Laboratório Completo: A Trilha DevOps - Sessão 1

Objetivo da Sessão: Compreender os conceitos fundamentais de DevOps e IaC, configurar o ambiente de desenvolvimento, iniciar o uso prático de Terraform para provisionar infraestrutura básica na AWS (uma VPC e um S3 Bucket) e utilizar Git/GitHub para controle de versão do código de infraestrutura.

Duração Estimada: 1 hora (15 min teoria, 45 min prática)

Pré-requisitos:

- Conta AWS ativa com permissões de administrador (ou suficientes para criar VPC, S3, IAM).
- Conhecimentos básicos de linha de comando (Bash/Shell).
- Computador com acesso à internet.
- Conta no GitHub (gratuita).

Parte 1: Conceitos Fundamentais (Teoria - 15 min)

(Esta seção seria idealmente apresentada verbalmente ou com slides, mas incluímos um resumo escrito para referência.)

1.1 O que é DevOps?

DevOps não é apenas uma ferramenta ou um cargo, é uma **cultura** e um conjunto de **práticas** que visam unificar o desenvolvimento de software (Dev) e as operações de TI (Ops). O objetivo principal é encurtar o ciclo de vida de desenvolvimento de sistemas e fornecer entrega contínua com alta qualidade de software.

Os pilares do DevOps podem ser resumidos pelo acrônimo **CALMS**:

- **C**ulture (Cultura): Promover colaboração, comunicação e responsabilidade compartilhada entre as equipes de desenvolvimento, operações e outras áreas (como segurança e negócios).
- **A**utomation (Automação): Automatizar o máximo possível do ciclo de vida da aplicação, desde o build, testes, até o deploy e provisionamento de infraestrutura. Isso reduz erros manuais e acelera a entrega.
- **L**ean (Enxuto): Aplicar princípios Lean para eliminar desperdícios (atrasos, trabalho desnecessário, defeitos) no processo de entrega de software.

- **M**easurement (Medição): Coletar métricas e monitorar todos os aspectos do processo e da aplicação para identificar gargalos, entender o desempenho e tomar decisões baseadas em dados.
- **S**haring (Compartilhamento): Incentivar o compartilhamento de conhecimento, ferramentas e feedback entre as equipes.

Benefícios: Maior velocidade de entrega, maior confiabilidade e estabilidade das aplicações, melhor colaboração entre equipes, segurança integrada ao processo (DevSecOps) e capacidade de resposta mais rápida às necessidades do mercado.

1.2 Infraestrutura como Código (IaC)

Infraestrutura como Código (IaC) é a prática de gerenciar e provisionar infraestrutura de TI (redes, máquinas virtuais, balanceadores de carga, bancos de dados, etc.) através de **arquivos de definição legíveis por máquina** (código), em vez de configuração manual ou ferramentas interativas.

Vantagens:

- **Automação:** Provisionamento rápido e repetível de ambientes complexos.
- **Versionamento:** O código de infraestrutura pode ser versionado em sistemas como o Git, permitindo rastrear mudanças, reverter para versões anteriores e colaborar.
- **Reprodutibilidade:** Garante que ambientes idênticos (desenvolvimento, teste, produção) possam ser criados de forma consistente.
- **Reutilização:** Criação de módulos e templates para componentes de infraestrutura comuns.
- **Documentação:** O próprio código serve como documentação da infraestrutura.

1.3 Introdução ao Terraform

Terraform é uma ferramenta open-source de IaC criada pela HashiCorp. Ela permite definir a infraestrutura em uma linguagem declarativa chamada **HCL (HashiCorp Configuration Language)** e gerencia o ciclo de vida dessa infraestrutura em múltiplos provedores de nuvem (AWS, Azure, GCP) e outros serviços.

Como funciona:

1. **Escrever:** Você define a infraestrutura desejada em arquivos `.tf` usando HCL.
2. **Planejar (`terraform plan`):** O Terraform compara o estado desejado (seu código) com o estado real da infraestrutura (registrado no *state file*) e mostra quais mudanças serão feitas (criação, atualização, destruição de recursos).
3. **Aplicar (`terraform apply`):** O Terraform executa as ações planejadas para alcançar o estado desejado, interagindo com as APIs do provedor (ex: AWS API).

