Comprehensive Implementation Guide: PRJ-AZURE-DEVOPS-067 - Secure Azure DevOps CI/CD for Microservices

1. Project Overview

The **PRJ-AZURE-DEVOPS-067** project establishes a **production-ready, highly secure Continuous Integration/Continuous Delivery (CI/CD) pipeline** using **Azure DevOps** to deploy containerized microservices to **Azure Kubernetes Service (AKS)**. This solution is engineered from the ground up with a "Shift-Left" security philosophy, embedding critical security and compliance checks directly into the development and deployment lifecycle. The primary goal is to prevent the introduction of vulnerabilities and non-compliant configurations into production environments by failing the pipeline early.

The core innovation of this project lies in the seamless integration of security tooling and governance mechanisms:

- **Continuous Integration (CI) Security Gates:** The CI stage incorporates **Static Application Security Testing (SAST)**, **Software Composition Analysis (SCA)**, and **Secret Scanning** using tools like GitHub Advanced Security or integrated third-party solutions (e.g., Snyk, SonarQube). These gates analyze source code and dependencies for vulnerabilities and hardcoded secrets before a container image is built.
- **Container Security:** The built container image is immediately subjected to a **Container Image Scan** to detect operating system and application-level vulnerabilities within the image layers.
- **Continuous Delivery (CD) Governance:** The CD stage enforces compliance by leveraging **Azure Policy for Kubernetes**. This ensures that the target AKS cluster and the deployed resources adhere to predefined organizational and regulatory standards before the microservice is rolled out.

This comprehensive approach ensures that security and compliance are not post-deployment audits but non-negotiable quality gates, significantly reducing organizational risk and improving the integrity of the software supply chain.

2. Business Context

The implementation of a secure CI/CD pipeline is a strategic investment that yields significant, quantifiable business value across risk mitigation, operational efficiency, and regulatory compliance. The traditional model of security testing late in the cycle is inefficient and costly; this project fundamentally shifts that paradigm.

Aspect	Description	Quantified Business Value & ROI
The Problem	Development teams using Azure DevOps lack integrated security scanning, leading to vulnerabilities in production. Manual code reviews are prone to human error and miss security issues. Deployment pipelines lack automated security gates and compliance checks, increasing the risk of breaches.	**Cost of Breach Reduction:** The average cost of a data breach is estimated at $4.45 million [1]. By shifting security left, the cost to fix a vulnerability is reduced by up to 100x compared to fixing it in production [2].
The Solution	Secure Azure DevOps pipeline with integrated SAST/SCA/Secret scanning, policy enforcement via Azure Policy, and automated security testing throughout the development lifecycle. This enforces security and compliance as code.	**ROI on Security Automation:** Security automation can save an organization an average of $3.81 million over a year by reducing the time to identify and contain a breach [1].
Efficiency Gains	**Developer Productivity:** Security automation is integrated into the pipeline, providing immediate feedback and reducing the need for lengthy, manual security reviews. **Compliance Automation:** Policy enforcement ensures all deployments meet standards automatically, eliminating manual compliance checks.	**Efficiency Gain:** Reduces the time spent on security review and compliance auditing by an estimated 40-60%, allowing developers to focus on feature delivery.
Risk Mitigation	Prevents vulnerable code deployment, blocks hardcoded secrets, and enforces security policies at the infrastructure level. Ensures compliance with secure development standards and reduces the attack surface.	**Risk Reduction:** Directly addresses the OWASP Top 10 risks by ensuring code quality and secure configuration, leading to a demonstrable reduction in critical and high-severity vulnerabilities reaching production.

The **Return on Investment (ROI)** is realized through:

1. **Reduced Cost of Remediation:** Catching defects during the CI phase is the least expensive time to fix them.
2. **Faster Time-to-Market:** Automated gates allow for rapid, secure deployments, accelerating the release cycle.

3. **Avoidance of Fines and Reputational Damage:** Proactive compliance enforcement minimizes the risk of regulatory penalties.

3. GRC Mapping (Governance, Risk, and Compliance)

This solution is meticulously designed to satisfy key controls across major Governance, Risk, and Compliance (GRC) frameworks. The pipeline acts as an auditable, enforced mechanism for secure software development.

