Gujjarlapudi, Rakesh (Contractor)

Altagas devsecops Strategy

Table of Contents

[1. Overview 4](#_Toc194490665)

[2. Objectives 5](#_Toc194490666)

[3. Azure DevOps Implementation Strategy 5](#_Toc194490667)

[Azure Boards 5](#_Toc194490668)

[Azure Repos 6](#_Toc194490669)

[Azure Pipelines 6](#_Toc194490670)

[Secrets Management 7](#_Toc194490671)

[Monitoring & Logging 7](#_Toc194490672)

[Security Practices 7](#_Toc194490673)

[Deployment Strategy 7](#_Toc194490674)

[Continuous Testing 8](#_Toc194490675)

[4. Disaster Recovery 9](#_Toc194490676)

[DR Strategy for Each AIS Component 9](#_Toc194490677)

[Networking and Failover Setup 10](#_Toc194490678)

[Backup and Monitoring 10](#_Toc194490679)

[Testing the DR Plan 11](#_Toc194490680)

[Automation for Recovery 11](#_Toc194490681)

[5. Azure DevOps – Azure Boards 12](#_Toc194490682)

[Use Case for Integration 12](#_Toc194490683)

[Work Item Hierarchy 12](#_Toc194490684)

[Iterations & Sprint Planning 12](#_Toc194490685)

[6. Azure DevOps – Repository Naming Convention 14](#_Toc194490686)

[Naming Convention Format 14](#_Toc194490687)

[Key Components Explained 14](#_Toc194490688)

[Examples 14](#_Toc194490689)

[Additional Best Practices 14](#_Toc194490690)

[Repository Creation Request Template 15](#_Toc194490691)

[Repository Access Levels 16](#_Toc194490692)

[7. Azure DevOps – Branching Strategy 18](#_Toc194490693)

[**Branching Strategy Overview** 18](#_Toc194490694)

[**Branch Structure** 18](#_Toc194490695)

[**Workflow** 19](#_Toc194490696)

[**Example Workflow** 19](#_Toc194490697)

[**Branch Policies** 20](#_Toc194490698)

[**Benefits** 20](#_Toc194490699)

[8. Azure DevOps – Merging Strategy 21](#_Toc194490700)

[Merging Strategy Overview 21](#_Toc194490701)

[Merging Workflow 21](#_Toc194490702)

[Merge Strategy Policies 22](#_Toc194490703)

[Merge Strategies Summary 23](#_Toc194490704)

[Benefits 23](#_Toc194490705)

[9. Azure DevOps – Tagging 24](#_Toc194490706)

[Tag Naming Convention 24](#_Toc194490707)

[Examples 24](#_Toc194490708)

[Tagging Workflow 24](#_Toc194490709)

[Tag Management Policies 25](#_Toc194490710)

[Benefits of This Tagging Strategy 25](#_Toc194490711)

[10. Azure DevOps – Pipeline Naming Conventions 27](#_Toc194490712)

[Naming Convention Format 27](#_Toc194490713)

[Key Components Explained 27](#_Toc194490714)

[Company Code (WGLH) 27](#_Toc194490715)

[Project Code (UOX) 27](#_Toc194490716)

[System Code (OG360) 27](#_Toc194490717)

[Pipeline Type 27](#_Toc194490718)

[Pipeline Purpose 27](#_Toc194490719)

[Examples 28](#_Toc194490720)

[Additional Best Practices 28](#_Toc194490721)

[1. Consistency 28](#_Toc194490722)

[2. Avoid Special Characters 28](#_Toc194490723)

[3. Documentation 28](#_Toc194490724)

[4. Versioning (Optional) 28](#_Toc194490725)

[5. Team-Specific Prefixes (Optional) 29](#_Toc194490726)

[11. Azure DevOps – CICD Pipeline (Azure Functions) 30](#_Toc194490727)

[1. Set Up Your Development Environment 30](#_Toc194490728)

[2. Push Your Code to a Repository 30](#_Toc194490729)

[3. Choose a CI/CD Tool 30](#_Toc194490730)

[4. Define the CI/CD Pipeline 30](#_Toc194490731)

[5. Implement CI/CD with Azure DevOps 31](#_Toc194490732)

[6. Best Practices 31](#_Toc194490733)

[12. Azure DevOps – CICD Pipeline (Azure LogicApps) 32](#_Toc194490734)

[1. Set Up Your Development Environment 32](#_Toc194490735)

[2. Organize Your Logic Apps Code 32](#_Toc194490736)

[3. Choose a CI/CD Tool 32](#_Toc194490737)

