecOps, que é a ideia de incluir o time de segurança junto ao time de desenvolvimento e operações. DevSecOps é o DevOps que integra e automatiza continuamente a segurança ao longo do ciclo de vida do DevOps, do início ao fim, do planejamento ao feedback e de volta ao planejamento novamente (REDHAT, 2022a).

Vale citar que, algumas definições ainda optam por apenas utilizar o termo DevOps por considerarem que o DevSecOps é o que DevOps deveria ser desde o começo, ou seja, a integração de segurança no fluxo deve ser desde o início. A IBM ainda diz: "O DevSecOps surgiu como um esforço específico para integrar e automatizar a segurança, conforme planejado originalmente. No DevSecOps, a segurança é um cidadão e um *stakeholder* de "primeira classe" junto com o desenvolvimento e as operações, e traz segurança para o processo de desenvolvimento com foco no produto".

2.1.5 AWS

Trabalhar com a nuvem requer que o usuário esteja constantemente atualizado com as novas tecnologias que vão surgindo no mercado devido a natureza da área que permite uma maior velocidade para lançar novos serviços ou atualização dos mesmos. Por isso, nesta sessão, serão revisados os serviços da AWS que serão utilizados neste trabalho para documentar aqueles que foram utilizados durante a realização do projeto.

2.1.5.1 EC2

A EC2 (*Amazon Elastic Compute Cloud*) é o serviço da Amazon que oferece a infraestrutura de computação. Normalmente referidas como instâncias, existem mais de 700 tipos diferentes, seja as opções de processadores, redes, sistemas operacionais e etc.

2.1.5.2 VPC

O VPC (*Virtual Private Cloud*) é o serviço da AWS utilizado para criar ambientes de redes virtuais permitindo um controle sobre tipos de conectividade, posicionamento de recursos na rede e etc.

2.1.5.3 Lambda

O Lambda é o serviço da AWS que permite executar um código somente quando necessário e sem ter que especificar os recursos computacionais necessários para a execução do mesmo. Esse serviço é gerenciado pela AWS, permitindo que o usuário possa focar apenas na funcionalidade do código sem se preocupar com a infraestrutura necessária para execução.

2.1.5.4 SNS

O SNS (*Simple Notification Service*) é um serviço da AWS que envia mensagens por *push* (envia o conteúdo independente do status do receptor). Esse serviço é utilizado normalmente para enviar mensagens/conteúdo gerado por um remetente (seja uma aplicação ou outro serviço) para vários receptores.

2.1.5.5 Internet Gateway

O *Internet Gateway* é um serviço na AWS que permite a comunicação entre a VPC e a Internet com suporte para tráfego IPv4 e IPv6. Permite que recursos em uma sub-rede inicie uma conexão com a Internet e vice-versa.

2.1.5.6 Security Group

O *Security Group* é o *firewall* virtual para as instâncias EC2. Dessa forma é possível realizar o controle de tráfego de entrada e saída da instância.

2.2 Infraestrutura como código

2.2.1 Definição

A infraestrutura como código (IaC) é uma metodologia de criar, configurar e provisionar toda uma infraestrutura de maneira automatizada e programática. Ao invés de configurar todos os serviços manualmente na provedora de nuvem, é criado um arquivo de configuração onde é codificado os recursos que serão provisionados assim como a configuração de cada um desses serviços. Assim, garante-se consistência e uma maior velocidade na disponibilização da infraestrutura. Outra vantagem de usar IaC, é a possibilidade de versionamento da infraestrutura que, por ter sido codificada, é possível usar ferramentas de versionamento de código (como o Github) para ter esse benefício extra.

Figura 3 – Exemplo 1 de IaC - Terraform

```
terraform {  
  required_providers {  
    aws = {  
      source = "hashicorp/aws"  
      version = "~> 4.16"  
    }  
  }  
  
  required_version = ">= 1.2.0"  
}  
  
provider "aws" {  
  region = "us-west-2"  
}  
  
resource "aws_instance" "app_server" {  
  ami = "ami-830c94e3"  
  instance_type = "t2.micro"  
  
  tags = {  
    Name = "ExampleAppServerInstance"  
  }  
}
```

Fonte: Elaborada pelo autor.

Figura 4 – Exemplo 2 - CloudFormation

```

{
  "Resources": {
    "Ec2Instance": {
      "Type": "AWS::EC2::Instance",
      "Properties": {
        "SecurityGroups": [
          {
            "Ref": "InstanceSecurityGroup"
          }
        ],
        "KeyName": "mykey",
        "ImageId": ""
      }
    },
    "InstanceSecurityGroup": {
      "Type": "AWS::EC2::SecurityGroup",
      "Properties": {
        "GroupDescription": "Enable SSH access via port 22",
        "SecurityGroupIngress": [
          {
            "IpProtocol": "tcp",
            "FromPort": 22,
            "ToPort": 22,
            "CidrIp": "0.0.0.0/0"
          }
        ]
      }
    }
  }
}

```

Fonte: Elaborada pelo autor.

2.2.2 Terraform

O HashiCorp Terraform é uma ferramenta de infraestrutura como código que permite que definir os recursos em nuvem e locais em arquivos de configuração legíveis por humanos que podem ser versionados, reutilizados e compartilhados (HASHICORP, 2022b). Ele também permite integrar o provisionamento de infraestrutura na sua *pipeline* auxiliando também no gerenciamento da mesma. É utilizado a linguagem HCL (*HashiCorp Configuration Language*),

como é possível ver na Figura 3.

Quando é executado um arquivo do Terraform ele cria um arquivo de estado que, vai salvar em forma de código toda a infraestrutura que foi provisionada por ele. Dessa forma, ele consegue verificar as diferenças durante a execução desse arquivo, e fazer apenas as alterações necessárias (sem ter que derrubar e provisionar tudo novamente).

```
{
  "version": 4,
  "terraform_version": "1.2.3",
  "serial": 1,
  "lineage": "a2d45b6b-71b5-4760-8a83-ae8b86ca5c84",
  "outputs": {},
  "resources": [
    {
      "module": "example_module",
      "mode": "managed",
      "type": "aws_instance",
      "name": "example_instance",
      "provider": "provider.aws",
      "instances": [
        {
          "schema_version": 0,
          "attributes": {
            "ami": "ami-12345678",
            "instance_type": "t2.micro",
            "subnet_id": "subnet-abcdefg",
            "tags": {
              "Name": "ExampleInstance"
            },
            "vpc_security_group_ids": [
              "sg-12345678"
            ]
          }
        }
      ]
    }
  ]
}
```

Figura 5 – Exemplo de um arquivo de estado

2.2.3 Terraform Cloud

O Terraform Cloud é uma aplicação que ajuda equipes a usar o Terraform de forma colaborativa. Ele gerencia as execuções do Terraform em um ambiente consistente e confiável e inclui acesso fácil a estados compartilhados e dados secretos, controles de acesso para aprovar

mudanças na infraestrutura, um registro privado para compartilhar módulos do Terraform, controles de políticas detalhadas para governar o conteúdo das configurações do Terraform e muito mais (HASHICORP, 2022a).

