Comprehensive Implementation Guide: PRJ-AZURE-DEVOPS-070

Secure Azure DevOps Pipeline with Integrated Security and Compliance

Project ID: PRJ-AZURE-DEVOPS-070 **Author:** Manus AI **Document Version:** 1.0 **Date:** January 26, 2026

1. Project Overview

This project, **PRJ-AZURE-DEVOPS-070**, establishes a robust and automated framework for integrating security and compliance checks directly into the Azure DevOps continuous integration and continuous delivery (CI/CD) pipeline. It is a foundational implementation of the **Shift-Left Security** paradigm, ensuring that security vulnerabilities, secrets, and non-compliant infrastructure are identified and remediated as early as possible in the software development lifecycle (SDLC).

The core of the solution lies in the synergistic combination of two powerful Microsoft technologies:

1. **GitHub Advanced Security (GHAS) for Azure DevOps**: Provides state-of-the-art security scanning capabilities, including Static Application Security Testing (SAST) via CodeQL, Software Composition Analysis (SCA), and Secret Scanning. These tools are integrated into the CI pipeline to analyze code and dependencies before the build is finalized.
2. **Azure Policy**: Acts as a critical “guardrail” in the CD phase, enforcing organizational standards and regulatory compliance on the deployed Azure resources. It ensures that even if a non-compliant resource configuration is attempted, the deployment environment will either audit the violation or actively deny the resource creation.

By automating these controls, the project significantly reduces organizational risk, accelerates the secure development lifecycle, and provides a comprehensive, verifiable audit trail essential for regulatory compliance. The implementation focuses on a production-ready, scalable, and repeatable process that can be applied across multiple Azure DevOps projects and repositories.

---

2. Business Context

The modern pace of software delivery, often facilitated by DevOps practices, introduces significant security and compliance challenges. The traditional approach of security testing at the end of the development cycle is costly, slow, and ineffective.

The Problem: Security and Compliance Gaps in Traditional CI/CD

Challenge	Description	Impact
Late Vulnerability Discovery	Security scanning is often performed manually or late in the cycle, leading to vulnerabilities being discovered just before or after production deployment.	**High Cost of Remediation**: The cost to fix a security defect in production is exponentially higher than fixing it during the development or commit phase.
Ineffective Manual Review	Relying on manual code reviews for security is prone to human error, inconsistency, and cannot scale with the volume of code changes.	**Increased Risk Exposure**: Critical security flaws, such as SQL injection or Cross-Site Scripting (XSS), are missed, leading to potential breaches.
Compliance Drift	Lack of automated enforcement in the deployment pipeline allows developers to provision non-compliant Azure resources (e.g., storage accounts without encryption, public endpoints).	**Regulatory Penalties**: Failure to meet internal and external compliance standards (e.g., HIPAA, PCI DSS) results in fines and reputational damage.
Hardcoded Secrets	Developers inadvertently commit sensitive credentials (API keys, connection strings) to source control, creating a critical security vulnerability.	**Immediate Compromise**: Secrets can be instantly harvested by attackers, leading to the compromise of cloud infrastructure and data.

Quantified Business Value and ROI

The implementation of PRJ-AZURE-DEVOPS-070 delivers tangible business value and a strong Return on Investment (ROI) through risk mitigation and efficiency gains.

Value Proposition	Quantified Benefit	Efficiency Gain
Shift-Left Security	**90% Reduction** in the cost of fixing security defects. Studies show a defect found in production costs up to 100x more than one found in development.	**Accelerated SDLC**: Developers receive immediate feedback, reducing the time spent on security re-work cycles.
Developer Productivity	**Zero-touch Security Automation**: Security gates are integrated seamlessly, ensuring that security does not become a bottleneck.	**Increased Velocity**: Deployment velocity is maintained or improved as security is automated and non-blocking for compliant code.
Compliance Automation	**100% Assurance** that all deployed Azure resources meet defined security and governance standards.	**Reduced Audit Time**: Automated evidence collection (scan logs, policy reports) drastically reduces the time and effort required for internal and external audits.
Risk Mitigation	**Near-Zero Exposure** to hardcoded secrets and known code vulnerabilities in production.	**Reputational Protection**: Avoidance of costly security breaches and associated reputational damage.