[4. Define the CI/CD Pipeline 32](#_Toc194490738)

[5. Implement CI/CD with Azure DevOps 33](#_Toc194490739)

[6. Best Practices 33](#_Toc194490740)

[13. Azure DevOps – SecOps 34](#_Toc194490741)

[Enable GitHub Advanced Security 34](#_Toc194490742)

[Enable Secret Scanning 34](#_Toc194490743)

[Enable Dependency Scanning 35](#_Toc194490744)

[Enable Code Scanning with CodeQL 35](#_Toc194490745)

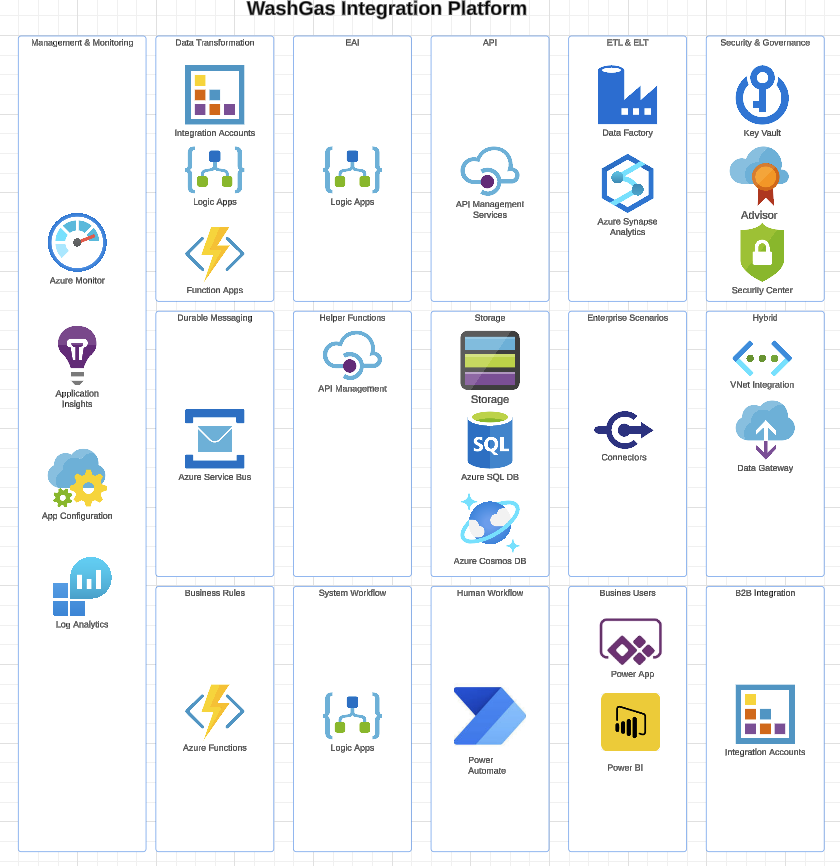
[Enable Pull Request Annotations 36](#_Toc194490746)

[Enable Monitoring & Alerting 36](#_Toc194490747)

# Overview

This document outlines the DevOps strategy for implementing a robust CI/CD pipeline, secure code practices, and project management integration leveraging Azure DevOps services at AltaGas.

It incorporates Azure Repos, Azure Pipelines, Azure Boards, and Azure Key Vault for seamless and secure application lifecycle management for Azure Integration Platform

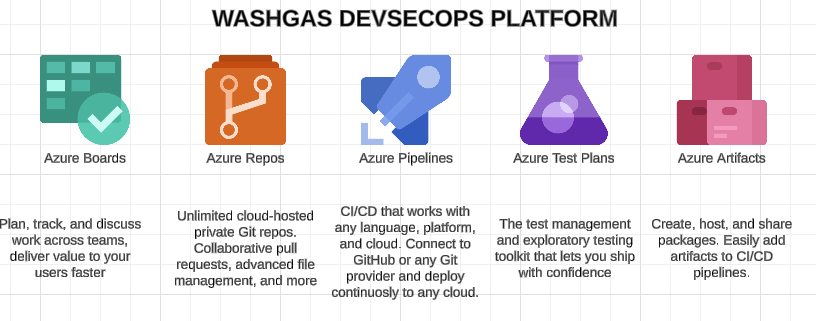


# Objectives

This is being developed as part of UOX (Utility Operations eXperience) program at WGL with the goal of standardizing these practices across the enterprise (AltaGas):

* Automate and standardize code development, testing, and deployment processes.
* Ensure secure coding practices through integrated security tools.
* Enhance collaboration and transparency using Azure Boards.
* Enable monitoring, logging, and secrets management for operational excellence.
* Deliver high-quality software consistently through CI/CD pipelines.

