Aula 33 – Construindo um Pipeline de CD para o Kubernetes com GitHub Actions

Imagine que você passou semanas desenvolvendo uma nova funcionalidade incrível para sua aplicação. O código está pronto, testado e elegantemente armazenado em seu repositório Git. E agora? Entre esse ponto e o momento em que seus usuários podem de fato utilizar a novidade, existe um abismo, muitas vezes preenchido por processos manuais, repetitivos e suscetíveis a erros. É como ter um carro de corrida fantástico, mas precisar empurrá-lo manualmente até a linha de largada para cada corrida.

O objetivo desta aula é construir uma ponte robusta e automatizada sobre esse abismo. Ao final de nossa jornada de 60 minutos, você não apenas entenderá, mas será capaz de construir um pipeline de Entrega Contínua (CD) completo. Usando a força do **GitHub Actions**, ensinaremos nosso repositório a pegar o código, construir um contêiner Docker, armazená-lo com segurança e, o mais importante, implantar a nova versão em um cluster **Kubernetes** sem que você precise digitar um único comando manual.

Nós começaremos estabelecendo uma relação de confiança segura entre o GitHub e seu cluster, a base de toda a automação. Em seguida, criaremos nossa linha de montagem digital para fabricar e estocar nossas imagens de contêiner. Finalmente, orquestraremos o fluxo completo, fazendo com que uma simples ação de git push desencadeie uma série de eventos que levam sua aplicação ao ar. Se você já se sentiu confortável criando um Dockerfile, prepare-se para ver esse processo ganhar superpoderes.

A Chave do Reino: Autenticação Segura entre Mundos

Antes que nosso pipeline possa começar a dar ordens ao nosso cluster Kubernetes, precisamos resolver uma questão fundamental de confiança. Como uma máquina, o executor do GitHub Actions, pode provar sua identidade e ser autorizada a fazer alterações em outra máquina, o nosso cluster? Deixar uma senha ou um arquivo de configuração com acesso total exposto em nosso repositório seria o equivalente digital a deixar a chave de um cofre embaixo do tapete. É um convite ao desastre.

> **O Desafio da Confiança:** Não é apenas conectar os dois sistemas, mas fazê-lo de forma segura, auditável e seguindo o princípio do menor privilégio.

O desafio, portanto, não é apenas conectar os dois sistemas, mas fazê-lo de uma forma que seja segura, auditável e que siga o princípio do menor privilégio. Precisamos de um método que conceda ao pipeline apenas as permissões estritamente necessárias para realizar seu trabalho, e nada mais. É aqui que a segurança deixa de ser um obstáculo e se torna uma peça fundamental do design do nosso sistema, uma filosofia conhecida como **DevSecOps**, ou segurança "deslocada para a esquerda" (*Shift-Left Security*).

01

Criar ServiceAccount no Kubernetes

Uma identidade específica para o pipeline com permissões limitadas

02

Gerar Token de Acesso

O "crachá" que permite acesso controlado ao cluster

03

Armazenar no GitHub Secrets

Guardar o token de forma segura como ${{ secrets.KUBE_CONFIG_DATA }}

A solução se assemelha a conceder um crachá de acesso a um visitante em um prédio corporativo. Em vez de dar a chave mestra, criamos uma identidade específica para o nosso pipeline dentro do Kubernetes, chamada de ServiceAccount. Associamos a essa conta apenas as permissões de que ela precisa (por exemplo, atualizar um *Deployment* específico). O token de acesso gerado para essa ServiceAccount é o nosso crachá. Este token, que é um texto longo e secreto, é então armazenado no cofre do GitHub, o **GitHub Actions Secrets**. Em nosso arquivo de workflow, nós nos referimos a ele de forma segura, como ${{ secrets.KUBE_CONFIG_DATA }}, sem nunca expor seu valor real no código.

A Linha de Montagem: Construindo a Imagem Docker

Agora que a porta do nosso cluster está aberta (mas apenas para o visitante autorizado), podemos focar na primeira tarefa do nosso pipeline: a fabricação do nosso produto. O código-fonte em nosso repositório é como a matéria-prima; para que ele possa ser executado no ambiente padronizado do Kubernetes, ele precisa ser processado e empacotado em uma unidade funcional e portátil: uma **imagem de contêiner Docker**.

