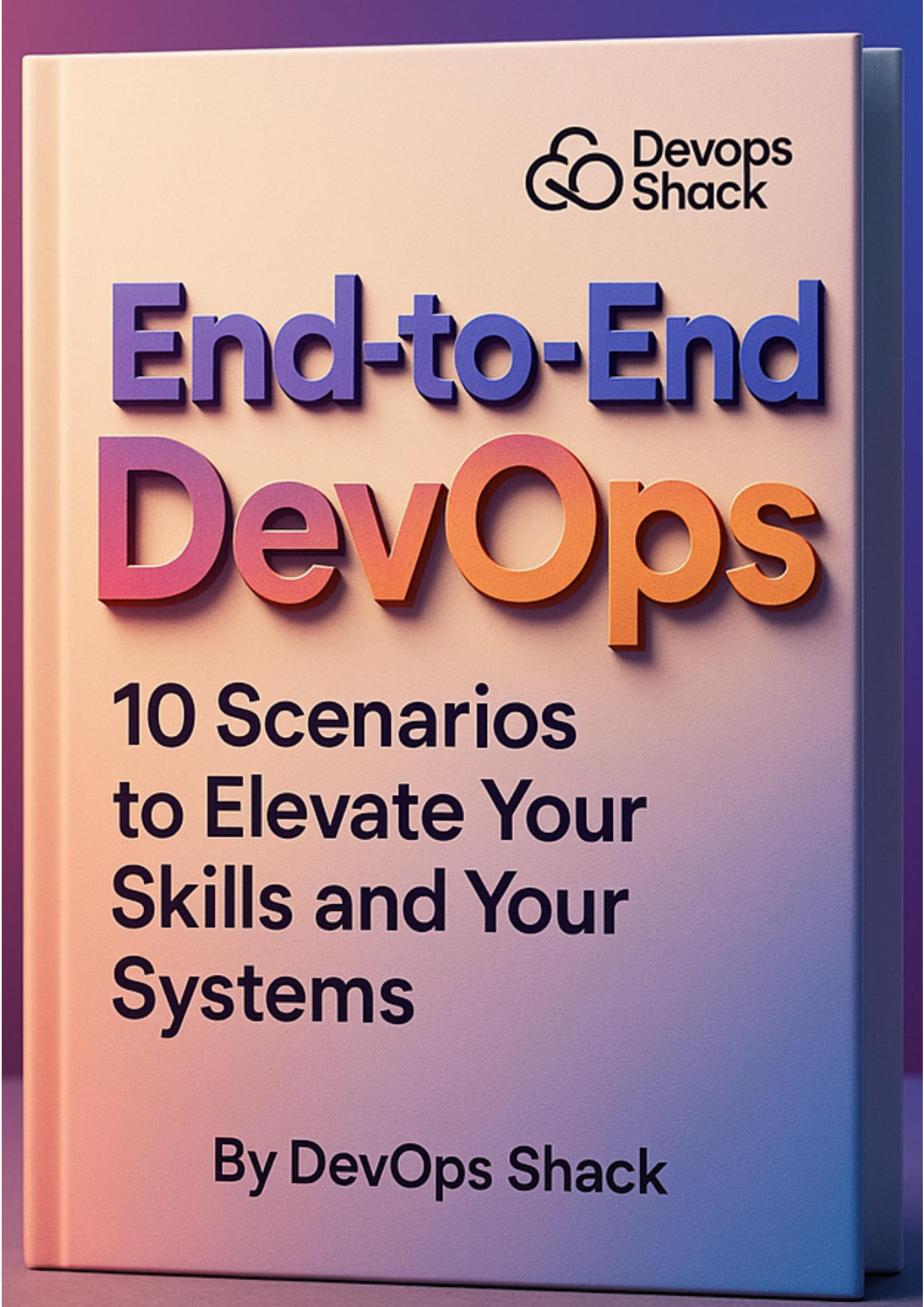
Devops Shack
End-to-End DevOps
10 Scenarios to Elevate Your Skills and Your Systems
By DevOps Shack