Componentes Chave:

- **Providers:** Plugins que permitem ao Terraform interagir com APIs específicas (ex: `aws`, `azure`, `google`).
- **Resources:** Blocos de código que definem um componente da infraestrutura (ex: `aws_instance`, `aws_s3_bucket`).
- **Variables:** Permitem parametrizar o código (ex: região AWS, nome do ambiente).
- **Outputs:** Expõem informações sobre a infraestrutura criada (ex: IP público de uma instância, nome DNS de um load balancer).
- **State File (`terraform.tfstate`):** Arquivo JSON que armazena o estado da infraestrutura gerenciada pelo Terraform. **É crucial não perdê-lo e, em ambientes de equipe, armazená-lo remotamente e de forma segura.**

Comandos Básicos:

- `terraform init`: Inicializa o diretório de trabalho (baixa providers, configura backend).
- `terraform fmt`: Formata o código HCL para um estilo padrão.
- `terraform validate`: Verifica a sintaxe do código.
- `terraform plan`: Mostra o plano de execução.
- `terraform apply`: Aplica as mudanças.
- `terraform destroy`: Destrói a infraestrutura gerenciada.

1.4 Controle de Versão com Git e GitHub

Git é um sistema de controle de versão distribuído, open-source, essencial para o desenvolvimento de software moderno e também para IaC. Ele permite:

- Rastrear mudanças no código ao longo do tempo.
- Trabalhar em diferentes funcionalidades ou correções simultaneamente usando *branches*.
- Mesclar (`merge`) mudanças de diferentes branches.
- Reverter para versões anteriores.
- Colaborar com outros desenvolvedores.

GitHub é uma plataforma baseada na web que hospeda repositórios Git e oferece ferramentas adicionais para colaboração, como:

- Hospedagem de código (repositórios remotos).
- Pull Requests (mecanismo para propor e revisar mudanças).
- Issues (rastreamento de tarefas e bugs).
- Actions (automação de workflows - veremos na Sessão 2).
- Wikis, Projects, etc.

Usar Git e GitHub para o código Terraform é fundamental para aplicar as vantagens do versionamento à infraestrutura.

Parte 2: Configuração do Ambiente e Primeiro Código Terraform (Prática - 45 min)

Vamos colocar a mão na massa! Siga os passos abaixo detalhadamente.

2.1 Instalação do Terraform

O Terraform é distribuído como um binário único. A forma mais fácil de instalar e gerenciar versões é usando um gerenciador de versões como o `tfenv` (recomendado) ou baixando diretamente.

Opção A: Usando `tfenv` (Linux/macOS)

1. **Instalar `tfenv`:**

    - Abra seu terminal.
    - Se você não tem o `git` instalado, instale-o primeiro (ex: `sudo apt update && sudo apt install git -y` no Ubuntu/Debian, `brew install git` no macOS).
    - Clone o repositório do `tfenv`:

      git clone --depth=1 https://github.com/tfutils/tfenv.git ~/.tfenv

    - Adicione o `tfenv` ao seu PATH. Adicione as seguintes linhas ao seu arquivo de configuração do shell (ex: `~/.bashrc`, `~/.zshrc`):

      export PATH="$HOME/.tfenv/bin:$PATH"

    - Recarregue a configuração do shell (ex: `source ~/.bashrc`) ou abra um novo terminal.
    - Verifique a instalação: `tfenv --version`

2. **Instalar uma versão do Terraform:**

    - Liste as versões disponíveis: `tfenv list-remote`
    - Instale a versão mais recente (ou uma específica, ex: 1.8.4): `tfenv install latest` (ou `tfenv install 1.8.4`)

- Defina a versão a ser usada globalmente: `tfenv use latest` (ou `tfenv use 1.8.4`)
- Verifique a instalação do Terraform: `terraform version`

Opção B: Download Manual (Linux/macOS/Windows)

1. Acesse a página de downloads do Terraform: https://developer.hashicorp.com/terraform/downloads
2. Baixe o pacote apropriado para seu sistema operacional (ex: Linux AMD64, Windows AMD64).
3. Descompacte o arquivo baixado. Ele conterá um único executável `terraform` (ou `terraform.exe` no Windows).
    - Pesquise variáveis de ambiente
4. Mova este executável para um diretório que esteja no seu PATH do sistema (ex: `/usr/local/bin` no Linux/macOS, ou adicione o diretório onde você o colocou à variável de ambiente PATH no Windows).
5. Abra um novo terminal/prompt de comando e verifique a instalação: `terraform version`

2.2 Configuração das Credenciais AWS

O Terraform precisa de permissões para interagir com sua conta AWS. Existem várias formas de configurar isso, mas a mais comum para desenvolvimento local é usar as credenciais de um usuário IAM.