Você provavelmente já fez isso manualmente, executando o comando docker build. O problema dessa abordagem é a inconsistência. A sua máquina pode ter uma versão diferente do Docker, ou você pode esquecer de um passo, resultando em uma imagem diferente da do seu colega. A automação resolve isso.

Rastreabilidade: Etiquetamos cada imagem com o hash do commit do Git para saber exatamente qual versão do código a originou.

O Processo Automatizado

Git Push

Desenvolvedor envia código para branch principal

GitHub Actions

Workflow é acionado automaticamente

Docker Build

Imagem construída seguindo o Dockerfile

Tag Única

Imagem etiquetada com hash do commit

Para isso, nosso workflow do GitHub Actions atuará como o gerente desta fábrica automatizada. Ele usará o Dockerfile do nosso projeto como o manual de instruções detalhado. A cada novo git push para a branch principal, o gerente (o workflow) inicia o processo. Usando uma ação pré-fabricada, como a docker/build-push-action, ele segue o manual à risca para construir a imagem. Para garantir a **rastreabilidade** — um pilar da **Observabilidade** —, nós a etiquetamos (*tag*) com um identificador único, como o hash do commit do Git. Assim, sempre saberemos exatamente qual versão do código deu origem a cada imagem que produzimos.

O Armazém Global: Enviando para um Registro de Contêineres

Nossa linha de montagem acaba de produzir uma imagem de contêiner novinha em folha. No entanto, ela foi construída em um ambiente temporário, uma máquina virtual que o GitHub Actions criou apenas para executar nosso pipeline e que será destruída em minutos. Para que nosso cluster Kubernetes possa acessar essa imagem, precisamos movê-la para um local permanente e acessível pela internet: um **Registro de Contêineres** (*Container Registry*).

Docker Hub

Registro público mais popular, ideal para projetos open source

GitHub Container Registry (GHCR)

Integrado ao GitHub, perfeito para workflows nativos

Registros de Nuvem

GCR (Google), ECR (AWS), ACR (Azure) para ambientes corporativos

Pense em um registro como um armazém ou um centro de distribuição global para nossas aplicações. Plataformas como o Docker Hub, o GitHub Container Registry (GHCR) ou os registros oferecidos por provedores de nuvem (GCR, ECR, ACR) são projetados especificamente para armazenar, gerenciar e distribuir imagens de contêiner de forma eficiente e segura. Sem esse passo, nossa imagem recém-criada ficaria isolada e se perderia ao final da execução do pipeline.

> **Analogia:** É como subir um vídeo para uma plataforma de streaming. Você o produz e edita em seu computador local (o *build* da imagem no runner do GitHub), mas para que o mundo possa assisti-lo, você o envia para o YouTube ou Vimeo (o *push* para o registro).

A lógica é muito parecida com a de subir um vídeo para uma plataforma de streaming. Você o produz e edita em seu computador local (o *build* da imagem no runner do GitHub), mas para que o mundo possa assisti-lo, você o envia para o YouTube ou Vimeo (o *push* para o registro). Nosso pipeline, continuando a usar a ação docker/build-push-action, agora realizará a segunda parte de sua função. Após o *build*, ele se autenticará no registro — novamente, usando um token de acesso armazenado de forma segura nos **GitHub Secrets** — e fará o *push* da imagem etiquetada para o nosso armazém.

Isso nos leva à próxima etapa crucial do processo. A mercadoria já está no centro de distribuição. Como notificamos o cliente final, nosso cluster Kubernetes, que há uma nova versão disponível para ele?

O Mensageiro Automático: Atualizando o Manifesto de Implantação

O fato de uma nova imagem da nossa aplicação estar disponível no registro não muda nada para os usuários. O cluster Kubernetes continua executando a versão antiga, pois ele opera com base no estado descrito em seus arquivos de manifesto, como o deployment.yaml. É este arquivo que diz ao Kubernetes: "Execute a aplicação X, usando a imagem Y, com 3 réplicas". Enquanto a "imagem Y" não for atualizada no manifesto, nada acontece.