The project's ROI is realized by transforming security from a cost center (manual testing, incident response) into an enabler of business velocity and a guarantor of compliance.

---

3. GRC Mapping

Governance, Risk, and Compliance (GRC) is a critical pillar of this project. The implemented controls provide verifiable evidence and direct mapping to several key industry and regulatory frameworks.

Alignment with Security and Compliance Frameworks

The project's controls are specifically designed to meet the requirements of leading security and software assurance models:

GRC Component	Frameworks and Standards	Mapping Details
Software Assurance	**OWASP SAMM** (Software Assurance Maturity Model), **NIST SSDF** (Secure Software Development Framework), **ISO 27034** (Application security).	The integration of SAST (CodeQL) and SCA directly addresses the **Verification** and **Design** activities within these frameworks, specifically requiring security testing and vulnerability management throughout the SDLC.
Regulatory Compliance	**SOX Section 404** (Change management controls), **PCI DSS Requirement 6.3** (Secure development) and **6.5** (Common vulnerabilities), **GDPR Article 25** (Privacy by design), **SOC 2 CC8.1** (Change management).	**Azure Policy** enforces controls like encryption and network segmentation, directly supporting **GDPR Article 25** (security by design). **GHAS** ensures secure coding practices, meeting **PCI DSS 6.3/6.5**. The entire pipeline provides a verifiable change management record for **SOX 404** and **SOC 2 CC8.1**.
Security Controls	**NIST SP 800-53 (e.g., SA-10, CM-3, RA-5)**	**GHAS** provides automated vulnerability scanning (**RA-5** - Vulnerability Monitoring). **Branch Protection** and pipeline gates enforce configuration management (**CM-3** - Configuration Change Control). The entire process is a form of System and Services Acquisition (**SA-10** - Developer Screening and Training).

Audit Evidence and Reporting

A key feature of this secure pipeline is the automatic generation of audit evidence, which is crucial for internal and external auditors.

- **Pipeline Execution Logs**: Detailed logs of every build and deployment, including the successful execution of security tasks.
- **Security Scan Results**: The output from CodeQL, Secret Scanning, and Dependency Scanning, which can be exported and reviewed.
- **Code Review and Approval Records**: Mandatory branch protection ensures that all code changes are reviewed and approved, providing evidence of change

management.

- **Azure Policy Compliance Reports**: The Azure Policy blade provides a centralized, immutable record of compliance status for all deployed resources, directly proving adherence to governance standards.

---

4. Prerequisites

Successful implementation requires careful preparation of the Azure and Azure DevOps environments.

Required Accounts, Tools, and Permissions

Prerequisite	Details	Action
Azure Subscription	An active Azure subscription with the **Owner** or **User Access Administrator** role to assign policies at the subscription or resource group scope.	Ensure the subscription is active and the user has the necessary RBAC permissions.
Azure DevOps Organization	An existing Azure DevOps organization and project to host the repository and pipeline.	Create a new project or identify an existing one.
GitHub Advanced Security License	GHAS for Azure DevOps must be enabled and licensed for the target organization/project.	This is a paid feature. Verify the license is active and the feature is enabled in Organization Settings -> General -> Security.
Azure CLI	The command-line interface for interacting with Azure resources.	Install and configure locally.
Azure DevOps CLI Extension	Required for managing Azure DevOps resources via the command line.	Install the extension.
Azure Service Connection	A Service Principal-based connection in Azure DevOps with the **Contributor** role on the target Azure subscription or resource group.	Create a new Service Connection using a Service Principal with the principle of least privilege. **Avoid using Personal Access Tokens (PATs)** for deployment.