Figura 6 – Exemplo da console do Terraform Cloud

WORKSPACE NAME	RUN STATUS	LATEST CHANGE	RUN	REPO
exceed-limit	✓ APPLIED	5 months ago	run-BBAc	NICKF/terraform-minimum
filetest-dev	✗ ERRORED	3 months ago	run-SLSz	nfagerlund/terraform-filetest
migrated-default	✓ PLANNED	5 months ago	run-BVyj	nfagerlund/terraform-minimum
migrated-first	✓ PLANNED	5 months ago	run-A2sp	nfagerlund/terraform-minimum
migrated-second	✓ PLANNED	5 months ago	run-KqNV	nfagerlund/terraform-minimum
migrated-solo	✓ APPLIED	5 months ago	run-1RkX	NICKF/terraform-minimum
migrated-solo2	✓ PLANNED	5 months ago	run-Rlh7	nfagerlund/terraform-minimum
migrate-first-2	! NEEDS CONFIRMATION	3 months ago	run-hR57	nfagerlund/terraform-minimum

Fonte: (HASHICORP, 2022a)

Uma importante funcionalidade do Terraform Cloud é que ele permite que o usuário armazene o arquivo de estado das execuções do arquivo de Terraform. Dessa maneira, mesmo provisionando a infraestrutura em máquinas diferentes, é possível comparar os arquivos de estado das execuções e assim, evita-se problemas como perder a referência de estado da infraestrutura, duplicação, conflito de versões e etc.

2.2.4 Checkov

O *Checkov* é uma ferramenta que escaneia arquivos de infraestrutura de *cloud* para encontrar má configurações antes que essas infraestruturas sejam implantadas. O Checkov consegue escanear os seguintes tipos de *IaC*:

- Terraform
- CloudFormation
- Azure Resource Manager
- Serverless framework
- Helm charts
- Kubernetes

- Docker

O *Checkov* também permite que o usuário crie políticas personalizadas, dando mais liberdade para fazer as verificações necessárias no seu *IaC*.

Outro benefício dessa ferramenta é que ele tem integração nativa com o *GitHub Actions* permitindo que o usuário integre essa ferramenta mais facilmente na sua *pipeline* criada no *GitHub*.

2.3 Cloud Security Posture Management

2.3.1 Definição

O *CSPM* (em português, Gerenciamento de Postura de Segurança na Nuvem), é uma ferramenta que faz verificações de postura e segurança dos recursos na sua *cloud* baseado em *frameworks* de segurança.

As principais funções dessa tecnologia é trazer visibilidade, segurança e governança ao ambiente na nuvem. O *CSPM* vai conseguir listar para o usuário todos os recursos que estão implantados na nuvem, assim trazendo visibilidade aos times de arquitetura e segurança dos possíveis recursos que podem apresentar riscos de segurança por não existir uma supervisão em cima dele. A ferramenta também gera alerta quando detecta serviços que estão mal configurados que apresentam um risco de segurança para o ambiente. O *CSPM* traz recomendações de segurança baseadas nos *frameworks* de segurança sobre a melhor prática de como deveriam ser implantados os recursos na nuvem de maneira segura. E por fim, ainda permite que o usuário configure as regras com base nas demandas regulatórias que devem ser seguidas. Por exemplo, no Brasil, a maioria das empresas que trabalham com dados pessoais deve seguir a LGPD.

2.3.2 Conformity

O *Conformity* é o *CSPM* da empresa *Trend Micro*. Esse módulo visa especificar todas as capacidades dessa ferramenta uma vez que ela será utilizada no trabalho para fazer o gerenciamento de riscos de segurança no nosso ambiente na nuvem.

2.3.2.1 Funcionamento

O *Conformity* acessa nosso ambiente na nuvem com uma permissão de leitura que dará para a ferramenta a possibilidade de ver todos os recursos que existem na conta. Assim, usando a base de regras da *Trend Micro*, ele fará verificações que irão gerar uma nota de conformidade. Cada uma dessas verificações, na plataforma, é denominada como *check*. Vale citar que o *Conformity* não faz apenas verificações de segurança. Alguns exemplos desses *checks* (nomenclatura utilizada pela ferramenta):

- Portas de rede com acesso irrestrito (22, 3389, etc)
- Usuários com privilégio mínimo
- Usuários com várias chaves de acesso para *cloud*
- *Bucket* S3 público

Como ele tem uma forte base nos seis pilares do *AWS Well-Architected Framework*, ele também faz verificações de otimização de custo, confiabilidade, sustentabilidade, excelência operacional e eficiência de performance.

2.3.2.2 Integrações

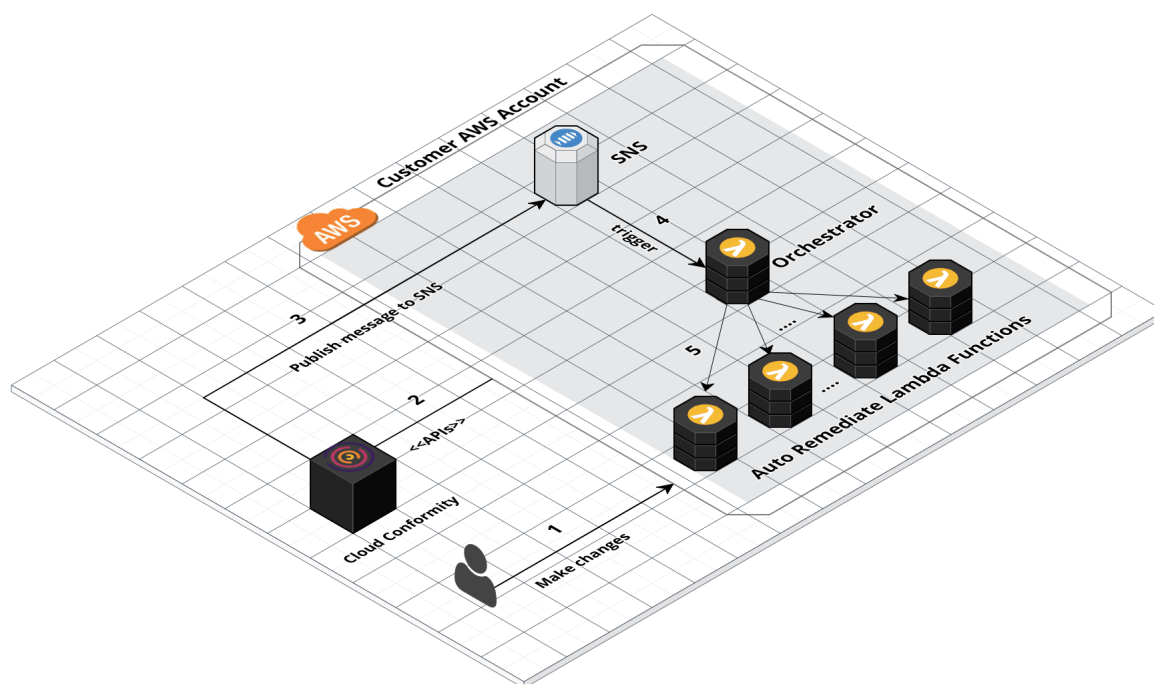
É importante que a ferramenta que será integrada na *pipeline* tenha capacidade de interagir com outras tecnologias. O contrário faz com que a ferramenta fique isolada e não contribua para o fluxo da aplicação. O *Conformity* tem diversas integrações nativas que colaboram para a mais fácil integração na *pipeline*:

- PagerDuty: ferramenta de gerenciamento de incidents
- Jira: ferramenta de gerenciamento de projetos
- Zendesk: ferramenta focada no atendimento do cliente (por meio de tickets, por exemplo)
- ServiceNow: *ITSM*
- Amazon SNS: serviço da Amazon utilizado para envio de mensagens
- Webhook: método de comunicação de sistemas

2.3.2.3 Auto Remediação

No contexto do *DevOps*, o uso de automações é um dos principais atrativos da metodologia já que a automação acelera processos repetitivos e permite que os recursos humanos sejam alocados para tarefas mais exigentes. O *Auto Remediation* é uma função do *Conformity* que ao detectar um erro na *cloud* vai automaticamente remediar o erro. Para que essa função seja ativada, a ferramenta fornece um repositório no *GitHub* que vai implantar a estrutura necessária para que a auto remediação funcione.