[Click here for DevSecOps & Cloud DevOps Course]

DevOps Shack

End-to-End DevOps:

10 Scenarios to Elevate Your Skills and Your Systems

Table of Contents

- Steps to Implement Security in CI/CD
- Best Practices for Security

6. **Implementing GitOps with Argo CD**
   - Objective
   - Steps to Implement GitOps with Argo CD
   - Best Practices for GitOps
7. **Disaster Recovery and Backup Automation**
   - Objective
   - Steps to Implement Disaster Recovery
   - Best Practices for Disaster Recovery
8. **Implementing Observability with OpenTelemetry**
   - Objective
   - Steps to Implement Observability
   - Best Practices for Observability
9. **Container Security with Image Signing and Policy Enforcement**
   - Objective
   - Steps to Implement Container Security
   - Best Practices for Container Security
10. **Multi-Cluster Kubernetes Management**
    - Objective
    - Steps to Implement Multi-Cluster Management
    - Best Practices for Multi-Cluster Management

Introduction

As organizations aim to achieve faster development cycles, continuous delivery, and enhanced security, the role of a DevOps engineer becomes critical. This document delves into five key scenarios every DevOps engineer must master, covering automation, infrastructure as code, monitoring, scaling, and security.

We will explore:

1. Automating CI/CD Pipelines.
2. Infrastructure as Code (IaC) Implementation.
3. Monitoring and Incident Management.
4. Scaling Applications with Kubernetes.
5. Security and Compliance in the DevOps Pipeline.

Each section includes implementation details, code examples, and best practices to help you master these scenarios.

Scenario 1: Automating CI/CD Pipelines

Objective

Automating Continuous Integration (CI) and Continuous Deployment (CD) pipelines is crucial to streamline software delivery and ensure high-quality releases. In this section, we will create a multi-stage CI/CD pipeline using Jenkins, Docker, and Kubernetes.

Steps to Implement CI/CD Pipeline:

1. **Set Up Jenkins:** First, set up Jenkins on an EC2 instance or a Kubernetes pod.

Jenkins Setup Script (EC2 example):

```
sudo apt update

sudo apt install openjdk-11-jdk -y

wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -

sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'
```

```
sudo apt update
sudo apt install jenkins -y
sudo systemctl start jenkins
sudo systemctl enable jenkins
```

2. **Install Required Plugins**: Install the necessary Jenkins plugins like **Git**, **Docker Pipeline**, and **Kubernetes CLI Plugin** for integration.
3. **Create a Jenkinsfile for Your Application:** A Jenkinsfile defines the steps in your CI/CD pipeline. Here is an example for a simple Java application with Docker.

Example Jenkinsfile:

```
pipeline {
  agent any
  stages {
    stage('Clone Repository') {
      steps {
        git 'https://github.com/your-repo/your-app.git'
      }
    }
    stage('Build Docker Image') {
      steps {
        script {
          docker.build('your-app-image')
        }
      }
    }
    stage('Deploy to Kubernetes') {
      steps {
```

```
                sh 'kubectl apply -f k8s/deployment.yaml'
            }
        }
    }
    post {
        always {
            archiveArtifacts artifacts: '**/target/*.jar', allowEmptyArchive: true
        }
    }
}
```

4. **Deploy the Application to Kubernetes**: Define a Kubernetes deployment manifest to deploy the built image.

Sample Kubernetes deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: your-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: your-app
  template:
    metadata:
      labels:
        app: your-app
    spec:
```