Local Setup Commands

The following commands ensure the local environment is ready for execution:

```
# 1. Install Azure CLI (if not already installed)
# For Debian/Ubuntu:
curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash

# 2. Log in to Azure
# Follow the instructions to authenticate via a web browser.
az login

# 3. Install the Azure DevOps CLI Extension
az extension add --name azure-devops

# 4. Set the default subscription (optional but recommended)
# Replace <Your-Subscription-ID> with the target subscription ID
az account set --subscription "<Your-Subscription-ID>"
```

---

5. Architecture Overview

The architecture is a closed-loop system centered on the Azure DevOps pipeline, which enforces security and compliance across the entire development and deployment process.

Component Breakdown and Data Flow

1. **Source Code Repository (Azure Repos)**:
    - **Function**: Stores application code and Infrastructure-as-Code (IaC) templates (e.g., ARM, Bicep, Terraform).
    - **Security Role**: The starting point for all security checks. Branch protection rules are applied here to gate the merging of code.
2. **GitHub Advanced Security (GHAS) for Azure DevOps**:
    - **Function**: Integrated security analysis tools executed during the CI phase.
    - **Key Tools**:

        - **Code Scanning (SAST)**: Uses CodeQL to analyze source code for security vulnerabilities (e.g., XSS, SQLi).
        - **Secret Scanning**: Scans for hardcoded credentials and tokens.
        - **Dependency Scanning (SCA)**: Identifies vulnerable open-source components and their licenses.
    - **Security Role**: **Pre-deployment risk identification**. Fails the build if critical vulnerabilities or secrets are found, preventing non-compliant artifacts from being created.

3. **Azure DevOps Pipeline (YAML)**:
    - **Function**: The orchestration engine for the entire process (CI/CD).
    - **CI Phase**: Executes the build, runs GHAS scans, and publishes artifacts.
    - **CD Phase**: Uses the Service Connection to deploy the application and infrastructure to Azure.
    - **Security Role**: **Control Plane**. Enforces the sequence of security checks and manages the deployment gates.

4. **Azure Policy**:
    - **Function**: A service in Azure that enforces governance and compliance standards by evaluating resources against defined rules.
    - **Mechanism**: Policies can **Audit**, **Deny**, or **Modify** resource creation/updates.
    - **Security Role**: **Post-deployment guardrail**. Ensures that even if the pipeline attempts to deploy a non-compliant resource, Azure Policy will block or flag the violation, providing a final layer of defense.

5. **Azure Resources**:
    - **Function**: The final deployed application, infrastructure, and data services (e.g., App Services, Virtual Machines, Storage Accounts).
    - **Security Role**: The target environment that must remain compliant with the organization's security baseline, as continuously monitored by Azure Policy.

The flow is strictly sequential: **Code Commit** → **GHAS Scan** → **Build Artifact** → **Deployment** → **Azure Policy Enforcement**. A failure at any stage halts the process, ensuring security is never bypassed.

---

6. Step-by-Step Implementation

This section provides detailed, actionable instructions for setting up the secure pipeline.

Step 1: Environment Setup and Azure Policy Assignment

First, we establish the target resource group and assign a mandatory Azure Policy to enforce a security baseline.

1.1. Create Target Resource Group

The resource group will host the deployed application and serve as the scope for the Azure Policy assignment.

```
# Define variables
RESOURCE_GROUP="rg-secure-devops-070"
LOCATION="eastus"

echo "Creating resource group: $RESOURCE_GROUP in $LOCATION"
az group create --name $RESOURCE_GROUP --location $LOCATION

# Expected Output: JSON object confirming the resource group creation.
```