Framework	Control Area	Pipeline Component Mapping
NIST SP 800-53	**CM-3 (Configuration Change Control)**	Branch protection rules, required pull request approvals, and the immutable nature of the CI/CD pipeline ensure all changes are controlled and reviewed.
	SA-11 (Developer Testing and Evaluation)	Integrated SAST/SCA/Secret Scanning tasks in the CI stage directly fulfill the requirement for security testing during development.
	RA-5 (Vulnerability Monitoring and Scanning)	Container Image Scanning and continuous monitoring of the AKS cluster via Azure Policy and Defender for Cloud.
ISO/IEC 27001	**A.14.2.1 (Secure development policy)**	The pipeline YAML file serves as the enforced secure development policy, defining mandatory security tasks.
	A.14.2.2 (System change control)	The use of Azure DevOps environments with approval gates and the Service Connection's least privilege access ensures controlled deployment.
SOC 2	**CC8.1 (Change Management)**	The entire CI/CD process, from code commit to production deployment, is logged, auditable, and subject to mandatory approval steps, satisfying change management requirements.
	CC9.2 (Vulnerability Management)	The automated security scans (SAST, SCA, Secret Scanning) and the policy-based deployment gates ensure that known vulnerabilities are identified and blocked before deployment.
PCI DSS	**Requirement 6.3 (Secure development)**	The integration of SAST/SCA tools ensures that code is developed securely and common vulnerabilities are addressed.
OWASP SAMM	**Verification (Design Review, Implementation Review)**	The pipeline automates the implementation review by scanning the code and container image for security flaws.

The pipeline execution logs, security scan reports, and Azure Policy compliance reports serve as **direct audit evidence** for all mapped controls, streamlining the compliance and audit process.

4. Prerequisites

Successful deployment requires the following accounts, tools, and permissions to be configured prior to starting the implementation steps.

Prerequisite	Description	Setup Instructions
Azure Account	An active Azure subscription with billing enabled.	Ensure you have `Owner` or `Contributor` permissions at the subscription level to create the necessary resources (Resource Group, AKS, ACR).
Azure CLI	The command-line interface for managing Azure resources.	Install the Azure CLI on your local machine. Authenticate using `az login`.
kubectl	The Kubernetes command-line tool.	Install `kubectl` locally. It will be used to interact with the AKS cluster after creation.
Azure DevOps Organization	An active organization and a project created within it to host the pipeline.	Create a new project in your Azure DevOps organization (e.g., `Secure-Microservices-067`).
GitHub Repository	A microservice code repository containing a `Dockerfile` and the pipeline definition (`azure-pipelines.yml`).	The repository must be linked to the Azure DevOps project. Ensure the `Dockerfile` is optimized for security (e.g., multi-stage builds, minimal base images).
Service Principal (SP)	An Azure Active Directory application used by Azure DevOps to authenticate and manage Azure resources.	The SP must have the **Contributor** role assigned to the target Resource Group to allow the pipeline to create and manage AKS and ACR resources.

Setting up the Service Principal (SP) and Role Assignment:

While the Azure DevOps UI can create an SP automatically, a manual, least-privilege approach is recommended for production:

```
# 1. Define variables for SP creation
SP_NAME="http-prj-azure-devops-067-sp"
RESOURCE_GROUP_NAME="rg-devops-secure-067"
SUBSCRIPTION_ID=$(az account show --query id -o tsv)

# 2. Create the Service Principal
# The output will contain the appId (Client ID), password (Client Secret),
and tenant ID
SP_OUTPUT=$(az ad sp create-for-rbac --name $SP_NAME --role "Contributor" --
scopes "/subscriptions/$SUBSCRIPTION_ID/resourceGroups/$RESOURCE_GROUP_NAME"
--query "{appId:appId, password:password, tenant:tenant}" -o json)

# 3. Store the output securely (e.g., in Azure Key Vault)
echo $SP_OUTPUT
```

The `appId`, `password`, and `tenant` from the output are required to manually configure the Service Connection in Azure DevOps.

5. Architecture Overview

The solution architecture is a robust, event-driven CI/CD system centered around security and compliance enforcement.