O Problema Manual

- Editar o arquivo deployment.yaml
- Alterar a tag da imagem manualmente
- Executar kubectl apply
- Quebra o fluxo contínuo

A Solução Automatizada

- Pipeline atualiza o manifesto automaticamente
- Usa ferramentas como sed ou yq
- Executa kubectl apply de forma segura
- Mantém o fluxo contínuo

O gargalo aqui é a necessidade de atualizar esse manifesto. Fazer isso manualmente — editar o arquivo, alterar a tag da imagem para o novo hash do commit e aplicar a mudança com kubectl apply — quebra completamente o fluxo contínuo. Precisamos de um mensageiro automático, uma parte do nosso pipeline que, após confirmar que a nova imagem está no registro, se encarregue de notificar o cluster sobre a atualização.

> **Analogia do E-commerce:** Assim que o produto novo chega ao estoque (a imagem é enviada ao registro), o sistema precisa atualizar o catálogo online (o manifesto do Kubernetes) para que os clientes possam ver e comprar a nova versão.

Pense neste passo como o sistema de notificação de uma loja online. Assim que o produto novo chega ao estoque (a imagem é enviada ao registro), o sistema precisa atualizar o catálogo online (o manifesto do Kubernetes) para que os clientes possam ver e comprar a nova versão. No nosso pipeline, podemos adicionar um novo passo após o *push* da imagem. Este passo pode usar ferramentas de linha de comando, como sed ou yq, para encontrar a linha image: no nosso deployment.yaml e substituir a tag antiga pela nova. Em seguida, usando as credenciais seguras que configuramos, ele executa kubectl apply -f deployment.yaml, instruindo o Kubernetes a iniciar uma atualização gradual (*rolling update*) para a nova versão.

A Orquestração Completa: O Fluxo de CI/CD em Ação

Até agora, montamos as peças individuais da nossa automação: a autenticação, a construção da imagem, o envio para o registro e a atualização do manifesto. Chegou a hora de juntar tudo em um único fluxo de trabalho coeso e orquestrado. É aqui que o poder do GitHub Actions realmente se manifesta, permitindo-nos definir uma sequência de eventos lógica, dependente e totalmente automatizada.

Workflow = Partitura Musical

Define os "movimentos" (jobs) e as "notas" (steps) da automação

Dependências Inteligentes

O deploy só inicia após o build ser concluído com sucesso

Caminho Pavimentado

Base da Engenharia de Plataforma para entregas rápidas e seguras

Um workflow do GitHub Actions é como uma partitura musical para a nossa automação. Ele define os "movimentos" (os jobs ou trabalhos) e as "notas" dentro de cada movimento (os steps ou passos). Crucialmente, podemos definir dependências. O trabalho de implantação (deploy) só deve começar *depois* que o trabalho de construção e envio (build_and_push) for concluído com sucesso. Isso garante que nunca tentaremos implantar uma versão que falhou ao ser construída ou que não chegou ao registro.

Estrutura do Workflow YAML

```
name: Pipeline de Entrega Contínua para Kubernetes
on:
  push:
    branches: [ main ]  # Gatilho: um push para a branch main

jobs:
  build_and_push:
    name: Construir e Enviar Imagem
    runs-on: ubuntu-latest
    steps:
      - name: Checkout do código
        uses: actions/checkout@v4
      # ... outros passos para login no registro e build/push ...

  deploy:
    name: Implantar no Kubernetes
    runs-on: ubuntu-latest
    needs: build_and_push  # <- A mágica da dependência!
    steps:
      - name: Checkout do código
        uses: actions/checkout@v4
      # ... outros passos para configurar kubectl e aplicar o manifesto ...
```

Abaixo, vemos um esqueleto simplificado de como essa orquestração se parece em nosso arquivo de workflow YAML. Observe a diretiva needs: build_and_push, que cria essa dependência fundamental. Esse pequeno trecho de código é a espinha dorsal de um processo que, na prática, é a base do que equipes de **Engenharia de Plataforma** (*Platform Engineering*) buscam oferecer: um "caminho pavimentado" para que desenvolvedores possam entregar valor de forma rápida e segura, abstraindo a complexidade da infraestrutura.

Além do Óbvio: GitOps como a Evolução Natural

O pipeline que construímos é poderoso e representa uma abordagem muito comum, conhecida como **implantação baseada em push**. Nela, o sistema de CI/CD (GitHub Actions) é o agente ativo que "empurra" as mudanças para dentro do ambiente de produção. Funciona bem, mas exige que nosso pipeline tenha credenciais de acesso direto ao cluster, o que pode aumentar a superfície de ataque. E se pudéssemos inverter essa lógica?