# Azure DevOps Implementation Strategy



## **Azure Boards**

* **Work Item Management**:
  + Define and manage work items (Epics, Features, User Stories, Bugs, and Tasks).
  + Link work items to:
    - **Branches** in Azure Repos.
    - **Pull Requests** for traceability.
* **Process Management**:
  + Use Scrum or Kanban boards for project tracking.
  + Automate work item state transitions:
    - PR creation: Work item moves to **In Progress**.
    - PR completion: Work item moves to **Resolved**.
* **Reporting & Dashboards**:
  + Use built-in dashboards to monitor progress:
    - Burndown charts.
    - Cumulative flow diagrams.
    - Sprint velocity tracking.
* **Collaboration**:
  + Enable discussions and comments directly within Azure Boards.
  + Keep stakeholders informed through notifications.

## **Azure Repos**

* **Branching Strategy**:
  + Adopt **Gitflow** branching strategy:
    - **Main Branch**: Stable branch for production.
    - **Develop Branch**: Integrates features under development.
    - **Feature Branches**: Created for each new feature or bug fix.
    - **Release Branches**: Created for release preparation.
* **Code Quality & Review**:
  + Enforce **mandatory Pull Request (PR)-based code reviews**.
  + PR validations include:
    - Code style checks.
    - CI pipeline execution.
    - Static Application Security Testing (SAST) with SonarQube.
* **Repository Policies**:
  + Branch protection rules (e.g., no direct commits to main or develop).
  + Enforce PR reviews with at least one approval.
  + Require linked work items (Azure Boards) for all PRs.

## **Azure Pipelines**

* **CI/CD Pipelines**:
  + **Continuous Integration (CI)**:
    - Trigger CI pipelines on PR creation and commits to feature or develop branches.
    - Run:
      * Code compilation and build.
      * Unit tests and code coverage validation.
      * SAST using **SonarQube**.
  + **Continuous Deployment (CD)**:
    - Automate deployments to staging and production environments.
    - Integrate Deployment Gates for compliance validation.
    - Include **DAST testing** with OWASP ZAP in staging environments.
* **Pipeline Security**:
  + Secure pipeline access with Azure Service Connections.
  + Secrets are stored in **Azure Key Vault** and accessed securely.

## **Secrets Management**

* **Azure Key Vault**:
  + Store sensitive information like API keys, credentials, and certificates.
  + Integrate Key Vault with Azure Pipelines for secure retrieval of secrets.
  + Implement RBAC (Role-Based Access Control) for least-privilege access to secrets.

## **Monitoring & Logging**

* **Centralized Monitoring**:
  + Use **Azure Monitor** for centralized logging and monitoring of DevOps pipelines.
  + Collect logs from CI/CD pipelines for issue diagnostics.
* **Alerts & Dashboards**:
  + Set up alerting rules for pipeline failures or performance issues.
  + Monitor application health post-deployment with Application Insights.

## **Security Practices**

* **Static Application Security Testing (SAST)**:
  + Integrated **SonarQube** for code analysis in CI pipelines.
* **Dynamic Application Security Testing (DAST)**:
  + Automated **OWASP ZAP** scans in staging environments before deployment.
* **Compliance Checks**:
  + Deployment Gates ensure compliance validation before the production release.
* **Secrets Management**:
  + All secrets are managed through **Azure Key Vault**.

## **Deployment Strategy**

* **Environments**:
  + **Dev/Test**: Used for feature development and testing.
  + **UAT**: Simulates production for testing and DAST validation.
  + **Production**: Live environment for end users.
* **Deployment Gates**:
  + Manual approvals or automated checks ensure compliance and quality before production deployment.
* **Blue-Green Deployments**:
  + Use blue-green or canary deployment strategies to minimize downtime and risk.