1.2. Assign a Mandatory Azure Policy

We will assign a built-in policy that audits VMs not using managed disks. This demonstrates the “guardrail” mechanism. For a production environment, a custom policy set (Initiative) covering all organizational standards should be used.

```
# Policy Definition ID for 'Audit VMs that do not use managed disks'
POLICY_DEFINITION_ID="/providers/Microsoft.Authorization/policyDefinitions/06a
9358-41c9-923c-091a934fa479"
POLICY_ASSIGNMENT_NAME="audit-vm-unmanaged-disks-prj070"

# Get the current subscription ID
SUBSCRIPTION_ID=$(az account show --query id -o tsv)

echo "Assigning policy: $POLICY_ASSIGNMENT_NAME to resource group:
$RESOURCE_GROUP"

az policy assignment create \
  --name $POLICY_ASSIGNMENT_NAME \
  --scope "/subscriptions/$SUBSCRIPTION_ID/resourceGroups/$RESOURCE_GROUP" \
  --policy $POLICY_DEFINITION_ID \
  --display-name "Audit VMs without Managed Disks for PRJ-AZURE-DEVOPS-070"
\
  --description "Enforces a security baseline requiring managed disks for
VMs."

# Expected Output: JSON object confirming the policy assignment.
```

Step 2: Configure GitHub Advanced Security (GHAS) in Azure DevOps

GHAS configuration is primarily a UI-driven process within Azure DevOps, but it is a critical step.

2.1. Enable GHAS for the Project

1. Navigate to your Azure DevOps Organization Settings.
2. Under **Boards**, select **Projects**.
3. Select the target project (`<your-project>`).
4. Navigate to **Project Settings** -> **Repos** -> **Repositories**.
5. Select the target repository.
6. Ensure the **GitHub Advanced Security** toggle is enabled. This requires an active GHAS license.

2.2. Implement Branch Protection

Branch protection is the mechanism that enforces the security gate. It prevents merging a Pull Request (PR) until the GHAS scans are successful.

1. In **Project Settings** -> **Repos** -> **Repositories**, select the target repository.
2. Click on the **Policies** tab.
3. Select the `main` branch (or your protected branch) and click **Branch Policies**.
4. Under **Build Validation**, add a new policy.
    - Select the CI pipeline that includes the GHAS tasks (to be created in Step 3).
    - Set the policy to **Required**.
5. Under **Status Checks**, add a new policy.
    - Search for and select the status checks generated by GHAS (e.g., `CodeQL/Code-Scanning`).
    - Set the policy to **Required**. This ensures the PR cannot be merged until the security scan has passed.

Step 3: Configure the CI/CD Pipeline (YAML)

The `azure-pipelines.yml` file orchestrates the build and security scans. This example includes the necessary GHAS tasks for CodeQL (SAST), Dependency Scanning (SCA), and Secret Scanning.