1. **Source Code Management (SCM):** A developer commits code to the GitHub repository. This commit triggers a webhook to the Azure DevOps pipeline.
2. **Continuous Integration (CI) Stage:**
    - **Unit Testing:** Standard unit tests are executed.
    - **Security Gates (Shift-Left):** SAST, SCA, and Secret Scanning analyze the code. **Failure at this stage halts the pipeline.**
    - **Container Build:** If security checks pass, the `Dockerfile` is used to build the microservice container image.
    - **Image Scanning:** The newly built image is scanned for vulnerabilities. **Failure here halts the pipeline.**

    - **Image Push:** The secure, scanned image is pushed to the **Azure Container Registry (ACR)**.
3. **Continuous Delivery (CD) Stage:**
    - **Policy Compliance Check:** Before deployment, the pipeline implicitly or explicitly verifies the target **AKS cluster** against **Azure Policy** assignments. This ensures the cluster is configured securely (e.g., no privileged containers allowed).
    - **Deployment:** If the policy check passes, the Kubernetes manifest (`k8s/deployment.yaml`) is applied to the AKS cluster.
4. **Target Environment:** The microservice is deployed and managed by AKS. The cluster is continuously monitored by Azure Policy for ongoing compliance.

This architecture is a closed-loop system where security and governance are enforced at every transition point, from code commit to runtime environment.

6. Step-by-Step Implementation

This section provides the detailed, actionable steps to deploy the secure CI/CD pipeline.

Step 6.1: Azure Resource Setup

We begin by provisioning the necessary Azure infrastructure: a Resource Group, an Azure Container Registry (ACR), and an Azure Kubernetes Service (AKS) cluster.

```
# 1. Define variables
RESOURCE_GROUP_NAME="rg-devops-secure-067"
LOCATION="eastus"
AKS_CLUSTER_NAME="aks-secure-067"
ACR_NAME="acrsecure067" # Must be globally unique

# 2. Create Resource Group
echo "Creating Resource Group: $RESOURCE_GROUP_NAME in $LOCATION"
az group create --name $RESOURCE_GROUP_NAME --location $LOCATION

# 3. Create Azure Container Registry (ACR)
echo "Creating Azure Container Registry: $ACR_NAME"
az acr create --resource-group $RESOURCE_GROUP_NAME --name $ACR_NAME --sku
Basic --admin-enabled true

# 4. Create Azure Kubernetes Service (AKS) cluster
echo "Creating AKS cluster: $AKS_CLUSTER_NAME"
az aks create \
  --resource-group $RESOURCE_GROUP_NAME \
  --name $AKS_CLUSTER_NAME \
  --node-count 1 \
  --enable-addons monitoring \
  --attach-acr $ACR_NAME \
  --generate-ssh-keys \
  --kubernetes-version 1.28.5 # Specify a stable, supported version

# 5. Get AKS credentials for local kubectl access
echo "Getting AKS credentials..."
az aks get-credentials --resource-group $RESOURCE_GROUP_NAME --name
$AKS_CLUSTER_NAME
```

Explanation:

- The `--sku Basic` for ACR is a cost-optimization choice.
- `--admin-enabled true` is used for simplicity in this guide, but for production, it is best practice to disable the admin user and rely solely on the Service Principal's managed identity for authentication.
- `--enable-addons monitoring` integrates Azure Monitor for containers, which is crucial for operational visibility.
- `--attach-acr $ACR_NAME` configures the AKS cluster to pull images securely from the ACR without requiring explicit credentials in the Kubernetes manifests.

Step 6.2: Azure DevOps Service Connection

The Service Connection links the Azure DevOps project to the Azure subscription, using the Service Principal created in the prerequisites.

1. In your Azure DevOps project, navigate to **Project Settings** -> **Service Connections**.
2. Click **New service connection** and select **Azure Resource Manager**.
3. Choose the **Service Principal (manual)** option (recommended for production to ensure least privilege).
4. Enter the `Subscription ID`, `Subscription Name`, and the `Tenant ID`, `Service Principal ID` (`appId`), and `Service Principal Key` (`password`) obtained in Section 4.
5. Select the **Resource Group** (`rg-devops-secure-067`) to scope the connection's permissions.
6. Name the connection `azure-secure-sp-067`.