Figura 7 – Desenho dos recursos na *Cloud* para o funcionamento da auto remediação



Fonte: citar Trend

Na Figura 7 é possível ver a arquitetura dos serviços da AWS que serão implantados na conta. O funcionamento dessa funcionalidade segue da seguinte forma:

1. Quando é detectada uma nova falha, ela é enviada para o tópico SNS
2. O SNS envia essa mensagem para o Lambda orquestrador
3. Caso exista e esteja ativada a regra de remediação dessa falha, é acionado o próximo Lambda que realizará a remediação

3 Metodologia

3.1 Aplicativo

O objetivo final do trabalho é criar uma *pipeline* integrando serviços de segurança de nuvem. Por conta disso, foi desenvolvido uma aplicação simples para que o fluxo esteja “vivo” e não exista somente a infraestrutura na nuvem.

3.1.1 Discord

O Discord é uma plataforma de comunicação que permite bate-papo por texto, voz e vídeo entre os usuários. A plataforma permite a criação de grupos onde os usuários têm acesso a diferentes canais onde podem interagir com os outros membros da comunidade. Um dos principais atrativos da plataforma são os *bots*. O Discord disponibiliza inúmeros *bots*, criados pela própria comunidade, que tem as mais diferentes funções, como: *bots* que ajudam no gerenciamento de pessoas, tocam música e etc.

3.1.2 ShotBot

A aplicação criada para esse trabalho é um *bot* de Discord feito em Python apelidada como ShotBot. A principal função do ShotBot é salvar frases memoráveis ditas pelos membros de um grupo e listar as mesmas quando solicitada pelo usuário. Como a aplicação não é o ponto principal da tese, esse módulo não abordará detalhes técnicos de como a aplicação foi feita, porém alguns pontos sobre os pré-requisitos para o funcionamento da aplicação serão abordados no módulo de arquitetura.

Lista de comandos:

- `!help`: lista os comandos disponíveis e suas funcionalidades para o usuário
- `!shot @nome "frase"`: salva a frase para a lista de frases do usuário
- `!list @nome`: lista todas as frases memoráveis dita pelo usuário
- `!ping`: comando utilizado para testar se o *bot* está ativo (retorna "pong" como mensagem)

Figura 8 – Exemplo de comando do ShotBot no *Discord*

Fonte: Elaborada pelo autor.

Caso o leitor tenha interesse, a documentação do ShotBot está no repositório que está referenciado no final do trabalho.

Figura 9 – Trecho do código do ShotBot

```
@abstractmethod
def save(self):
    pass

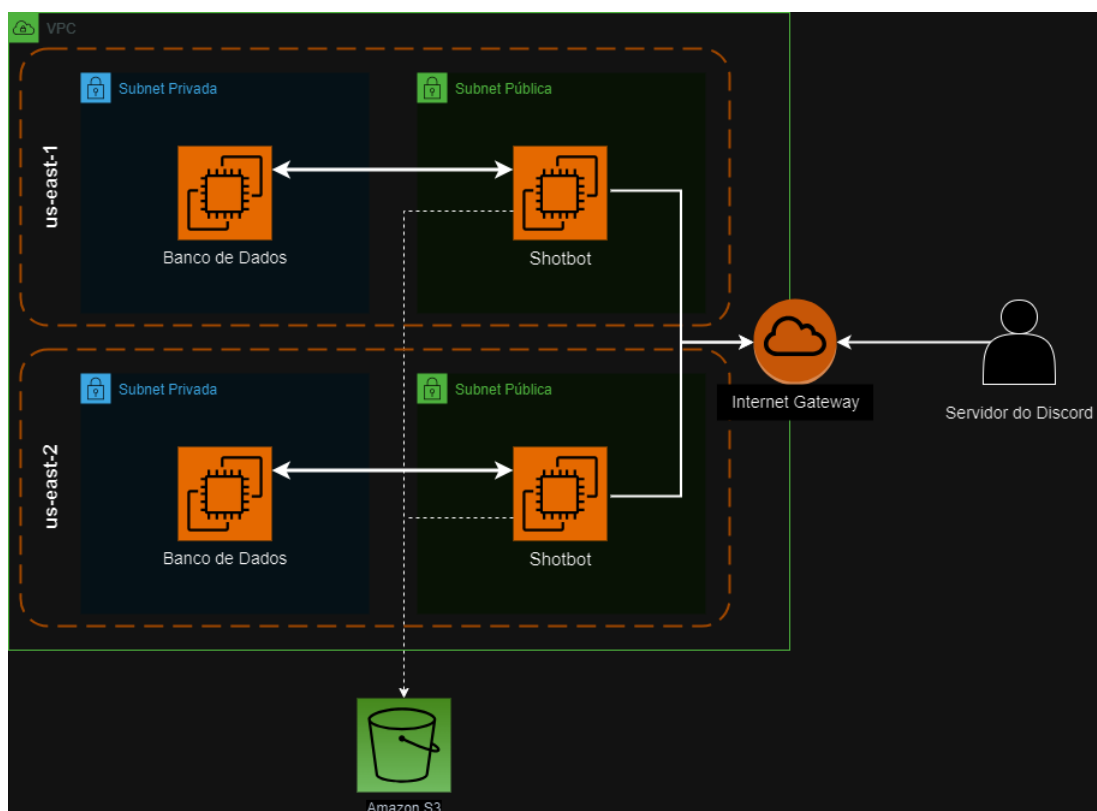
@classmethod
def load(cls, id):
    cls.lock.acquire()
    try:
        cur = cls.conn.cursor()
        res = cur.execute('SELECT * FROM {0} WHERE id = {1} LIMIT 1'.format(cls.table_name, id))
        row = res.fetchone()
        if row is not None:
            return cls.convert_row(row)
        else:
            return None
    finally:
        cls.lock.release()

@classmethod
def load_all(cls):
    cls.lock.acquire()
    try:
        cur = cls.conn.cursor()
        res = cur.execute('SELECT * FROM {0} LIMIT 1000'.format(cls.table_name, id))
        rows = res.fetchall()
        return [cls.convert_row(row) for row in rows]
    finally:
        cls.lock.release()
```

Fonte: Elaborada pelo autor.

3.2 Arquitetura da aplicação

Figura 10 – Desenho da arquitetura da aplicação



Fonte: Elaborada pelo autor.

A aplicação é executada em uma instância EC2 localizada em uma *subnet* pública dentro da VPC ShotBot-VPC. Essa mesma máquina tem acesso à uma *Internet Gateway* que permite que o ShotBot receba as requisições do usuário que são enviadas pelo servidor do Discord. Para o funcionamento dessa comunicação, é necessário configurar as permissões de entrada e saída no *security group* associado à essa instância. Como a comunicação com o servidor do Discord é realizado seguindo o protocolo HTTPS, a porta 443 foi configurada para viabilizar a conexão.