```
    containers:
      - name: your-app
        image: your-app-image:latest
        ports:
          - containerPort: 8080
```

5. **Trigger the Pipeline**: Push code changes to your Git repository, and Jenkins will automatically trigger the pipeline.

Best Practices for CI/CD:

- **Use Multi-Stage Pipelines:** Define separate stages for building, testing, and deploying to avoid bottlenecks.
- **Automate Rollbacks:** Implement automatic rollback strategies in case of failures.
- **Utilize Caching:** Use Docker layer caching to speed up the build process.

Scenario 2: Infrastructure as Code (IaC) Implementation

Objective

Automating cloud infrastructure provisioning with Infrastructure as Code (IaC) ensures consistency and reduces manual errors. We'll use **Terraform** for provisioning and **Ansible** for configuration management.

Steps to Implement Infrastructure as Code:

1. **Install Terraform**: Download and install Terraform on your local machine or CI environment.

Install Terraform (Linux):

sudo apt-get update && sudo apt-get install -y gnupg software-properties-common curl

curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -

sudo apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com $(lsb_release -cs) main"

sudo apt-get update && sudo apt-get install terraform

2. **Create Terraform Configuration Files**: Define a simple Terraform configuration for deploying a Virtual Private Cloud (VPC) and an EC2 instance.

Terraform Configuration (main.tf):

```
provider "aws" {
  region = "us-west-2"
}

resource "aws_vpc" "my_vpc" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_instance" "my_instance" {
```

```
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"


  tags = {
    Name = "MyInstance"
  }
}
```

3. **Initialize and Apply Terraform Configuration**: Use the following commands to initialize and apply the Terraform configuration.

Terraform Commands:

```
terraform init
terraform apply
```

4. **Install and Use Ansible for Configuration**: Once the infrastructure is provisioned, use Ansible to install packages or configure the instance.

Ansible Playbook (playbook.yml):

```
- hosts: all
  become: yes
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
```

5. **Run the Ansible Playbook**: Run the playbook to configure the EC2 instance.

Command:

```
ansible-playbook -i inventory playbook.yml
```

Best Practices for IaC:

- **Modularize Code**: Use Terraform modules to break down infrastructure into reusable components.
- **Version Control**: Always use version control systems like Git for your IaC files to track changes.
- **Automate Testing**: Use tools like terratest or InSpec to automatically test your infrastructure code.

Scenario 3: Monitoring and Incident Management

Objective

Setting up robust monitoring and incident management is crucial for maintaining high availability in production systems. In this scenario, we'll use **Prometheus** and **Grafana** for monitoring, along with **Alertmanager** for incident management.

Steps to Implement Monitoring:

1. **Set Up Prometheus**: Install and configure Prometheus to scrape metrics from your application or infrastructure.

Prometheus Configuration (prometheus.yml):

```
global:
  scrape_interval: 15s


scrape_configs:
  - job_name: 'node_exporter'
    static_configs:
      - targets: ['localhost:9100']
```

Run Prometheus:

```
./prometheus --config.file=prometheus.yml
```

2. **Set Up Grafana for Visualization**: Install Grafana and configure it to use Prometheus as a data source for visualizing metrics.

Grafana Docker Command:

```
docker run -d -p 3000:3000 --name=grafana grafana/grafana
```

3. **Configure Alertmanager**: Set up Alertmanager to receive alerts from Prometheus and route them to email, Slack, or other platforms.

Alertmanager Configuration (alertmanager.yml):

```
global:
  smtp_smarthost: 'smtp.gmail.com:587'
  smtp_from: 'your-email@gmail.com'
  smtp_auth_username: 'your-email@gmail.com'
```

```
  smtp_auth_password: 'your-password'


route:
  receiver: 'email-alert'


receivers:
  - name: 'email-alert'
    email_configs:
      - to: 'admin@example.com'
```

4. **Create Alert Rules in Prometheus**: Define alert rules in Prometheus to trigger alerts based on metrics.

Prometheus Alert Rule:

```
groups:
  - name: example
    rules:
      - alert: InstanceDown
        expr: up == 0
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: "Instance {{ $labels.instance }} down"
          description: "{{ $labels.instance }} of job {{ $labels.job }} has been down for more than 5 minutes."
```

5. **Respond to Alerts**: When an incident occurs, Alertmanager will trigger notifications. Make sure to have an incident response plan in place.

Best Practices for Monitoring:

- **Use Service-Level Objectives (SLOs)**: Define clear SLOs and metrics that align with business objectives.
- **Alert on Symptoms, Not Causes**: Focus on customer-facing symptoms rather than internal issues.
- **Automate Incident Responses**: Automate responses where possible, such as auto-scaling or restarting failed services.

Scenario 4: Scaling Applications with Kubernetes

Objective

As applications grow, scaling becomes critical to handling increased user load. In this scenario, we'll use **Kubernetes Horizontal Pod Autoscaler (HPA)** to scale a web application based on CPU usage.

Steps to Implement Kubernetes Scaling:

1. **Deploy the Application**: Start by deploying a web application to a Kubernetes cluster.

Deployment Manifest (deployment.yaml):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web-app
  template:
    metadata:
      labels:
        app: web-app
    spec:
      containers:
      - name: web-app
        image: web-app-image
        resources:
          requests:
            cpu: "100m"
```

```
        limits:
          cpu: "200m"
        ports:
        - containerPort: 80
```

2. **Enable Horizontal Pod Autoscaler (HPA)**: Define an HPA to scale the application based on CPU usage.

HPA Manifest (hpa.yaml):

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: web-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web-app
  minReplicas: 2
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

3. **Apply the HPA Configuration**: Apply the HPA configuration to the cluster.

Command:

```
kubectl apply -f hpa.yaml
```

4. **Test Scaling**: Simulate high load to observe the autoscaler in action. You can use tools like **Apache JMeter** or **wrk** to load test the application.
5. **Monitor Scaling Events**: Use kubectl get hpa to monitor scaling events and ensure that the application is scaling correctly based on CPU usage.

Command:

```
kubectl get hpa
```

Best Practices for Scaling:

- **Optimize Resource Requests and Limits**: Define appropriate CPU and memory requests/limits for your pods.
- **Use Metrics Wisely**: You can scale based on other metrics like memory, custom metrics, or even external metrics.
- **Avoid Over-Scaling**: Set a reasonable maximum limit for your autoscaler to prevent unnecessary resource consumption.

Scenario 5: Security and Compliance in the DevOps Pipeline

Objective

Ensuring security and compliance is a key responsibility in the DevOps lifecycle. We'll integrate security scanning tools like **Snyk** and **SonarQube** into a CI/CD pipeline to detect vulnerabilities early in the development process.

Steps to Implement Security in CI/CD:

1. **Integrate Snyk into Jenkins Pipeline**: Use **Snyk** to scan your application dependencies for known vulnerabilities.

Add Snyk Stage to Jenkinsfile:

```
pipeline {
    agent any
    stages {
        stage('Snyk Security Scan') {
            steps {
                sh 'snyk test --all-projects'
            }
        }
    }
}
```

Snyk Command (for local testing):

```
snyk test --all-projects
```

2. **Integrate SonarQube for Code Quality and Security**: Set up **SonarQube** to analyze code quality and security vulnerabilities.

Add SonarQube Stage to Jenkinsfile:

```
pipeline {
    agent any
    stages {
        stage('SonarQube Analysis') {
            steps {
```

```
            withSonarQubeEnv('SonarQube') {
                sh 'mvn clean verify sonar:sonar'
            }
        }
    }
}
```

3. **Run Compliance Checks**: For infrastructure compliance, you can use **Terraform Sentinel** or tools like **OPA (Open Policy Agent)** to enforce policies during IaC provisioning.

Sentinel Example:

```
import "tfplan"
import "strings"

main = rule {
  strings.has_prefix(tfplan.resource_changes[*].address, "aws_instance")
}
```

4. **Automate Security Tests**: Ensure that security tests are part of every pipeline run, with automatic blocking for critical vulnerabilities.
5. **Review and Respond to Reports**: Regularly review the security reports from Snyk and SonarQube and address the critical issues in a timely manner.

Best Practices for Security:

- **Shift Left on Security**: Start security testing early in the development process to identify issues sooner.
- **Use Dependency Scanning**: Regularly scan your project dependencies for known vulnerabilities.

- **Implement Automated Security Gates**: Block deployments that fail security tests or have critical vulnerabilities.

Scenario 6: Implementing GitOps with Argo CD

Objective

Implement GitOps to continuously deploy Kubernetes applications by syncing the cluster state with a Git repository, using Argo CD for automated deployment and rollback.