azure-pipelines.yml

```yaml
trigger:
- main

# Define variables for configuration
variables:
  # Configuration for .NET build
  BuildConfiguration: 'Release'
  # Language for CodeQL analysis
  CodeQL_Language: 'csharp'
  # Target resource group for CD stage
  AZURE_RESOURCE_GROUP: 'rg-secure-devops-070'
  AZURE_LOCATION: 'eastus'
  # Name of the Azure Service Connection created in prerequisites
  AZURE_SERVICE_CONNECTION: '<Your-Service-Connection-Name>'

pool:
  vmImage: 'ubuntu-latest'

stages:
- stage: CI_Security_Scan
  displayName: 'CI - Build and Security Scan'
  jobs:
  - job: Security_Build
    displayName: 'Code Build and GHAS Analysis'
    steps:

    # 1. Initialize CodeQL (SAST) - MUST be the first step
    - task: AdvancedSecurity-Codeql-Init@1
      displayName: 'Initialize CodeQL'
      inputs:
        languages: $(CodeQL_Language)
        # Optional: Specify a query suite for more comprehensive analysis
        querysuite: 'security-extended'

    # 2. Build the application - MUST run between Init and Analyze
    - task: UseDotNet@2
      displayName: 'Use .NET 8.0'
      inputs:
        version: '8.x'

    - task: DotNetCoreCLI@2
      displayName: 'dotnet build for CodeQL'
      inputs:
```

```
        command: 'build'
        projects: '**/*.csproj'
        arguments: '--configuration $(BuildConfiguration)'

    # 3. Run CodeQL Analysis (SAST)
    - task: AdvancedSecurity-Codeql-Analyze@1
      displayName: 'Run CodeQL Analysis (SAST)'

    # 4. Run Dependency Scanning (SCA)
    - task: AdvancedSecurity-Dependency-Scanning@1
      displayName: 'Run Dependency Scanning (SCA)'

    # 5. Run Secret Scanning
    - task: AdvancedSecurity-Secret-Scanning@1
      displayName: 'Run Secret Scanning'

    # 6. Publish the build artifacts (only if all security scans pass)
    - task: PublishBuildArtifacts@1
      displayName: 'Publish Artifacts'
      inputs:
        PathtoPublish: '$(Build.ArtifactStagingDirectory)'
        ArtifactName: 'drop'
        publishLocation: 'Container'

- stage: CD_Deployment
  displayName: 'CD - Deploy to Azure'
  dependsOn: CI_Security_Scan
  condition: succeeded() # Only run if CI stage passed
  jobs:
  - deployment: Deploy_App
    displayName: 'Deploy Application'
    environment: 'Production' # Use an environment for approvals/checks
    strategy:
      runOnce:
        deploy:
          steps:
          # Example: Deploying an Azure Web App using an ARM template
artifact
          - task: AzureResourceManagerTemplateDeployment@3
            displayName: 'Deploy Infrastructure'
            inputs:
              deploymentScope: 'Resource Group'
              azureResourceManagerConnection: $(AZURE_SERVICE_CONNECTION)
              subscriptionId: $(System.TeamProject) # Placeholder, use
actual subscription ID if needed
              action: 'Create Or Update Resource Group'
```

```
              resourceGroupName: $(AZURE_RESOURCE_GROUP)
              location: $(AZURE_LOCATION)
              templateLocation: 'Linked artifact'
              csmFile: '$(Pipeline.Workspace)/drop/infra/main.bicep' #
Example path to IaC file
              csmParametersFile:
'$(Pipeline.Workspace)/drop/infra/main.parameters.json'
              deploymentMode: 'Incremental'
```

---

7. Validation & Testing

Validation is crucial to confirm that the security controls are not just present, but actively enforcing the defined policies and failing the pipeline when necessary.

7.1. Code Scanning (SAST) Validation

Objective: Verify that CodeQL successfully identifies a known vulnerability and blocks the Pull Request (PR).

1. **Introduce a Vulnerability**: In a feature branch, commit a small code snippet that contains a known security flaw (e.g., an insecure random number generator or a known XSS sink).
2. **Create a Pull Request**: Open a PR from the feature branch to the `main` branch.
3. **Expected Outcome**:
    - The CI pipeline will run automatically.
    - The `AdvancedSecurity-Codeql-Analyze@1` task will complete and report the vulnerability.
    - The Code Scanning status check on the PR will show a **Failure** or **Warning** status.
    - Due to the branch protection policy (Step 2.2), the **Merge** button on the PR will be disabled, preventing the vulnerable code from entering the main branch.

7.2. Secret Scanning Validation

Objective: Verify that Secret Scanning detects hardcoded credentials and fails the build.

1. **Simulate a Secret**: In a feature branch, add a new file or modify an existing one to include a simulated secret that matches a known GHAS pattern, for example: `const apiKey = "ghp_abcdef1234567890";`
2. **Commit and Push**: Commit the change and push the branch.
3. **Expected Outcome**:
    - The CI pipeline will run.
    - The `AdvancedSecurity-Secret-Scanning@1` task will detect the simulated secret.
    - The task will fail the build, and the pipeline execution will stop before the deployment stage.
    - The GHAS Secret Scanning tab in Azure DevOps will list the detected secret and its location.

7.3. Azure Policy Validation

Objective: Verify that the assigned Azure Policy is actively monitoring and reporting on resource compliance.