A aplicação também tem acesso a um banco de dados MySQL localizado em uma instância em uma *subnet* privada que vai conter dados dos usuários. Então foi configurado a porta 3306 para as duas instâncias para que a comunicação entre elas fosse possível. Além disso, a instância rodando o banco de dados só permite conexões de IPs do VPC ShotBot-VPC.

O ShotBot também tem acesso a um *bucket* S3 já que a aplicação também faz uso de imagens que são submetidas pelos usuários. Para isso, foi criado uma política IAM (serviço utilizado para fazer gestão de acessos na AWS) que garante o acesso apenas ao *bucket* designado para o uso do ShotBot.

Vale comentar que a arquitetura proposta não foi realizada seguindo todas as melhores

práticas uma vez que, com o CSPM for implantado, é necessário que existam más configurações no ambiente para gerar evidências da ferramenta para o trabalho. Um exemplo de má configuração seria que essa configuração não está com o *auto scale* ativado. Caso uma grande quantidade de *request* ocorra para os servidores, é possível que os mesmos sejam sobrecarregados e a aplicação fique fora de serviço. Com o *auto scale* as máquinas seriam provisionadas baseadas no uso, o que permitira uma maior flexibilidade para a aplicação.

3.2.1 Terraform

Depois de arquitetar toda o projeto, foi criado um arquivo de configuração em Terraform para a infraestrutura do ShotBot. Com isso é possível garantir consistência e rapidez na hora implantar e manter a infraestrutura necessária para o trabalho. Além disso, o Terraform permite criar automações que serão integrados na *pipeline* nas próximas etapas.

Figura 11 – Trecho do terraform para o ShotBot

```
# Configure the AWS Provider
provider "aws" {
  region = "us-east-1"
}

# Create a VPC
resource "aws_vpc" "shotBot_vpc" {
  cidr_block      = "10.0.0.0/16"
  instance_tenancy = "default"

  tags = {
    Name = "ShotBot-vpc"
  }
}

# Create public subnet
resource "aws_subnet" "public_subnet" {
  vpc_id            = aws_vpc.shotBot_vpc.id
  cidr_block        = "10.0.1.0/24"
  availability_zone = "us-east-1b"

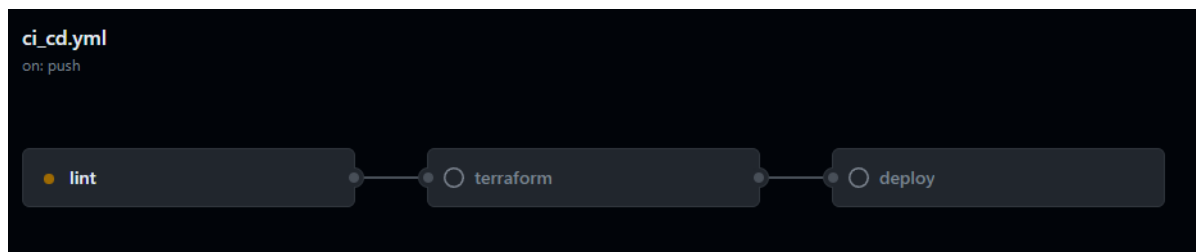
  tags = {
    Name = "public-subnet"
  }
}
```

Fonte: Elaborada pelo autor.

3.3 GitHub Actions

O GitHub Actions foi a plataforma escolhida para criar e gerenciar a *pipeline* para este trabalho. O fluxo foi dividido em três etapas: *lint*, *terraform* e *deploy*.

Figura 12 – Pipeline construída para o trabalho



Fonte: Elaborada pelo autor.

3.3.1 Lint

Esse *job* é responsável por fazer verificações nos arquivos de código do repositório para fazer verificações de sintaxe, segurança e complexidade do código.

Nesse *job* foi utilizada uma *action* disponível no *MarketPlace* do GitHub conhecida como Black. Essa ação fará verificações de sintaxe e formatação em todos os arquivos de código no repositório do projeto.

Figura 13 – Trecho do código responsável pelo *lint*

```
jobs:
  lint:
    runs-on: ubuntu-latest
    permissions:
      checks: write
      contents: write

    steps:
      - name: Check out Git repository
        uses: actions/checkout@v2

      - name: Set up Python 3.8
        uses: actions/setup-python@v2
        with:
          python-version: 3.8

      - name: Install black
        run: pip install black

      - name: Run linters
        uses: wearerequired/lint-action@v2
        with:
          black: true
```

Fonte: Elaborada pelo autor.

3.3.2 Terraform

Esse *job* é responsável por executar o arquivo de Terraform. Inicia-se esse fluxo clonando o repositório localmente e em seguida instalando todas as dependências necessárias para rodar o Terraform. Depois, é feita uma verificação no Terraform Cloud para comparar os arquivos de estado do repositório local e do que está salvo no Terraform Cloud. Caso existam diferenças, é provisionado ou derrubado os serviços que estão salvos no arquivo de estado novo e, caso contrário, não é realizada nenhuma ação.

Vale citar que foi criado também um *job* chamado *destroy* que é um fluxo que irá derrubar todos os serviços criados por essa *pipeline*. Porém, esse *job* não está integrado no fluxo da aplicação pois entende-se que, como a aplicação estará disponível o tempo inteiro, não há necessidade de destruir a infraestrutura. Esse *job* foi criado preventivamente para caso seja necessário derrubar a infraestrutura.

Figura 14 – Trecho do código responsável pelo Checkov e pelo Terraform

```
- name: install Checkov
  run: pip install checkov

- name: Run Checkov
  run: checkov -d .

- name: Set up Terraform
  uses: hashicorp/setup-terraform@v2
  with:
    cli_config_credentials_hostname: 'app.terraform.io'
    cli_config_credentials_token: {{ secrets.TFE_TOKEN }}

- name: Terraform Init
  run: terraform init

- name: Terraform Apply
  run: terraform apply -auto-approve

- name: Upload Terraform State
  uses: actions/upload-artifact@v2
  with:
    name: terraform-state
    path: terraform.tfstate
```

Fonte: Elaborada pelo autor.

3.3.3 Deploy

E, por fim, essa etapa irá implantar o código novo na infraestrutura na *cloud*. Usando ações do *GitHub Actions*, a *runner* irá utilizar do *AWS CLI* para se conectar a instância usando o serviço *SSM*. Com isso, ela vai usar *scripts* que foram criados para encerrar o *ShotBot*, atualizar o código e, enfim, rodar o código novamente.

Figura 15 – Trecho do código responsável por fazer o deploy do código

```
deploy:
  needs: terraform
  runs-on: ubuntu-latest

  steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up AWS CLI
      uses: aws-actions/configure-aws-credentials@v1
      with:
        aws-access-key-id: ${ secrets.AWS_ACCESS_KEY_ID }
        aws-secret-access-key: ${ secrets.AWS_SECRET_ACCESS_KEY }
        aws-region: us-east-1
```

Fonte: Elaborada pelo autor.