CD Tradicional (Push)

Como funciona: O pipeline se conecta e "empurra" as atualizações para o cluster.

✓ Vantagem: Mais simples de configurar inicialmente, sem agentes extras.

× Desafio: Exige que as credenciais do cluster sejam expostas ao sistema de CI/CD.

GitOps (Pull)

Como funciona: Um agente no cluster "puxa" as mudanças declaradas no repositório Git.

✓ Vantagem: Segurança aprimorada (credenciais não saem do cluster); tudo é auditável via Git.

× Desafio: Requer a instalação e configuração de um agente específico no cluster.

Essa inversão de responsabilidade é a ideia central por trás do **GitOps**, uma tendência que está redefinindo o gerenciamento de infraestrutura em 2025. No modelo GitOps, o repositório Git é tratado como a **única fonte da verdade** para o estado desejado do nosso cluster. Em vez de o pipeline se conectar ao Kubernetes, um agente de software instalado *dentro* do cluster (como o Argo CD ou o Flux) monitora constantemente o repositório Git.

> **Analogia:** O modelo *push* é como um serviço de entregas que precisa do endereço e da chave da sua casa para deixar um pacote. O modelo *pull* do GitOps é como uma assinatura de jornal; você não dá sua chave ao entregador. Em vez disso, você mesmo pega o jornal novo na sua porta toda manhã.

A analogia é a seguinte: o modelo *push* é como um serviço de entregas que precisa do endereço e da chave da sua casa para deixar um pacote. O modelo *pull* do GitOps é como uma assinatura de jornal; você não dá sua chave ao entregador. Em vez disso, você mesmo pega o jornal novo na sua porta toda manhã. O agente GitOps no cluster "pega" o estado mais recente do repositório e se ajusta para corresponder a ele. O nosso pipeline de CI ainda constrói e envia a imagem, mas sua última tarefa é simplesmente comitar a mudança da tag da imagem no arquivo de manifesto de volta para o repositório Git. O agente GitOps cuida do resto, tornando o processo mais seguro e declarativo.

Refinando a Linha de Montagem: Segurança e Eficiência

Um pipeline funcional é o primeiro passo, mas um pipeline de nível profissional é rápido, eficiente e, acima de tudo, seguro. A filosofia de **DevSecOps** nos ensina a integrar a segurança em cada etapa do ciclo de vida do desenvolvimento, não como uma verificação final, mas como uma parte intrínseca do processo. Nossa linha de montagem automatizada é o lugar perfeito para fazer isso.

Segurança Integrada: Varredura de Vulnerabilidades

Trivy

Scanner de vulnerabilidades open source, rápido e abrangente

Snyk

Plataforma comercial com foco em segurança de código e dependências

Grype

Ferramenta da Anchore para detecção de CVEs em imagens

Imagine adicionar um posto de controle de qualidade e segurança em nossa fábrica digital. Antes de enviar a imagem de contêiner recém-construída para o nosso armazém global (o registro), podemos submetê-la a uma varredura automática de vulnerabilidades. Ferramentas como Trivy, Snyk ou Grype podem ser integradas como um passo no nosso workflow. Elas analisam as camadas da nossa imagem em busca de bibliotecas com vulnerabilidades conhecidas (CVEs). Se uma falha de segurança crítica for encontrada, o pipeline pode ser configurado para falhar, impedindo que uma imagem insegura sequer seja publicada.

FinOps: Otimização de Custos

- Imagens menores = menos custos de armazenamento
- Builds mais rápidos = menos tempo de computação
- Downloads mais rápidos = menos tráfego de rede

GreenOps: Sustentabilidade

- Menor consumo de energia nos builds
- Redução da pegada de carbono
- Uso eficiente de recursos computacionais

Essa mentalidade de otimização contínua também se aplica à eficiência, o que nos conecta às práticas de **FinOps** (Otimização de Custos de Nuvem) e **GreenOps** (Sustentabilidade em TI). Um Dockerfile bem escrito, que utiliza técnicas como *multi-stage builds*, gera imagens muito menores. Imagens menores são mais rápidas de construir, mais baratas de armazenar no registro e mais rápidas para o Kubernetes baixar e iniciar. Ao adicionar uma etapa de varredura e otimizar nosso Dockerfile, nosso pipeline não apenas entrega código, mas entrega código de qualidade, seguro e eficiente.