Steps to Implement GitOps with Argo CD:

1. **Install Argo CD on Kubernetes**
   Command:

```
kubectl create namespace argocd
```

```
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

2. **Access the Argo CD UI**

```
kubectl port-forward svc/argocd-server -n argocd 8080:443
```

Login using:

```
kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d
```

3. **Create an Argo CD Application**
   Example application.yaml:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: my-app
  namespace: argocd
spec:
  project: default
  source:
    repoURL: 'https://github.com/your-org/your-repo.git'
    targetRevision: main
    path: k8s
```

```
  destination:
    server: 'https://kubernetes.default.svc'
    namespace: default
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

Apply:

```
kubectl apply -f application.yaml
```

Best Practices for GitOps:

- Keep separate repositories for infrastructure and applications.
- Enforce branch protection and PR reviews before changes.
- Enable Argo CD RBAC and SSO for secure access.

Scenario 7: Disaster Recovery and Backup Automation

Objective

Automate Kubernetes backups for disaster recovery, ensuring quick restoration of workloads and persistent data using Velero.

Steps to Implement Disaster Recovery:

1. **Install Velero**

```
velero install \
 --provider aws \
 --plugins velero/velero-plugin-for-aws:v1.5.0 \
 --bucket my-backup-bucket \
 --backup-location-config region=us-west-2 \
 --secret-file ./credentials-velero
```

2. **Schedule Automated Backups**

```
velero schedule create daily-backup \
 --schedule "0 1 * * *" \
 --include-namespaces default
```

3. **Restore from Backup**

```
velero restore create --from-backup daily-backup-2025-08-13
```

Best Practices for Disaster Recovery:

- Regularly test restores in staging environments.
- Encrypt backup storage and rotate credentials.
- Document RTO/RPO and make them business-aligned.

Scenario 8: Implementing Observability with OpenTelemetry

Objective

Collect, process, and visualize logs, metrics, and traces using OpenTelemetry, Jaeger, and Grafana for full-stack observability.

Steps to Implement Observability:

1. **Deploy OpenTelemetry Collector**
   Example otel-collector.yaml:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: otel-collector-config
  namespace: observability
data:
  otel-collector-config.yaml: |
    receivers:
      otlp:
        protocols:
          grpc:
          http:
    exporters:
      logging:
      jaeger:
        endpoint: jaeger:14250
        insecure: true
    service:
      pipelines:
        traces:
          receivers: [otlp]
          exporters: [jaeger, logging]
```

2. **Deploy Jaeger for Tracing**

```
kubectl create namespace observability
kubectl apply -f jaeger-all-in-one.yaml
```

3. **Visualize in Grafana**
    - Add Jaeger and Prometheus as data sources.
    - Create dashboards to correlate latency, errors, and traffic.

Best Practices for Observability:

- Instrument critical services with OpenTelemetry SDKs.
- Use sampling to control storage and performance impact.
- Correlate metrics, logs, and traces for faster root-cause analysis.

Scenario 9: Container Security with Image Signing and Policy Enforcement

Objective

Ensure only trusted container images run in Kubernetes by implementing image signing and enforcing security policies.

Steps to Implement Container Security:

1. **Sign Images with Cosign**

```
cosign generate-key-pair
cosign sign --key cosign.key your-registry.com/your-app:latest
```

2. **Verify Signatures**

```
cosign verify --key cosign.pub your-registry.com/your-app:latest
```

3. **Enforce Policy with Kyverno**
   Example verify-image-policy.yaml:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: verify-signed-images
spec:
  validationFailureAction: enforce
  rules:
    - name: verify-signature
      match:
        resources:
          kinds:
            - Pod
      verifyImages:
        - image: "your-registry.com/*"
```

```
      key: "cosign.pub"
```

Apply:

```
kubectl apply -f verify-image-policy.yaml
```

Best Practices for Container Security:

- Maintain a private, trusted registry.
- Rotate signing keys periodically.
- Scan images for vulnerabilities before signing.

Scenario 10: Multi-Cluster Kubernetes Management

Objective
Manage and monitor multiple Kubernetes clusters using Rancher for centralized governance.

Steps to Implement Multi-Cluster Management:

1. **Deploy Rancher**

```
docker run -d --restart=unless-stopped \
 -p 80:80 -p 443:443 \
 rancher/rancher:latest
```

2. **Import Existing Clusters**
    - Log into Rancher UI.
    - Use “Import Cluster” option and apply the generated YAML on target clusters.
3. **Centralized Policy and Monitoring**
    - Configure global RBAC roles.
    - Integrate Prometheus and Grafana for multi-cluster monitoring.

Best Practices for Multi-Cluster Management:

- Keep all clusters aligned on Kubernetes versions.
- Use consistent RBAC and network policies.
- Automate drift detection and remediation.

Conclusion

Mastering the ten scenarios outlined in this guide is not just about learning tools and commands — it's about cultivating the mindset, discipline, and adaptability required to thrive in the modern DevOps landscape. Each scenario, from CI/CD automation to multi-cluster Kubernetes governance, represents a critical pillar in building a resilient, scalable, and secure software delivery ecosystem.

In **Scenarios 1 and 2**, we explored the foundation of DevOps — **automation** and **Infrastructure as Code**. Automation removes manual bottlenecks, while IaC ensures environments are reproducible, consistent, and version-controlled. Together, they form the backbone of every high-performing DevOps organization, enabling teams to ship faster without sacrificing quality.

Scenarios 3 and 4 moved us into operational excellence — monitoring, incident management, and intelligent scaling. These are the skills that keep systems alive under pressure. Real-world DevOps is as much about preventing outages as it is about reacting to them, and the combination of proactive observability with automated scaling allows teams to sustain growth without sacrificing user experience.

Scenario 5 reminded us that speed without security is a recipe for disaster. Embedding security and compliance into every stage of the pipeline — shifting left — transforms security from a last-minute checklist into a continuous, automated safeguard. This not only prevents vulnerabilities from reaching production but also helps organizations meet regulatory and business compliance requirements without slowing down delivery.

The **next five scenarios (6–10)** elevate DevOps maturity from efficient pipelines to strategic, enterprise-grade capabilities. **GitOps with Argo CD** provides a single source of truth and hands-free synchronization of infrastructure and applications. **Disaster recovery automation** ensures that data and workloads can survive the worst failures with minimal downtime. **OpenTelemetry-driven observability** breaks down silos by unifying logs, metrics, and traces for faster root-cause analysis. **Image signing and policy enforcement** ensure that only trusted code reaches production, closing a critical supply chain security gap. Finally, **multi-cluster Kubernetes management** enables organizations to scale DevOps principles across geographies, business units, and hybrid environments without losing control.

By mastering these scenarios, you position yourself as more than a tool user — you become a **DevOps architect** capable of designing systems that are not just functional, but also **secure, observable, resilient, and future-ready**. These capabilities are no longer optional in competitive, fast-moving industries; they are the baseline expectations of any modern software delivery team.

The journey doesn't end here. DevOps is not a static skillset — it's a continuously evolving discipline. New technologies, patterns, and challenges will emerge: AI-driven pipeline optimization, self-healing infrastructure, policy-as-code at scale, serverless observability, and beyond. But with the scenarios in this guide under your belt, you'll have the solid technical and strategic foundation to adapt to whatever the future brings.

In the end, DevOps is less about the tools and more about the culture of **collaboration, automation, and continuous improvement**. The tools and techniques here are powerful, but their true value lies in how they empower teams to deliver better software, faster, and with greater confidence. Whether you're working on a small startup product or a global enterprise platform, these scenarios provide a proven blueprint for success in a world where speed, stability, and security must coexist.

Your next step is clear: pick a scenario, implement it end-to-end in your environment, learn from the process, and then iterate. Over time, these scenarios won't just be individual skills — they'll become part of your instinct as a DevOps professional, enabling you to design, deliver, and defend production systems like a pro.