3.4 Ferramentas de segurança

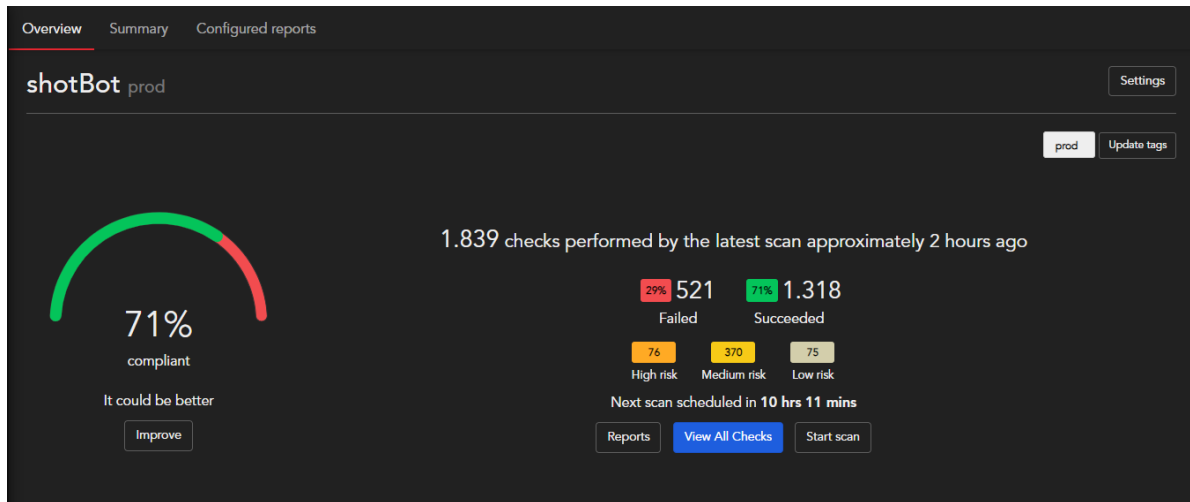
3.4.1 Conformity

Com a *pipeline* criada, como que o Conformity conseguiria se integrar a ela? Existem diversas funcionalidades do CSPM que vão permitir que a ferramenta converse com o resto das tecnologias.

3.4.2 Alertas

Os alertas gerados pelo CSPM podem ser enviados de forma automática ou manual para fora da plataforma por meio das integrações nativas que a ferramenta possui, como e-mail, JIRA, Microsoft Teams e outros.

Figura 16 – Porcentagem de conformidade da conta hospedando o ShotBot



Fonte: Elaborada pelo autor.

Figura 17 – Tela de alertas detectados pelo Conformity

View by Rule		View by Resource				
Rule	Service	Categories	Risk level	Counts		
+ AWS Config Enabled Updated	Config	Security, Sustainability	High	FAILURE: 17	Resolve...	
+ CloudTrail Enabled Updated	CloudTrail	Security, Sustainability	High	FAILURE: 17	Resolve...	
+ EBS Encrypted	EBS	Security	High	FAILURE: 6	Resolve...	
+ AWS IAM Users with Admin Privileges	IAM	Security	High	FAILURE: 5	Resolve...	
+ Check for IAM User Group Membership	IAM	Security	High	FAILURE: 5	Resolve...	
+ Access Keys Rotated 90 Days	IAM	Security	High	FAILURE: 4	Resolve...	

Fonte: Elaborada pelo autor.

Quando é contextualizado essa funcionalidade no DevOps, essa ação representaria uma das últimas etapas do ciclo que é o feedback e a descoberta (figura 1). Com o alerta gerado, as equipes de arquitetura e segurança devem planejar os próximos passos para tratar as vulnerabilidades de segurança apontadas pela ferramenta que daria início a um novo ciclo de desenvolvimento.

3.4.3 Auto Remediação

A auto-remediação é uma função do Conformity que deve ser implantada no ambiente porém não de imediato. Como a auto-remediação implica que a ferramenta terá acesso de escrita na nuvem, é essencial que as equipes responsáveis tenham um entendimento completo do ambiente para que os serviços que estão rodando na *cloud* não se tornem indisponíveis caso o CSPM faça uma alteração.

Para ativar essa *feature* é necessário seguir as seguintes etapas:

1. Fazer *deploy* do *stack* usando a ferramenta provisionada pelo *Conformity* que será responsável por criar toda a infraestrutura mencionada no Capítulo 2.
2. Alterar as regras que serão ativadas na conta
3. Configurar o canal de comunicação que enviará os erros para o tópico SNS que foi criado na etapa 1.
4. Configurar uma chave KMS e permitir que a conta do *Conformity* consiga usá-la para acessar o tópico SNS

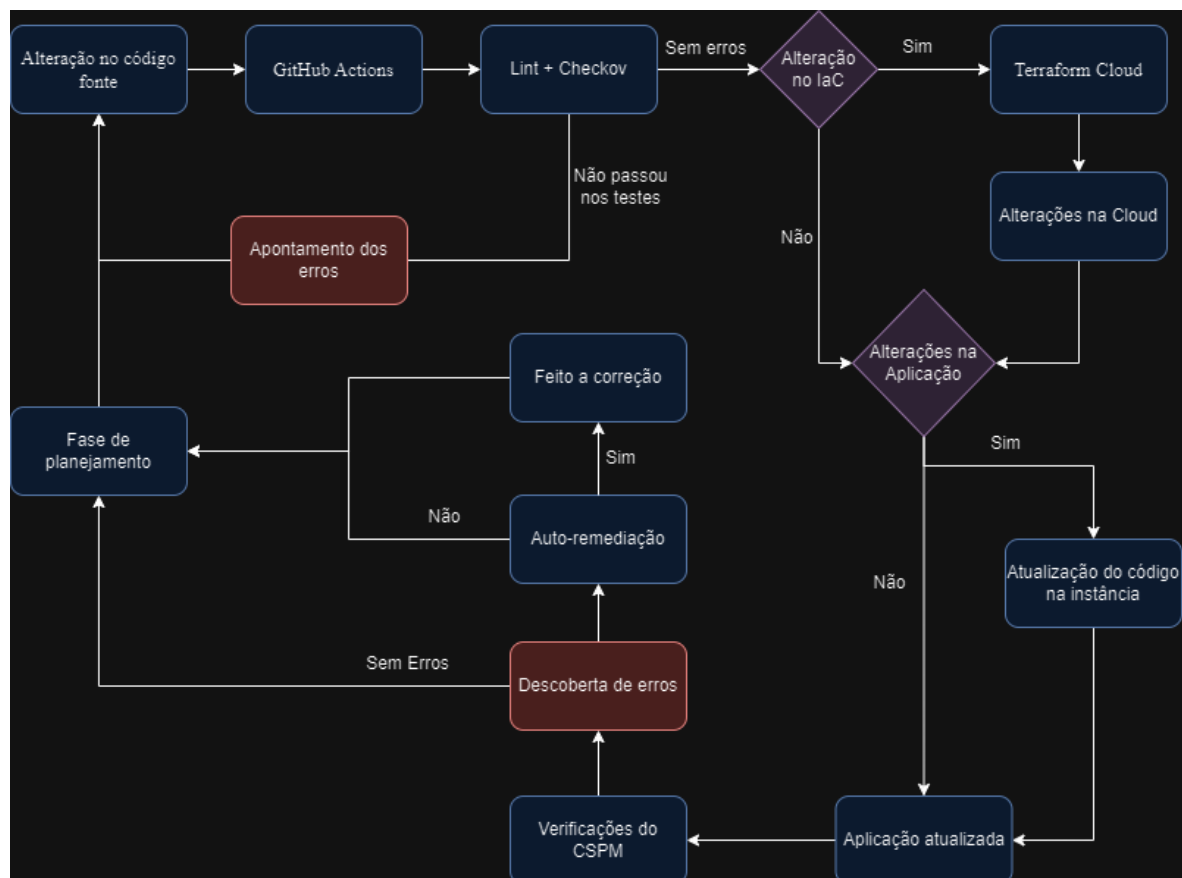
Todas essas etapas podem ser vistas com mais detalhes na documentação providenciada pela ferramenta:]

4 Resultados

4.1 Pipeline

Essa sessão vai focar em mostrar como todas as ferramentas que foram apresentadas e implementadas funcionam em conjunto.