Da Reação à Previsão: Do Monitoramento à Observabilidade

Nosso pipeline executou com sucesso, o kubectl apply retornou "configured", e os novos pods estão em execução. Missão cumprida? Não tão rápido. A implantação bem-sucedida é apenas metade da história. A outra metade é garantir que a nova versão da aplicação está realmente saudável e performando como esperado. É aqui que evoluímos do simples monitoramento para a profunda **Observabilidade**.

Monitoramento Tradicional

Abordagem: Reativa

Responde: "O QUE está errado?"

Exemplo: "A utilização da CPU está em 95%!" ou "O serviço X não está respondendo!"

Analogia: Alarme de incêndio — só dispara quando o fogo já começou

Observabilidade Moderna

Abordagem: Proativa e investigativa

Responde: "POR QUE algo está errado?"

Exemplo: Correlaciona aumento de latência com erros específicos nos logs e identifica o serviço causador

Analogia: Sistema de detecção de fumaça com análise preditiva

Os Três Pilares da Observabilidade

1

Métricas

Dados numéricos agregados ao longo do tempo

- Uso de CPU e memória
- Taxa de requisições por segundo
- Latência média de resposta

2

Logs

Registros de eventos discretos do sistema

- "Usuário X fez login"
- "Erro ao conectar ao banco de dados"
- "Transação Y concluída"

3

Traces (Rastreamento)

Caminho completo de uma requisição através de múltiplos serviços

- Visualização de ponta a ponta
- Tempos de espera entre serviços
- Identificação de gargalos

O **monitoramento** tradicional é reativo. Ele nos diz *o que* está errado, geralmente através de alertas: "A utilização da CPU está em 95%!" ou "O serviço X não está respondendo!". É como o alarme de incêndio do seu sistema; ele é essencial, mas só dispara quando o fogo já começou. A **Observabilidade**, por outro lado, é proativa e investigativa. Ela nos ajuda a entender *por que* algo está errado, nos permitindo fazer perguntas que não sabíamos que precisávamos fazer.

A Observabilidade se apoia em três pilares de dados: **Métricas** (os dados numéricos, como uso de CPU), **Logs** (os registros de eventos, como "usuário X fez login") e **Traces** (o rastreamento do caminho completo de uma requisição através de múltiplos microsserviços). Ao instrumentar nossa aplicação para emitir esses três tipos de sinais e enviá-los a uma plataforma centralizada, ganhamos a capacidade de depurar sistemas complexos de forma eficaz. Após uma implantação, podemos, por exemplo, observar um aumento sutil na latência de uma API específica (métrica), correlacioná-lo com um novo tipo de erro nos logs e usar os traces para identificar exatamente qual serviço está causando o gargalo. É essa profundidade de análise que permite que as equipes de TI, apoiadas por **AIOps** (IA para Operações), identifiquem anomalias e prevejam problemas antes que eles impactem os usuários.

Consolidando a Automação e Olhando para o Futuro

Nesta aula, construímos muito mais do que um script. Criamos uma ponte automatizada e inteligente que conecta o código-fonte à produção. Começamos estabelecendo uma base segura de autenticação. Em seguida, montamos uma linha de produção para construir e armazenar nossas imagens de contêiner de forma consistente. Finalmente, orquestramos um fluxo completo que implanta automaticamente nossas atualizações no Kubernetes. Vimos como práticas modernas como **GitOps**, **DevSecOps** e **Observabilidade** não são apenas palavras da moda, mas evoluções cruciais que tornam nossos sistemas mais seguros, eficientes e resilientes.

Em Prática

Sempre use Secrets

Mantenha credenciais como tokens e senhas fora do seu código e utilize os mecanismos de segredos da sua plataforma de CI/CD.

Etiquete com Precisão

Use o hash do commit ou tags semânticas para suas imagens Docker, garantindo que você sempre saiba o que está em execução.

Comece Simples, Evolua

Inicie com um pipeline de *push* direto para ganhar familiaridade e, à medida que a complexidade e os requisitos de segurança aumentarem, explore a migração para uma abordagem GitOps.

Segurança Não é Opcional

Integre uma ferramenta de varredura de vulnerabilidades em seu pipeline desde o primeiro dia. É mais fácil e barato corrigir um problema antes que ele chegue à produção.