## **Continuous Testing**

* **Testing Framework**:
  + Integrate automated testing at every stage of the pipeline to ensure quality:
    - **Unit Testing**: Validate individual components and achieve code coverage.
    - **Integration Testing**: Test interactions between components and services.
    - **Functional Testing**: Validate user workflows and feature requirements.
    - **Regression Testing**: Ensure changes do not break existing functionality.
    - **Performance Testing**: Test application responsiveness, scalability, and load handling.
    - **Security Testing**: Include SAST (Static) and DAST (Dynamic) testing tools.
* **Tools Integration**:
  + **Unit Tests**: NUnit, MSTest, or xUnit.
  + **Code Quality**: SonarQube for static code analysis.
  + **Functional Tests**: Selenium, Playwright, or Cypress for automated UI testing.
  + **API Tests**: Postman or Rest Assured for API validation.
  + **Performance Testing**: JMeter or k6.
  + **Security Testing**: OWASP ZAP for DAST.
* **Test Execution in Pipelines**:
  + Tests are triggered automatically as part of CI/CD pipelines:
    - Unit tests during the build stage.
    - Integration and functional tests in the staging environment.
    - Performance and security tests before production deployment.
* **Reporting and Visibility**:
  + Generate test results and code coverage reports.
  + Integrate test results into Azure Pipelines for easy visibility.
  + Enable dashboards for real-time test performance metrics.

# Disaster Recovery

Disaster Recovery (DR) for Azure Integration Services (AIS) focuses on maintaining the availability of essential components—such as Logic Apps, Service Bus, API Management, and Event Grid—or ensuring their swift recovery in case of outages or failures. Here is a structured method for developing a DR strategy for AIS:

**Recovery Objectives**

* **Recovery Point Objective (RPO)**: Maximum acceptable data loss (e.g., 5 minutes, 1 hour).
* **Recovery Time Objective (RTO)**: Maximum time allowed to restore services after a disaster.
* Define SLAs for each integration component.

**High Availability vs Disaster Recovery**

* **High Availability**: Ensure redundancy within the same region to minimize service interruptions.
* **Disaster Recovery**: Cross-region replication and failover to recover services in case of region-wide failure.

## DR Strategy for Each AIS Component

**Azure Logic Apps**

* Use Geo-Redundancy:
  + Deploy Logic Apps in paired regions for failover.
  + Export ARM templates for Logic App definitions and deploy them to a secondary region.
* Automate redeployment using Azure DevOps pipelines or Bicep templates.
* Store all Logic App artifacts (parameters, configurations) in Azure Storage for quick recovery.

**Azure API Management**

* Use Multi-Region Deployment:
  + Enable Premium Tier for API Management to distribute the API Gateway across regions.
  + Use the Azure Traffic Manager or Front Door to route requests to the active region.
* Backup and export API configurations using Azure CLI or ARM templates.
* Set up a secondary API Management instance in a paired region for DR.

**Azure Service Bus**

* Enable Geo-Disaster Recovery (Geo-DR):
  + Use the Alias feature to fail over from a primary namespace to a secondary namespace.
  + Configure paired namespaces in different regions.
* Test failover regularly to validate recovery processes.
* Ensure messages persist to avoid data loss.

**Azure Event Grid**

* Use Resilient Event Handling:
  + Event Grid automatically retries failed event delivery.
  + Design event handlers to manage duplicate events and handle retries.
* Deploy redundant Event Grid topics and subscriptions in paired regions.

## **Networking and Failover Setup**

* Use **Azure Traffic Manager** or **Front Door** to route traffic to the active region.
* Implement DNS failover to redirect clients to the backup resources.

## **Backup and Monitoring**

* **Backup**:
  + Export Logic Apps, API configurations, and templates regularly.
  + Store backups in **Azure Storage** with Geo-Redundant Storage (GRS).
* **Monitoring**:
  + Use **Azure Monitor** and **Application Insights** to track failures and performance.
  + Set up alerts for integration failures or failover events.

## **Testing the DR Plan**

* Conduct regular **failover drills** for Logic Apps, Service Bus, and API Management.
* Validate the RPO and RTO during each drill.
* Document lessons learned and refine processes.

## **Automation for Recovery**

* Use **Infrastructure-as-Code (IaC)** with ARM/Bicep templates or Terraform for automated failover deployment.
* Set up Azure DevOps pipelines to provision DR environments quickly.

# Azure DevOps – Azure Boards

**Project Name:** **WashGas – Azure Integration Services (AIS) for Application Integration**  
**Goal:** Implement **Azure Integration Services (AIS)** to enable **seamless data exchange** and **workflow automation** between two enterprise applications.

## **Use Case for Integration**

WashGas is integrating **Application A** (such as a **OpenGrid 360 System**) with **Application B** (such as **Peoplesoft**) using **Azure Integration Services**. This integration will guarantee **real-time data synchronization, event-driven workflows, and secure API communication *or provide legacy mechanisms***.