Figura 18 – Diagrama do Pipeline



Fonte: Elaborada pelo autor.

A Figura 18 é um desenho do fluxo que foi montado para o trabalho. Assim que é realizado alguma mudança no repositório do projeto, o *GitHub Actions* irá dar início a *pipeline*.

Inicia-se a etapa "*Lint*" (usar como referência Figura 12), que fará verificações de sintaxe e melhores práticas no código do trabalho. Se detectado algum erro de formatação, essa etapa é responsável por fazer as correções para que seja mantido um padrão de formatação no código do projeto.

Figura 19 – Exemplo caso existam erros pegos pelo Black

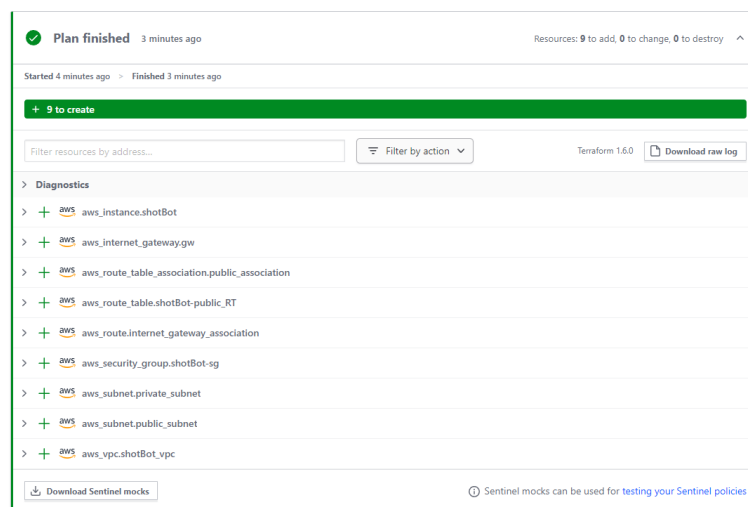
```
1 ▶ Run wearerequired/lint-action@v2
82 SHA of last commit is "f6f542e150eb0a25a7978e68c7b15df1de386045"
83 ▼ Run Black
84   Verifying setup for Black...
85   Verified Black setup
86   Will use Black to check the files with extensions py
87   Linting files in /home/runner/work/shotBot/shotBot with Black ...
88   would reformat /home/runner/work/shotBot/shotBot/shotBot.py
89   would reformat /home/runner/work/shotBot/shotBot/dal.py
90
91   Oh no! ✖️ 💔 ✖️
92   2 files would be reformatted.
93   Black found 10 errors (failure)
94 ▶ Create check runs with commit annotations
```

Fonte: Elaborada pelo autor.

Em seguida a ferramenta *Checkov* irá realizar as verificações de segurança no *Terraform* que é responsável por provisionar nossa infraestrutura (*IaC*). Se ele detectar alguma falha de segurança de alto risco, ele irá parar o fluxo, apontar o erro e criar uma tarefa apontando a correção que deve ser realizada.

Passando por essa fase inicial dos testes, será feita uma verificação para ver se os arquivos do *Terraform* sofreram alguma alteração. Se não houver nenhuma mudança, o fluxo continua. Mas caso seja detectado alguma alteração, começam as etapas do *Terraform* (descritas com mais detalhes nos capítulos anteriores), onde será feita a comparação do arquivo de estado no *Terraform Cloud*. Após essa etapa, são realizadas as alterações na *Cloud*.

Figura 20 – Exemplo na console do Terraform Cloud sobre a criação de recursos na cloud



Fonte: Elaborada pelo autor.

A próxima etapa foca mais na aplicação que está rodando na nuvem. É feita uma verificação nos arquivos responsáveis pelo *ShotBot* e caso sejam detectados mudanças, o *GitHub Actions* irá se conectar na instância responsável por rodar a aplicação por meio do *System Manager*, onde ele conseguirá executar os *scripts* responsáveis por parar a aplicação, atualizar o código e processá-lo novamente.

Realizada as alterações submetidas pelo usuário, o *CSPM* agora irá ajudar a fazer o gerenciamento dos riscos de segurança do ambiente. Com a ajuda do *Conformity*, é possível verificar todos os apontamentos que a ferramenta realiza e fazer o planejamento para corrigir ou não as indicações do *CSPM*. Também será possível realizar a correção automática dos erros que já estiverem mapeados para fazer parte dessa automação.

4.2 Testes no Checkov

A principal função do *Checkov* nesse fluxo, é de detectar erros de alto risco no *Terraform* e impedir que tais mudanças subam para produção. Por conta disso, o teste escolhido nessa etapa foi de alterar as portas abertas no *Security Group* do *ShotBot*. A porta escolhida para o teste foi a porta 22 (porta comumente usada para fazer conexões SSH) com acesso irrestrito (sem restrição de IPs) que é considerada uma configuração de alto risco.

Figura 21 – Configuração no Terraform permitindo acesso irrestrito na porta 22

```

07
08   ingress {
09     from_port = 22
10     to_port   = 22
11     protocol  = "tcp"
12     cidr_blocks = ["0.0.0.0/0"]
13   }
14
15   egress {

```

Fonte: Elaborada pelo autor.

Assim que o repositório no *GitHub* foi atualizado com essa nova configuração, inicia-se o fluxo de trabalho onde o *Checkov* fará a detecção e bloqueará a mudança:

Figura 22 – Checkov bloqueando a configuração que foi submetida

```

144 Check: CKV_AWS_24: "Ensure no security groups allow ingress from 0.0.0.0:0 to port 22"
145   FAILED for resource: aws_security_group.shotBot-sg
146   File: /main.tf:95-126
147   Guide: https://docs.paloaltonetworks.com/content/techdocs/en\_US/prisma/prisma-cl
148
149       95 | resource "aws_security_group" "shotBot-sg" {
150         96 |   name           = "allow_all_ingress"
151         97 |   description    = "Allow ingress on all ports - test"
152         98 |   vpc_id         = aws_vpc.shotBot_vpc.id
153         99 |

```

Fonte: Elaborada pelo autor.

Em seguida ele irá abrir um *card* no *Jira* apontando o erro encontrado durante a *pipeline*.

4.3 Testes no Conformity

4.3.1 Fluxo natural

O *CSPM* vai apontar para o usuário quais são as principais não conformidades na conta dele. Porém uma ferramenta que apenas aponta para os erros na *cloud* não é o suficiente para que a integração faça sentido pelo o que foi proposto neste trabalho. A principal vantagem é a integração que o *CSPM* tem com outras plataformas, que nesse caso foi o *Jira*, que permite que o usuário escolha os erros que deseja tratar e, no próprio *Conformity*, consiga criar cards quadro de tarefas do *Jira* para a fase de planejamento.