Passo 1: Criar um Usuário IAM (se ainda não tiver um para desenvolvimento)

1. Faça login no **Console de Gerenciamento da AWS**: https://aws.amazon.com/console/
2. Navegue até o serviço **IAM** (Identity and Access Management). Você pode usar a barra de busca no topo.
3. No menu lateral esquerdo, clique em **Users** (Usuários).
4. Clique no botão **Create user** (Criar usuário).
5. **User name:** Digite um nome descritivo, como `terraform-dev-user`.
6. **Provide user access to the AWS Management Console:** *Desmarque* esta opção, pois este usuário será usado apenas para acesso programático.
7. Clique em **Next**.
8. **Permissions options:** Selecione **Attach policies directly** (Anexar políticas diretamente).
9. **Permissions policies:** Na caixa de busca, procure e marque a política `AdministratorAccess`. **Atenção:** Para produção, siga o princípio do menor privilégio, concedendo apenas as permissões estritamente necessárias. Para este laboratório inicial, `AdministratorAccess` simplifica o processo.
10. Clique em **Next**.

11. Revise as configurações e clique em **Create user**.

Passo 2: Gerar Chaves de Acesso

1. Na lista de usuários IAM, clique no nome do usuário que você acabou de criar (`terraform-dev-user`).
2. Vá para a aba **Security credentials** (Credenciais de segurança).
3. Role para baixo até a seção **Access keys** (Chaves de acesso).
4. Clique em **Create access key** (Criar chave de acesso).
5. **Use case:** Selecione **Command Line Interface (CLI)**.
6. Marque a caixa **I understand the above recommendation and want to proceed to create an access key**.
7. Clique em **Next**.
8. **Set description tag (optional):** Você pode adicionar uma tag descritiva, como `Terraform Dev Key`.
9. Clique em **Create access key**.
10. **IMPORTANTE:** A AWS mostrará o **Access key ID** e o **Secret access key**. Este é o **único momento** em que a chave secreta será exibida. Copie ambos os valores e guarde-os em um local seguro (gerenciador de senhas, por exemplo). Você pode também clicar em **Download .csv file** para salvar as credenciais.
11. Clique em **Done**.

Passo 3: Configurar as Credenciais para o Terraform

Vamos usar variáveis de ambiente, que é um método comum.

1. Abra seu terminal.
2. Defina as variáveis de ambiente (substitua `YOUR_ACCESS_KEY_ID` e `YOUR_SECRET_ACCESS_KEY` pelos valores que você copiou):
    - **Linux/macOS:**

      export AWS_ACCESS_KEY_ID="YOUR_ACCESS_KEY_ID"

      export AWS_SECRET_ACCESS_KEY="YOUR_SECRET_ACCESS_KEY"

      # Opcional, mas recomendado: defina a região padrão

      export AWS_DEFAULT_REGION="us-east-1" # Ou a região de sua preferência

      Dica: Adicione esses exports ao seu `~/.bashrc` ou `~/.zshrc` para não precisar digitá-los toda vez que abrir um novo terminal, mas esteja ciente das implicações de segurança.

- **Windows (Command Prompt):**

  set AWS_ACCESS_KEY_ID=YOUR_ACCESS_KEY_ID

  set AWS_SECRET_ACCESS_KEY=YOUR_SECRET_ACCESS_KEY

  set AWS_DEFAULT_REGION=us-east-1

- **Windows (PowerShell):**

  $env:AWS_ACCESS_KEY_ID = "YOUR_ACCESS_KEY_ID"

  $env:AWS_SECRET_ACCESS_KEY = "YOUR_SECRET_ACCESS_KEY"

  $env:AWS_DEFAULT_REGION = "us-east-1"