Autoavaliação

1. **(Nível: Fácil)** Qual é o principal objetivo de armazenar o token do Kubernetes no GitHub Actions Secrets?
   - a) Para acelerar o tempo de build do pipeline.
   - b) Para evitar a exposição de credenciais sensíveis diretamente no código do workflow.
   - c) Para versionar o token junto com o código-fonte no Git.
   - d) Para permitir que qualquer pessoa execute o pipeline.
2. **(Nível: Médio)** No contexto de um pipeline de CD, qual a vantagem de usar o hash do commit como tag para uma imagem Docker?
   - a) É a tag mais curta e fácil de lembrar.
   - b) Garante que a imagem possa ser executada em qualquer arquitetura de hardware.
   - c) Cria um vínculo direto e inequívoco entre a imagem e a versão exata do código-fonte que a gerou.
   - d) É um requisito obrigatório do Docker Hub.
3. **(Nível: Desafiador)** Uma organização decide migrar sua estratégia de implantação de um modelo *push* (pipeline aplicando kubectl) para um modelo *pull* (GitOps com Argo CD). Qual das seguintes afirmativas, no contexto de uma avaliação para concurso, melhor descreve a principal motivação de segurança para essa mudança?
   - a) Aumentar a velocidade do pipeline de CI, pois ele não precisa mais esperar pela etapa de kubectl apply.
   - b) Centralizar todos os manifestos do Kubernetes em um único repositório, facilitando a busca por arquivos.
   - c) Reduzir a superfície de ataque, eliminando a necessidade de o sistema de CI/CD possuir credenciais de escrita com acesso direto à API do Kubernetes.
   - d) Permitir que os desenvolvedores apliquem mudanças diretamente no cluster a partir de suas máquinas locais.
4. **(Nível: Especialista)** Ao integrar práticas de Observabilidade, qual pilar de dados é mais eficaz para entender a jornada completa de uma requisição de um usuário através de múltiplos microsserviços?
   - a) Métricas, pois mostram o consumo de recursos de cada serviço.
   - b) Logs, pois registram cada evento de forma cronológica.
   - c) Traces (Rastreamento Distribuído), pois visualizam o fluxo de ponta a ponta e os tempos de espera entre os serviços.
   - d) Alertas, pois notificam a equipe quando algo crítico acontece.
5. **(Questão Discursiva)** Descreva brevemente, em 3 a 5 linhas, por que a prática de "Shift-Left Security" (como a varredura de vulnerabilidades no pipeline) é considerada mais eficaz do que realizar auditorias de segurança apenas no ambiente de produção.

Gabarito e Próximos Passos

1-B	2-C	3-C	4-C
Questão 1	**Questão 2**	**Questão 3**	**Questão 4**
Evitar exposição de credenciais	Vínculo direto com código-fonte	Reduzir superfície de ataque	Traces para jornada completa

Resposta Discursiva (Exemplo)

A "Shift-Left Security" é mais eficaz porque identifica e corrige falhas de segurança no início do ciclo de desenvolvimento, quando o custo e o esforço para a correção são muito menores. Prevenir que uma vulnerabilidade chegue à produção é mais seguro e econômico do que remediar um problema em um sistema já em operação, que poderia ser explorado por atacantes.

Conexão com a Próxima Aula

> Até agora, construímos um pipeline que interage com uma infraestrutura Kubernetes já existente. Mas e se pudéssemos criar e gerenciar a própria infraestrutura — o cluster, as redes, os balanceadores de carga — com código, da mesma forma que gerenciamos nossa aplicação? Essa é a promessa da **Infraestrutura como Código (IaC)**, e será o nosso foco na **Aula 34 – Introdução à Infraestrutura como Código (IaC)**.

Recursos Adicionais

Documentação Oficial do GitHub Actions

Essencial para aprofundar em todos os gatilhos, contextos e ações disponíveis.

The Twelve-Factor App

Uma metodologia clássica que fornece princípios para a construção de aplicações robustas e escaláveis na nuvem, muito alinhada ao que fizemos.

NOTA IMPORTANTE: As informações técnicas e as práticas recomendadas nesta aula estão atualizadas até setembro de 2025. Consulte sempre a documentação oficial das ferramentas para verificar as versões e comandos mais recentes.