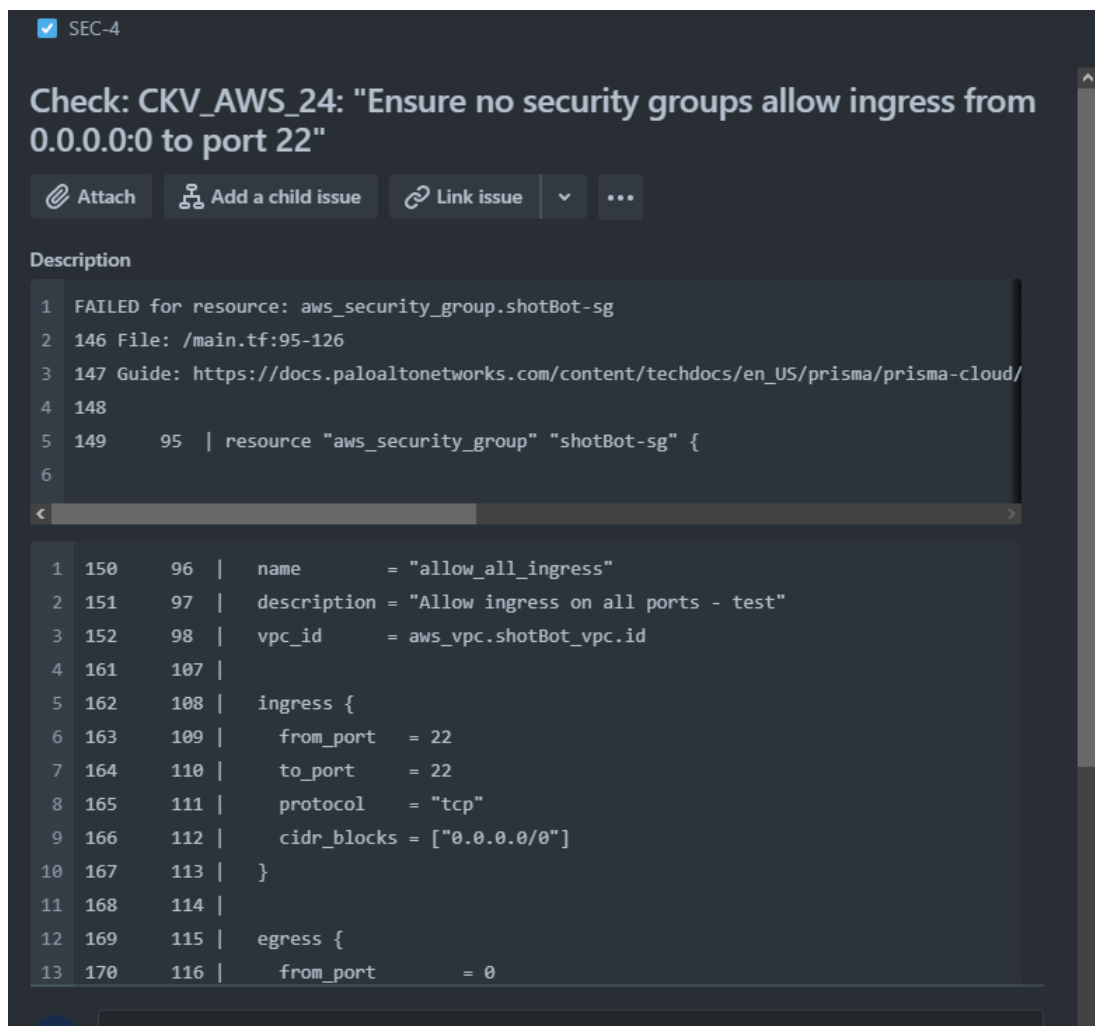
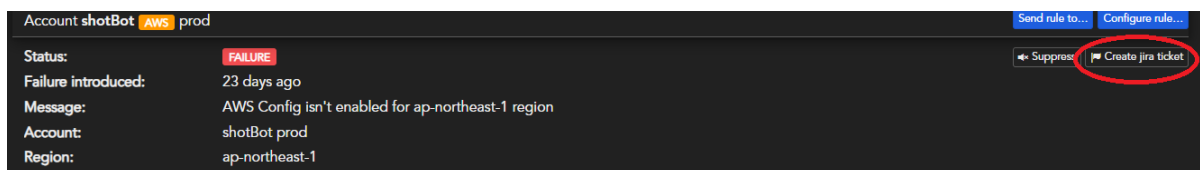


Figura 23 – Card criado no Jira mostrando a falha detectada durante o fluxo

Figura 24 – Botão de envio para o Jira



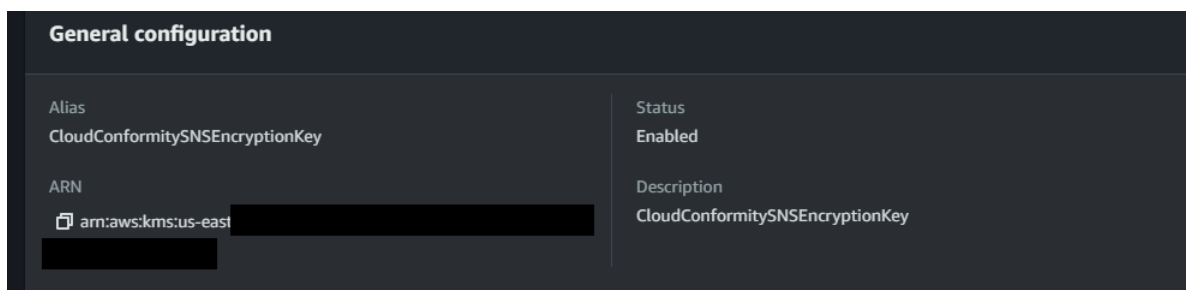
Fonte: Elaborada pelo autor.

4.3.2 Auto-Remediação

O *CSPM* é a ferramenta que verificará a postura do nosso ambiente em nuvem constantemente. É importante entender que o *Checkov* e o *Conformity* tem uma base de regras diferentes que, apesar de muitas serem parecidas, a cobertura de cada uma varia um pouco. As duas ferramentas têm a mesma cobertura de regras para as condições mais tradicionais, como: portas de rede abertas sem restrição, *buckets* públicos, falta de MFA e outros. Por isso, na *pipeline* construída, o *Checkov* acaba funcionando como uma defesa na linha de frente. Espera-se que nenhum desses erros que as duas ferramentas cobrem cheguem no *Conformity*. Mas mesmo que o *Checkov* tenha impedido que o usuário adicionasse ou criasse configurações de alto risco não significa que o ambiente não possua nenhuma outra configuração de grave risco. Logo, entende-se que integrar ferramentas de segurança que usam diferentes fontes de conhecimento nos ajudam a melhorar a área de cobertura do ambiente na nuvem.