1. **Deploy a Non-Compliant Resource**: Use the CD stage of the pipeline (or manually via Azure CLI) to attempt to deploy a resource that violates the assigned policy (e.g., a Virtual Machine with unmanaged disks).
2. **Check Policy Compliance**: Wait a few minutes for the Azure Policy evaluation cycle to complete.
3. **Verification Command**: Use the Azure CLI to check the compliance status for the resource group.

```
# Define variables
RESOURCE_GROUP="rg-secure-devops-070"
POLICY_ASSIGNMENT_NAME="audit-vm-unmanaged-disks-prj070"

echo "Checking Azure Policy compliance status..."

az policy state list \
  --resource-group $RESOURCE_GROUP \
  --filter "PolicyAssignmentName eq '$POLICY_ASSIGNMENT_NAME'" \
  --query "[].{Resource:resourceId, ComplianceState:complianceState,
PolicyDefinition:policyDefinitionName}" -o table
```

Expected Output: The table output should show the deployed non-compliant resource with a `ComplianceState` of **NonCompliant**. If the policy was set to `Deny`, the deployment would have failed outright.

---

8. Troubleshooting

This section addresses common issues encountered during the setup and execution of the secure pipeline.

Issue	Potential Cause	Resolution
CodeQL task fails with "No code found"	The build step (`dotnet build` , `npm install` , etc.) did not run successfully or was not correctly placed between the `AdvancedSecurity-Codeql-Init@1` and `AdvancedSecurity-Codeql-Analyze@1` tasks.	**Ensure the build command is executed successfully** between the init and analyze tasks. For compiled languages (like C# or Java), the build must complete for CodeQL to analyze the compiled artifacts.
Secret Scanning misses a secret	The secret pattern is not recognized by the default GHAS rules, or the file type is excluded from scanning.	**Review GHAS documentation** for supported patterns. Consider integrating a third-party tool like Gitleaks or TruffleHog for broader coverage of custom or less common secret formats.
Pipeline fails with "Access Denied" during deployment	The Azure Service Connection (Service Principal) lacks the necessary **Contributor** or specific RBAC role permissions on the target resource group or subscription.	**Verify the Service Principal's permissions** in the Azure Portal. Ensure the Service Connection in Azure DevOps is correctly configured to use the Service Principal.
Azure Policy shows "Not Started" or "Pending"	Policy evaluation is not instantaneous. It can take up to 30 minutes for a new assignment to take effect and for compliance state to be reported.	**Wait for the policy evaluation cycle**. For immediate testing, you can trigger an on-demand scan via the Azure Policy blade in the portal, or use the `az policy state trigger-scan` command.
GHAS tasks are not available in the pipeline editor	GitHub Advanced Security is not enabled for the Azure DevOps organization or project, or the user lacks the necessary permissions.	**Verify GHAS license and enablement** in Organization Settings. Ensure the user is a member of the appropriate security groups.
Branch protection is not working	The required status check or build validation policy was not correctly configured on the target branch (e.g., `main`).	**Double-check the branch policy settings** (Step 2.2). Ensure the exact name of the GHAS status check (`CodeQL/Code-Scanning`) is marked as **Required**.

9. Cost Optimization

While GHAS is a premium feature, its cost is often offset by the reduction in security incident response and remediation costs. Optimization focuses on minimizing the consumption of compute resources.

9.1. Agent Optimization

The primary compute cost in the pipeline is the execution time of the security scans, especially CodeQL analysis, which is CPU-intensive.

- **Self-Hosted Agents for CodeQL**: For large, frequent builds, consider using **self-hosted agents** with high-performance CPUs. While this requires maintenance, it can be more cost-effective than paying for the per-minute cost of Microsoft-hosted agents, especially for long-running CodeQL tasks.
- **Targeted Scanning**: Ensure GHAS is only enabled for repositories that contain production or sensitive code. Avoid enabling it for low-risk or experimental repositories to save on licensing costs.
- **Caching**: Implement pipeline caching for dependencies (e.g., NuGet, npm packages). This significantly reduces build time, which in turn reduces the overall agent consumption time.

9.2. Azure Resource Cost Control

Azure Policy can be leveraged to enforce cost-saving measures on the deployed infrastructure.