3. O Terraform usará automaticamente essas variáveis de ambiente para autenticar com a AWS.

2.3 Criação do Repositório no GitHub

1. Acesse o **GitHub**: https://github.com/
2. Faça login na sua conta.
3. No canto superior direito, clique no ícone `+` e selecione **New repository**.
4. **Repository name:** Digite um nome, por exemplo, `trilha-devops-lab`.
5. **Description (optional):** Adicione uma breve descrição, como `Laboratório do Treinamento A Trilha DevOps`.
6. Selecione **Public** ou **Private** (para este lab, Public é aceitável, mas para projetos reais, Private é geralmente preferido).
7. **IMPORTANTE:** Marque a opção **Add a README file**. Isso inicializa o repositório com um arquivo.
8. **Add .gitignore:** Selecione o template `Terraform` na lista suspensa. Isso adicionará um arquivo `.gitignore` padrão para excluir arquivos sensíveis ou desnecessários do Terraform (como o state file local).
9. **Choose a license:** Opcional, você pode escolher uma licença como MIT ou Apache 2.0.
10. Clique em **Create repository**.

2.4 Clonar o Repositório Localmente

1. Na página do seu novo repositório no GitHub, clique no botão verde **<> Code**.
2. Certifique-se de que a aba **HTTPS** (ou SSH, se você configurou chaves SSH) está selecionada.
3. Copie a URL do repositório (ex: `https://github.com/SEU_USUARIO/trilha-devops-lab.git`).

4. Abra seu terminal.
5. Navegue até o diretório onde você deseja guardar seus projetos (ex: `cd ~/Projects`).
6. Execute o comando `git clone` colando a URL copiada:

   git clone https://github.com/SEU_USUARIO/trilha-devops-lab.git

7. Entre no diretório do projeto recém-clonado:

   cd trilha-devops-lab

8. Você verá os arquivos `README.md` e `.gitignore` que o GitHub criou.

2.5 Escrevendo o Primeiro Código Terraform

Vamos criar a infraestrutura mais básica: uma VPC (Virtual Private Cloud) customizada e um S3 Bucket.

1. Dentro do diretório `trilha-devops-lab`, crie os seguintes arquivos usando seu editor de texto ou IDE preferido (VS Code com a extensão HashiCorp Terraform é recomendado):

   - `main.tf`: Arquivo principal onde definiremos os recursos.
   - `variables.tf`: Onde definiremos as variáveis de entrada.
   - `outputs.tf`: Onde definiremos os valores de saída.
   - `providers.tf`: Onde configuraremos o provider AWS.

2. **Edite `providers.tf`:**

   ```
   # providers.tf

   terraform {

     required_providers {

       aws = {

         source  = "hashicorp/aws"

         version = "~> 5.0" # Use uma versão recente do provider AWS

       }

     }
   ```

```
  required_version = ">= 1.0" # Exige versão mínima do Terraform
}

provider "aws" {
  region = var.aws_region
}
```

- `terraform {}`: Bloco de configuração do Terraform.
- `required_providers {}`: Especifica os providers necessários e suas versões.
- `required_version`: Especifica a versão mínima do Terraform.
- `provider "aws" {}`: Configura o provider AWS, definindo a região a ser usada através de uma variável `aws_region`.

3. **Edite `variables.tf`:**

```
# variables.tf

variable "aws_region" {
  description = "Região AWS para criar os recursos"
  type        = string
  default     = "us-east-1" # Você pode mudar para sua região preferida
}

variable "vpc_cidr_block" {
  description = "Bloco CIDR para a VPC"
  type        = string
  default     = "10.0.0.0/16"
}

variable "bucket_prefix" {
```

```
  description = "Prefixo para o nome do bucket S3 (será concatenado com um ID único)"
  type        = string
  default     = "trilha-devops-lab"
}

variable "common_tags" {
  description = "Tags comuns para aplicar a todos os recursos"
  type        = map(string)
  default = {
    Project     = "Trilha DevOps Lab"
    Environment = "Development"
    ManagedBy   = "Terraform"
  }
}
```

- Cada bloco `variable` define uma variável de entrada com descrição, tipo e valor padrão.