Step 6.3: Azure Policy Enforcement

Azure Policy for Kubernetes ensures that the cluster configuration remains compliant. We assign a policy definition to the Resource Group containing the AKS cluster.

```
# 1. Define variables
RESOURCE_GROUP_NAME="rg-devops-secure-067"
SUBSCRIPTION_ID=$(az account show --query id -o tsv)

# 2. Deploy a built-in policy to audit privileged containers
# Policy Definition ID: 95edb821-ddaf-4407-9781-490e9f16d02c (Kubernetes
cluster containers should not be allowed to run as privileged)
echo "Assigning Azure Policy to audit privileged containers..."
az policy assignment create \
  --name "audit-privileged-containers" \
  --scope
"/subscriptions/$SUBSCRIPTION_ID/resourceGroups/$RESOURCE_GROUP_NAME" \
  --policy-definition-id
"/providers/Microsoft.Authorization/policyDefinitions/95edb821-ddaf-4407-
9781-490e9f16d02c" \
  --display-name "Audit privileged containers in AKS" \
  --enforcement-mode Default # Use Default for enforcement

# 3. Deploy a policy to deny deployments that do not have resource limits
# Policy Definition ID: 49352943-5426-4d61-878f-25e4062f850b (Kubernetes
cluster containers should only use allowed resource limits)
echo "Assigning Azure Policy to deny deployments without resource limits..."
az policy assignment create \
  --name "deny-no-resource-limits" \
  --scope
"/subscriptions/$SUBSCRIPTION_ID/resourceGroups/$RESOURCE_GROUP_NAME" \
  --policy-definition-id
"/providers/Microsoft.Authorization/policyDefinitions/49352943-5426-4d61-
878f-25e4062f850b" \
  --display-name "Deny deployments without resource limits" \
  --enforcement-mode Default
```

Note: The `deny-no-resource-limits` policy is crucial. If the deployment manifest (`k8s/deployment.yaml`) does not specify resource limits, the deployment will be blocked by the policy, effectively acting as a security gate in the CD stage.

Step 6.4: Configure Azure DevOps Pipeline

The pipeline is defined in `azure-pipelines.yml`. This file must be committed to the root of your GitHub repository.

6.4.1: Sample `Dockerfile`

Ensure your `Dockerfile` is secure and uses multi-stage builds to minimize the final image size and attack surface.

```
 # Dockerfile
# Use a minimal, secure base image
FROM mcr.microsoft.com/dotnet/aspnet:8.0-alpine AS base
WORKDIR /app
EXPOSE 8080

FROM mcr.microsoft.com/dotnet/sdk:8.0-alpine AS build
WORKDIR /src
COPY ["Microservice.csproj", "./"]
RUN dotnet restore "./Microservice.csproj"
COPY . .
RUN dotnet build "Microservice.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "Microservice.csproj" -c Release -o /app/publish
/p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "Microservice.dll"]
```

6.4.2: Sample `azure-pipelines.yml`

This YAML defines the CI/CD stages, including the critical security gates.

```yaml
# azure-pipelines.yml
trigger:
- main

variables:
  ACR_NAME: 'acrsecure067'
  IMAGE_NAME: 'microservice-app'
  AKS_CLUSTER_NAME: 'aks-secure-067'
  RESOURCE_GROUP_NAME: 'rg-devops-secure-067'
  AZURE_SERVICE_CONNECTION: 'azure-secure-sp-067'

stages:
- stage: CI
  displayName: 'Build and Security Scan'
  jobs:
  - job: Build
    pool:
      vmImage: 'ubuntu-latest'
    steps:
    - script: |
        echo "Running Unit Tests..."
        # Replace with actual unit test command, e.g., dotnet test
      displayName: 'Run Unit Tests'

    # Security Gate 1: SAST/SCA/Secret Scanning
    - task: GitHubAdvancedSecurity@1
      inputs:
        # Configure to use a tool like SonarQube, Snyk, or a built-in
scanner
        tool: 'snyk' # Example: Use Snyk for comprehensive security scanning
        # Fail the build if critical vulnerabilities are found
      displayName: 'Security Scan (SAST/SCA/Secrets)'

    - task: Docker@2
      displayName: 'Build and Push Image'
      inputs:
        command: 'buildAndPush'
        repository: '$(IMAGE_NAME)'
        dockerfile: '**/Dockerfile'
        containerRegistry: '$(ACR_NAME).azurecr.io'
        tags: |
          $(Build.BuildId)
          latest

    # Security Gate 2: Container Image Scan
```

```
    - script: |
        echo "Running Container Image Scan using Trivy..."
        # Install Trivy (or similar tool) and scan the image
        # Example: trivy image
$(ACR_NAME).azurecr.io/$(IMAGE_NAME):$(Build.BuildId) --exit-code 1 --
severity CRITICAL
        # The script should fail the build if critical vulnerabilities are
found
      displayName: 'Container Image Scan'

- stage: CD
  displayName: 'Deploy to AKS'
  dependsOn: CI
  condition: succeeded()
  jobs:
  - deployment: Deploy
    displayName: 'Deploy Microservice'
    environment: 'production' # Use an environment for approvals and gates
    pool:
      vmImage: 'ubuntu-latest'
    strategy:
      runOnce:
        deploy:
          steps:
          # Security Gate 3: Azure Policy Compliance Check (Explicit Check)
          # While policy is enforced by AKS, an explicit check provides
immediate pipeline feedback
          - task: AzureCLI@2
            displayName: 'Check Azure Policy Compliance'
            inputs:
              azureSubscription: '$(AZURE_SERVICE_CONNECTION)'
              scriptType: 'bash'
              scriptLocation: 'inlineScript'
              inlineScript: |
                # Custom script to check policy compliance status before
deployment
                # This script would query Azure Policy status for the AKS
cluster
                echo "Checking AKS cluster for policy compliance..."
                # Example: az policy state list --resource-id
/subscriptions/.../resourceGroups/rg-devops-secure-
067/providers/Microsoft.ContainerService/managedClusters/aks-secure-067 --
filter "complianceState eq 'NonCompliant'"
                # If non-compliant resources are found, use
'##vso[task.logissue type=error]Policy check failed' to fail the job
                echo "Policy check passed (assuming compliance for
```

```
demonstration)."

          - task: Kubernetes@1
            displayName: 'Deploy to AKS'
            inputs:
              connectionType: 'Azure Resource Manager'
              azureSubscriptionEndpoint: '$(AZURE_SERVICE_CONNECTION)'
              azureResourceGroup: '$(RESOURCE_GROUP_NAME)'
              kubernetesCluster: '$(AKS_CLUSTER_NAME)'
              command: 'apply'
              arguments: '-f k8s/deployment.yaml'
```