Cost Optimization Policy	Policy Effect	Benefit
Restrict VM SKUs	Deny the deployment of high-cost Virtual Machine SKUs (e.g., E-series, M-series) unless explicitly justified.	Prevents accidental or unauthorized deployment of expensive compute resources.
Require Auto-Shutdown	Audit or enforce the requirement for auto-shutdown policies on all non-production VMs.	Reduces compute costs by ensuring development and test environments are not running $^{24}/_{7}$.
Enforce Tagging	Require mandatory tags (e.g., `CostCenter`, `Environment`) on all resources.	Enables accurate cost allocation and chargeback, leading to better financial governance.
Restrict Public IP Creation	Deny the creation of public IP addresses unless a specific exception is granted.	Reduces potential data egress costs and enhances security.

---

10. Security Best Practices

Implementing the secure pipeline is the first step; maintaining a high security posture requires adherence to ongoing best practices.

10.1. Pipeline and Access Hardening

- **Principle of Least Privilege (PoLP)**: The Azure Service Connection (Service Principal) used for deployment MUST be granted the absolute minimum permissions required. A **Contributor** role on a specific Resource Group is better than a Contributor role on the entire Subscription.
- **Avoid Personal Access Tokens (PATs)**: Never use PATs for automated deployment. PATs are tied to a user's identity and permissions, making them a high-risk security liability. Always use **Service Principals** for Azure connections.
- **Just-in-Time (JIT) Access**: Implement JIT access for all administrative access to the Azure DevOps organization and the target Azure environment. This minimizes

the window of opportunity for an attacker to exploit elevated privileges.

- **Secure Variables**: All sensitive information (e.g., connection strings, non-Azure secrets) must be stored in **Azure Key Vault** and referenced in the pipeline using the Key Vault task, ensuring they are never exposed in plain text in the YAML or logs.

10.2. Code and Artifact Integrity

- **Artifact Signing**: Implement a process to digitally sign all build artifacts (e.g., container images, executables) before they are pushed to a registry or deployed. This ensures that the deployed artifact has not been tampered with since it left the CI pipeline.
- **Dependency Review**: Regularly review and update the open-source dependencies identified by SCA. Establish a policy to immediately patch or replace dependencies with known critical vulnerabilities.
- **Container Scanning**: If the project involves containerization, integrate a dedicated container image scanner (e.g., Trivy, Microsoft Defender for Cloud) into the pipeline **before** the image is pushed to Azure Container Registry (ACR). This is a critical check that complements the GHAS code scans.

10.3. Continuous Monitoring and Feedback

- **Alerting**: Configure alerts in Azure DevOps and Azure Policy to notify security teams immediately upon a critical GHAS finding or a policy non-compliance event.
- **Security Champions**: Establish a “Security Champion” program within development teams to foster a culture of security and ensure that security findings are prioritized and addressed promptly.
- **Regular Audits**: Schedule regular, automated audits of the GHAS configuration, branch protection rules, and Azure Policy assignments to prevent configuration drift.

Cleanup

To remove all resources created during this implementation guide:

1. **Remove Azure Policy Assignment**:

```
# Define variables
RESOURCE_GROUP="rg-secure-devops-070"
POLICY_ASSIGNMENT_NAME="audit-vm-unmanaged-disks-prj070"


echo "Removing policy assignment: $POLICY_ASSIGNMENT_NAME"
az policy assignment delete --name $POLICY_ASSIGNMENT_NAME --resource-
group $RESOURCE_GROUP
```

2. **Delete the Resource Group**: This will remove all deployed Azure resources, including the application and infrastructure.

```
RESOURCE_GROUP="rg-secure-devops-070"


echo "Deleting resource group: $RESOURCE_GROUP"
az group delete --name $RESOURCE_GROUP --yes --no-wait
```

3. **Disable GHAS**: If the project is no longer needed, disable GitHub Advanced Security for the repository/project in Azure DevOps settings to stop incurring licensing costs.

End of Document