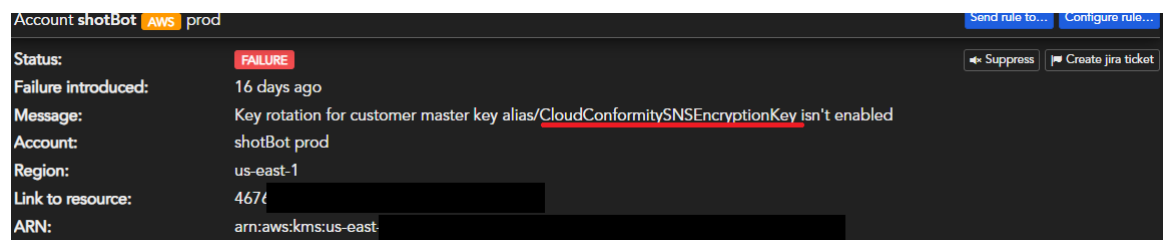
Para o teste no *Conformity* foi escolhida a regra "*Key Rotation Enabled*" que é uma regra que verifica se as chaves *KMS* da conta estão com a configuração de rotatividade ativada uma vez que entende-se que é uma boa prática de segurança trocar a chave após um determinado período de tempo para garantir que, caso a chave tenha vazado, o acesso garantido pela chave não seja mais possível.

Figura 25 – Criação da chave KMS



Fonte: Elaborada pelo autor.

Figura 26 – Conformity apontando falha na regra de rotatividade



Fonte: Elaborada pelo autor.

Conforme foi explicado em capítulos anteriores, após a detecção do erro pelo *CSPM*, isso irá iniciar o fluxo de remediação que, após a tratativa, deve atualizar o *check* na plataforma:

Figura 27 – Após um tempo com a correção feita

Status:	SUCCESS
Failure resolved:	6 minutes ago
Message:	Key rotation for customer master key alias/CloudConformitySNSEncryptionKey is enabled
Account:	shotBot prod
Region:	us-east-1
Link to resource:	
ARN:	arn:aws:kms:us-east-

Fonte: Elaborada pelo autor.

Como foi possível observar, a principal vantagem que uma automação dessa traz é que seria necessário uma equipe gastar um ciclo inteiro de desenvolvimento para realizar a tratativa deste erro. Uma vez que foi determinada que a correção desse erro não tem impacto negativo no ambiente, é possível alocar esforços para realizar outras tarefas prioritárias.

5 Conclusão

O objetivo do trabalho, que foi atingido, era de criar uma *pipeline* que integrasse ferramentas de segurança de gerenciamento de riscos em ambientes na nuvem seguindo as melhores práticas do DevOps.

Para isso, foi criada uma *pipeline* inicial capaz de provisionar um ambiente na nuvem, implantar a aplicação na infraestrutura e entregar a aplicação. Com a base do trabalho criada, foram implementadas as duas ferramentas de segurança focadas em gerenciamento de segurança na *cloud* de maneira que elas conversassem com as outras tecnologias que foram utilizadas na *pipeline*.

O *Checkov* é uma ferramenta importante que evita que más configurações sejam implementadas no código *laC*, evitando que infraestruturas mal configuradas sejam implementadas na nuvem. Dessa forma, foi possível adicionar uma camada de segurança antes de expor o ambiente da aplicação.

Uma observação importante é de que a função principal dessa ferramenta é de fazer o bloqueio, algo que foi citado como uma desvantagem neste trabalho já que, conceitualmente, seria contraditório ao DevOps. Porém no fluxo criado, quando ocorre um bloqueio, serão criadas tarefas no *board* do Jira do usuário que vão conter informações sobre a razão da interrupção no fluxo para agilizar o trabalho do desenvolvedor na hora de remediar o erro. Dessa forma, ao invés de apenas interromper o ciclo de vida da aplicação (Figura 1), o *Checkov* auxilia a recomeçar o ciclo.

A integração do *Conformity* na pipeline também respeitou o conceito do DevOps. A ferramenta traz visibilidade sobre todos os recursos criados no ambiente na nuvem, além de trazer *checks* apontando riscos de segurança dos serviços que estão em produção na *cloud*. A integração nativa que a ferramenta possui com o Jira, facilita a criação de tarefas que, consequentemente, facilita e ajuda a organizar as próximas etapas a serem tomadas na fase de planejamento para tornar o ambiente mais seguro.

A funcionalidade do *CSPM* mais dedicada para o DevOps, que é a auto-remediação, também se encaixou perfeitamente na *pipeline*. Quando o *Conformity* detecta uma não conformidade e realiza a correção, ele toma as ações poupando a equipe de um ciclo inteiro de desenvolvimento. É importante comentar sobre a importância que é ter um bom conhecimento do ambiente da aplicação para que essa funcionalidade traga apenas benefícios. Como o *CSPM* irá fazer as mudanças automaticamente, é importante que antes de ativar qualquer regra a equipe responsável tenha entendimento completo de tal função pode afetar o ambiente.

De maneira geral, a criação do fluxo não apresentou desafios significativos. Integrar

todas as diferentes ferramentas utilizadas no projeto foi fácil devido a grande flexibilidade que a ferramenta providencia. A maior dificuldade do projeto foi de estimar o custo que as ferramentas podem trazer com elas. Uma das principais vantagens da nuvem é de que o custo dos serviços é baseado no uso deles. O envio do erro no Conformity para o SNS da AWS é uma etapa importante do projeto que também tem um custo associado a ele. Porém para fazer uma estimativa de quanto esse serviço pode ter é necessário ter uma estimativa de quantos erros são detectados (melhor se separar por tipo de erro). Infelizmente, o Conformity não proporciona essa visibilidade sobre o histórico dos status dos erros.

Referências

ATLASSIAN. *What is DevOps*. 2022. Disponível em: <<https://www.atlassian.com/devops/what-is-devops>>. Acesso em: 22 out. 2018.

AWS. *Shared Responsibility Model*. 2023. Disponível em: <<https://aws.amazon.com/pt/compliance/shared-responsibility-model/>>.

CROWLEY, C. Understanding devops: Exploring the origins, composition, merits, and perils of a devops capability. *Maynooth University Research Archive Library*, 2018.

GARTNER. *Gartner Says More Than Half of Enterprise IT Spending Will Be Shifted to Cloud by 2025*. 2022. Disponível em: <<https://www.gartner.com/en/newsroom/press-releases/2022-02-09-gartner-says-more-than-half-of-enterprise-it-spending>>.

HASHICORP. *Terraform Cloud*. 2022. Disponível em: <<https://developer.hashicorp.com/terraform/cloud-docs>>.

HASHICORP. *Terraform Intro*. 2022. Disponível em: <<https://developer.hashicorp.com/terraform/intr>>.

IBM. *O que é DevOps*. 2021. Disponível em: <<https://www.ibm.com/br-pt/topics/devops>>.

MUNIZ, A. *Jornada DevOps*. [S.l.]: Brasport, 2019.

PEDROSA, P. H. Computação em nuvem. *Instituto de Computação da UNICAMP*, 2011.

REDHAT. *Devops*. 2022. Disponível em: <<https://www.redhat.com/pt-br/topics/devops>>. Acesso em: 22 out. 2023.

REDHAT. *What is CI/CD*. 2022. Disponível em: <<https://www.redhat.com/pt-br/topics/devops/what-is-ci-cd>>.

VIEIRA, C. S.; MEIRELLES, F. de S. Computação em nuvem: Análise bibliométrica da produção científica sobre os fatores que influenciam as empresas no seu uso. *FVP EAESP*, 2015.