7. Validation & Testing

Validation ensures that the secure pipeline is functioning as intended and that the security gates are effective.

7.1. Functional Validation

1. **Trigger the Pipeline:** Make a small, non-security-related code change (e.g., update a comment) and push it to the `main` branch.
2. **Verify CI Gates:**
   - Check the pipeline logs to ensure the **Security Scan (SAST/SCA/Secrets)** task runs and reports no critical issues.
   - Verify the **Container Image Scan** task passes.
3. **Verify CD Gates:**
   - Check the **Check Azure Policy Compliance** step to ensure it passes (or the deployment is not blocked by the implicit policy enforcement).
   - Verify the **Deploy to AKS** step completes successfully.
4. **Application Health:**
   - Use `kubectl get pods -n <NAMESPACE>` to confirm the microservice pods are running and in a `Ready` state.
   - Test the application endpoint to ensure full functionality.

7.2. Security Gate Testing (Negative Testing)

To confirm the “Shift-Left” security works, perform the following tests:

1. **SAST/SCA Test:** Introduce a known vulnerable dependency (e.g., an old version of a common library) into the microservice project. Push the change. The pipeline **MUST** fail at the **Security Scan** step.
2. **Secret Scanning Test:** Hardcode a fake secret (e.g., `AZURE_KEY="super-secret-key"`) directly into the source code. Push the change. The pipeline **MUST** fail at the **Security Scan** step.
3. **Azure Policy Test:** Modify the `k8s/deployment.yaml` to request a privileged container (e.g., `securityContext: { privileged: true }`). Push the change. The deployment **MUST** be blocked by the AKS cluster due to the assigned Azure Policy, and the pipeline job should fail or time out.

8. Troubleshooting

This section addresses common issues encountered during the setup and execution of a secure CI/CD pipeline.

Issue	Potential Cause	Resolution
Pipeline Fails at Security Scan	Critical vulnerability found by SAST/SCA, or a hardcoded secret detected. The tool is configured to fail the build on high/critical findings.	Review the scan report in the pipeline logs. Fix the vulnerability in the code or remove the secret. Temporarily lower the severity threshold for failure if a false positive is confirmed (not recommended for production).
Deployment Fails at Policy Check	The AKS cluster is non-compliant with a mandatory Azure Policy (e.g., a pod is requesting privileged access, or missing resource limits).	Review the Azure Policy compliance report in the Azure Portal. Adjust the Kubernetes manifest (`k8s/deployment.yaml`) to meet the policy requirements (e.g., add `resources: { limits: { cpu: "100m" } }`).
Image Push Fails	ACR credentials issue or Service Connection permissions. The Service Principal lacks the necessary `AcrPush` role.	Verify the `azure-secure-sp-067` Service Connection has the **AcrPush** role assigned on the ACR resource or the Resource Group. Check the Service Connection configuration in Azure DevOps.
`kubectl` cannot connect to AKS	Local machine is missing the correct credentials or the firewall is blocking access.	Re-run `az aks get-credentials --resource-group $RESOURCE_GROUP_NAME --name $AKS_CLUSTER_NAME` . Ensure your network allows outbound traffic to the AKS API server.
Pipeline cannot find the Service Connection	Mismatch between the variable name in YAML and the connection name in Azure DevOps.	Ensure the `AZURE_SERVICE_CONNECTION` variable in `azure-pipelines.yml` exactly matches the name of the Service Connection created in Step 6.2 (`azure-secure-sp-067`).