4. **Edite `main.tf`:**

```
# main.tf

# Criação da VPC
resource "aws_vpc" "main" {
  cidr_block           = var.vpc_cidr_block
  enable_dns_support   = true
  enable_dns_hostnames = true
```

```
  tags = merge(var.common_tags, {
    Name = "trilha-devops-vpc"
  })
}
# Criação do S3 Bucket
# Nota: Nomes de bucket S3 são globalmente únicos.
# Usaremos um nome aleatório para evitar conflitos.
resource "random_id" "bucket_id" {
  byte_length = 8
}
resource "aws_s3_bucket" "lab_bucket" {
  # Nome do bucket: prefixo definido na variável + ID aleatório
  bucket = "${var.bucket_prefix}-${random_id.bucket_id.hex}"
  tags = merge(var.common_tags, {
    Name = "trilha-devops-lab-bucket"
  })
}
# Configuração de versionamento para o bucket
resource "aws_s3_bucket_versioning" "lab_bucket_versioning" {
  bucket = aws_s3_bucket.lab_bucket.id # Referencia o bucket criado acima
  versioning_configuration {
```

```
    status = "Enabled"

  }

}

# Configuração de bloqueio de acesso público para o bucket (Boas práticas de segurança)

resource "aws_s3_bucket_public_access_block" "lab_bucket_pab" {

  bucket = aws_s3_bucket.lab_bucket.id

  block_public_acls       = true

  block_public_policy     = true

  ignore_public_acls      = true

  restrict_public_buckets = true

}
```

- `resource "aws_vpc" "main" {}`: Define um recurso do tipo VPC com o nome lógico `main`.
- `cidr_block = var.vpc_cidr_block`: Usa o valor da variável `vpc_cidr_block`.
- `tags = merge(...)`: Combina as tags comuns com uma tag específica `Name`.
- `resource "random_id" "bucket_id" {}`: Usa o provider `random` (implícito) para gerar um sufixo aleatório para o nome do bucket S3, garantindo unicidade global.
- `resource "aws_s3_bucket" "lab_bucket" {}`: Define o bucket S3.
- `bucket = "${var.bucket_prefix}-${random_id.bucket_id.hex}"`: Concatena o prefixo da variável com o ID aleatório gerado.
- `resource "aws_s3_bucket_versioning" ...`: Configura o versionamento, referenciando o bucket pelo seu ID (`aws_s3_bucket.lab_bucket.id`).
- `resource "aws_s3_bucket_public_access_block" ...`: Aplica configurações de segurança para bloquear acesso público.

5. **Edite `outputs.tf`:**

```
# outputs.tf
```

```
output "vpc_id" {
  description = "ID da VPC criada"
  value       = aws_vpc.main.id
}

output "s3_bucket_name" {
  description = "Nome do Bucket S3 criado"
  value       = aws_s3_bucket.lab_bucket.bucket
}

output "s3_bucket_arn" {
  description = "ARN do Bucket S3 criado"
  value       = aws_s3_bucket.lab_bucket.arn
}
```

- Cada bloco `output` define um valor que será exibido após a aplicação do Terraform.

2.6 Ciclo Básico do Terraform

Agora vamos executar os comandos do Terraform para criar a infraestrutura definida.

1. **Inicializar o Terraform:**
   - No terminal, dentro do diretório `trilha-devops-lab`, execute:

     ```
     terraform init
     ```

   - **O que acontece:** O Terraform baixa o provider `hashicorp/aws` na versão especificada e configura o backend (neste caso, local). Você verá uma mensagem `Terraform has been successfully initialized!`.
2. **Formatar o Código:**

- Execute:

  terraform fmt

- **O que acontece:** O Terraform verifica e ajusta a formatação dos seus arquivos `.tf` para o padrão. Se algum arquivo for modificado, ele será listado.

3. **Validar a Sintaxe:**

   - Execute:

     terraform validate

   - **O que acontece:** O Terraform verifica se a sintaxe do seu código HCL está correta e se todas as referências (variáveis, recursos) são válidas. Se tudo estiver OK, você verá `Success! The configuration is valid.`.

4. **Planejar as Mudanças:**

   - Execute:

     terraform plan

   - **O que acontece:** O Terraform lê seu código, verifica o estado atual (como não há estado ainda, ele assume que nada existe) e mostra um plano detalhado do que será criado, modificado ou destruído. Neste caso, ele mostrará que 4 recursos serão criados (`aws_vpc`, `random_id`, `aws_s3_bucket`, `aws_s3_bucket_versioning`, `aws_s3_bucket_public_access_block`). Revise o plano cuidadosamente.

5. **Aplicar as Mudanças:**

   - Execute:

     terraform apply

   - **O que acontece:** O Terraform mostrará o mesmo plano novamente e pedirá sua confirmação para prosseguir. Digite `yes` e pressione Enter.
   - O Terraform começará a criar os recursos na sua conta AWS, mostrando o progresso. Isso pode levar alguns segundos ou minutos.

- Ao final, você verá `Apply complete! Resources: 5 added, 0 changed, 0 destroyed.` e os valores definidos nos `outputs` (ID da VPC, nome e ARN do bucket).
- Um arquivo `terraform.tfstate` será criado no diretório. **Não o edite manualmente!**

6. **Verificar no Console AWS:**

    - Abra o Console da AWS.
    - Navegue até o serviço **VPC**.
        - No menu esquerdo, clique em **Your VPCs**.
        - Verifique se a VPC `trilha-devops-vpc` foi criada com o CIDR `10.0.0.0/16`.
    - Navegue até o serviço **S3**.
        - Verifique se o bucket com o nome `trilha-devops-lab-<id_aleatorio>` foi criado.
        - Clique no nome do bucket, vá para a aba **Properties** (Propriedades) e verifique se o **Bucket Versioning** está *Enabled*.
        - Vá para a aba **Permissions** (Permissões) e verifique se o **Block public access** está *On*.

2.7 Controle de Versão com Git

Vamos salvar nosso progresso no GitHub.(https://cli.github.com/)

1. **Verificar o Status do Git:**

    - No terminal, execute:

      git status

    - **O que acontece:** O Git mostrará os arquivos novos ou modificados que ainda não foram adicionados ao controle de versão (`main.tf`, `variables.tf`, `outputs.tf`, `providers.tf`). Ele também deve mostrar o diretório `.terraform/` e o arquivo `terraform.tfstate` como *untracked*, mas eles devem ser ignorados devido ao `.gitignore` que selecionamos ao criar o repositório (verifique o conteúdo do `.gitignore` se não estiverem sendo ignorados).

2. **Adicionar Arquivos ao Staging:**

    - Execute:

git add .

- **O que acontece:** Adiciona todos os arquivos novos/modificados (exceto os ignorados) à área de *staging*, preparando-os para o commit.
- Verifique novamente com `git status`. Os arquivos devem aparecer como `Changes to be committed`.

3. **Fazer o Commit:**

   - Execute:

     git commit -m "feat: Add Terraform code for VPC and S3 Bucket"

   - **O que acontece:** Grava as mudanças no histórico local do Git com uma mensagem descritiva (usamos o padrão Conventional Commits aqui: `feat:` para nova funcionalidade).

4. **Enviar para o GitHub:**

   - Execute:

     git push origin main

   - **O que acontece:** Envia os commits locais da branch `main` para o repositório remoto `origin` (seu repositório no GitHub).
   - Pode ser necessário digitar seu usuário e senha do GitHub (ou usar um token/SSH).

5. **Verificar no GitHub:**

   - Atualize a página do seu repositório no GitHub. Você verá os novos arquivos Terraform e o commit que você acabou de fazer.

2.8 Desafio

Modifique o código Terraform para adicionar uma tag extra (ex: `Owner = "Seu Nome"`) a **ambos** os recursos (VPC e S3 Bucket). Use a função `merge` como fizemos antes.

1. Edite o arquivo `main.tf` adicionando a nova tag dentro dos blocos `tags` da VPC e do Bucket.
2. Rode `terraform plan`. Verifique se o plano mostra apenas a **modificação** das tags nos recursos existentes.
3. Rode `terraform apply` e confirme com `yes`.

4. Verifique as tags atualizadas no Console da AWS.
5. Adicione, comite (`git commit -m "feat: Add Owner tag to resources"`) e envie (`git push`) as mudanças para o GitHub.

2.9 Limpeza (Opcional, mas recomendado)

Para evitar custos inesperados na AWS, destrua os recursos criados neste laboratório.

1. No terminal, dentro do diretório `trilha-devops-lab`, execute:

   terraform destroy

2. O Terraform mostrará um plano indicando que todos os recursos gerenciados serão destruídos.
3. Confirme digitando `yes` e pressionando Enter.
4. Aguarde a conclusão. Você verá `Destroy complete! Resources: 5 destroyed.`.
5. Verifique no Console da AWS que a VPC e o Bucket S3 foram removidos.

Fim da Sessão 1. Na próxima sessão, introduziremos CI/CD e automatizaremos a validação e o planejamento do nosso código Terraform usando GitHub Actions.