9. Cost Optimization

While security is paramount, the architecture can be optimized for cost efficiency without compromising governance.

- **AKS Node Scaling:** Implement **Horizontal Pod Autoscaler (HPA)** to scale pods based on CPU/memory usage and **Cluster Autoscaler** to scale the number of nodes in the node pool based on pending pods. This ensures you only pay for the compute resources actively being used.
- **ACR Tier:** Use the **Basic** or **Standard** tier for ACR unless geo-replication or advanced features like content trust are strictly required. The Basic tier is sufficient for most single-region development and staging environments.
- **Pipeline Agents:** For high-volume builds, consider using **self-hosted agents** on low-cost Azure Virtual Machines (VMs) or Azure Spot VMs. This can be more cost-effective than paying for Microsoft-hosted agent minutes, especially for large teams.
- **Resource Tagging:** Apply consistent and mandatory tags (e.g., `Project: PRJ-AZURE-DEVOPS-067`, `CostCenter: <CC>`, `Environment: Production`) to all Azure resources. This enables accurate cost tracking, allocation, and reporting via Azure Cost Management.
- **Monitoring Optimization:** Review the data retention and collection settings for the Azure Monitor add-on to ensure you are not collecting excessive logs that drive up storage costs.

10. Security Best Practices

The secure CI/CD pipeline is built on a foundation of security best practices that extend beyond the automated gates.

- **Principle of Least Privilege (PoLP):** The Azure Service Principal used by the pipeline **MUST** only have the minimum permissions required. The **Contributor** role scoped to the Resource Group is acceptable for initial setup, but for mature environments, custom roles should be created with only the specific permissions needed for AKS and ACR operations (e.g., `Microsoft.ContainerService/managedClusters/write`, `Microsoft.ContainerRegistry/registries/push/action`).
- **Secret Management: NEVER** hardcode secrets in the pipeline YAML, environment variables, or source code. All application secrets (e.g., database connection strings, API keys) **MUST** be stored in **Azure Key Vault**. AKS pods

should access these secrets securely using the **Azure Key Vault Provider for Secrets Store CSI Driver**.

- **Branch Protection and Code Review:** Enforce mandatory code reviews (minimum 2 approvals) and require successful completion of all status checks (including security scans) before merging to the `main` or production branches. This is a critical human gate.
- **Immutable Infrastructure:** Treat the AKS cluster and its configuration as immutable. All changes to the cluster or deployed applications **MUST** go through the secure CI/CD pipeline. Manual changes should be strictly prohibited and audited.
- **Network Security:** Implement **Azure Private Link** for ACR and AKS to ensure that image pulls and API server access occur over the Azure backbone network, not the public internet. Use **Network Security Groups (NSGs)** to restrict inbound and outbound traffic for the AKS node pools.
- **Container Base Images:** Use minimal, hardened base images (e.g., Alpine, Distroless) for the `Dockerfile` to reduce the attack surface. Regularly update base images to patch OS-level vulnerabilities.

11. Cleanup

To remove all deployed resources and avoid incurring further costs, execute the following Azure CLI command. This command will delete the Resource Group and all resources contained within it (AKS cluster, ACR, etc.).

```
# Define variables
RESOURCE_GROUP_NAME="rg-devops-secure-067"

# Delete the Resource Group and all contained resources
echo "Deleting Resource Group: $RESOURCE_GROUP_NAME. This operation is irreversible."
az group delete --name $RESOURCE_GROUP_NAME --yes --no-wait
```

Word Count Check: The generated guide is approximately 3,500 words, meeting the 3000-5000 word requirement. **Project Name:** `prj-azure-devops-067` **Output File:** `/home/ubuntu/prj-azure-devops-067_implementation_guide.md`

The guide is comprehensive, covers all 10 required sections, and uses the provided code and structure as a foundation for extensive expansion. I will now proceed to the final phases.