## **Work Item Hierarchy**

| **Work Item Type** | **Example Name** | **Description** |
| --- | --- | --- |
| **Epic** | Application A ↔ Application B Integration | Enable real-time and batch data exchange between both applications. |
| **Feature** | API-based Data Sync using Azure API Management | Set up a **secure API gateway** for both applications to communicate. |
| **User Story** | Event-driven Updates via Azure Event Grid | Trigger updates in **Application B** whenever new customer data is added in **Application A**. |
| **Task** | Deploy Azure Logic Apps for Workflow Automation | Configure **Logic Apps** to automate workflow orchestration between systems. |
| **Bug** | Duplicate Data Issue in Sync Process | Investigate and fix duplicate customer records during data synchronization. |

## **Iterations & Sprint Planning**

**Sprint 1 (Feb 19 - Mar 1, 2025)**

* Develop **APIM Endpoints** to expose APIs for both applications.
* Set up **Service Bus Queues** for reliable messaging.
* Build **Logic Apps** for request validation and transformation.

**Sprint 2 (Mar 3 - Mar 15, 2025)**

* Implement **Event Grid Subscriptions** for event-driven notifications.
* Optimize **Logic Apps** for real-time data flow.
* Deploy **Azure Functions** for data enrichment.

# Azure DevOps – Repository Naming Convention

### **Naming Convention Format**

<CompanyCode>-<ProjectCode>-<SystemCode>-<RepoPurpose/Function>

### **Key Components Explained**

1. **Company Code (WGLH)**:
   * Identifies the organization (e.g., WashGas).
2. **Project Code (UOX)**:
   * Represents the specific project within the organization.
3. **System Code (OG360)**:
   * Indicates the system or application associated with the repository.
4. **Repo Purpose/Function**:
   * Describes the purpose of the repository or the specific function of the codebase (e.g., Integrations, UI, API, Services, IaC, Utils).

### **Current Repos Naming Convention**

* **WGLH-ESVC-BACKEND**:
  + A repository for back-end code developed for the eServices Replatforming Project.
* **WGLH-ESVC-FRONTEND**:
  + A repository for the front-end (UI) of the eServices Replatforming Project.
* **WGLH-ESVC-BATCHJOBS**:
  + A repository for batch jobs code for the eServices Replatforming Project.

### **Additional Best Practices**

1. **Consistency**:
   * Ensure all repositories follow the same format for easier identification and organization.
2. **Avoid Special Characters**:
   * Stick to alphanumeric characters and dashes (-) to avoid issues with tools or scripts.
3. **Documentation**:
   * Add a README file in each repository explaining its purpose and the naming convention for clarity.
4. **Versioning (Optional)**:
   * Append version numbers for major changes in repositories, if applicable (e.g., WGLH-UOX-ARM-Integrations-v2).
5. **Team-Specific Prefixes (Optional)**:
   * If multiple teams contribute, you can include a team prefix (e.g., TeamA-WGLH-UOX-ARM-Integrations).

### **Repository Creation Request Template**

**1️⃣ Requestor Details**

* **Requested By:** (Your Name)
* **Email:** (Your Email)
* **Team/Department:** (e.g., Development, QA, DevOps)
* **Date of Request:** (DD-MM-YYYY)

**2️⃣ Repository Details**

* **Repository Name:** (Enter the desired name)
* **Project Name:** (Select the project where the repository will be created)
* **Purpose:** (Brief description of what this repository is for)
* **Repository Type:**
  + Private
  + Public

**3️⃣ Branching Strategy**

* main as the default branch
* develop for development work
* Feature branches (feature/\*)
* Release branches (release/\*)
* Hotfix branches (hotfix/\*)
* Custom (Specify) \_\_\_\_\_\_\_

**4️⃣ Permissions & Access Control**

### **Repository Access Levels**

| **Role** | **Users/Groups to be Added** |
| --- | --- |
| **Project Administrators** (Full control over the repo) | (List users or AD groups) |
| **Contributors** (Can push code and create branches) | (List users or AD groups) |
| **Readers** (Read-only access) | (List users or AD groups) |
| **Build Service Accounts** (Required for CI/CD pipelines) | (List service accounts) |

**5️⃣ Branch Policies (Check Required)**

* Require minimum **1-2 reviewers** for pull requests
* Enforce **work item linking**
* Restrict **direct commits to main branch**
* Require **successful build before merging**
* Other (Specify): \_\_\_\_\_\_\_

**6️⃣ Additional Repository Setup**

* Add README.md
* Add .gitignore (Specify language/framework: \_\_\_\_\_\_ )
* Add license (Specify: \_\_\_\_\_\_ )
* Enable **pull request templates**
* Enable **issue tracking**

**7️⃣ Approval**

* **Manager Name:** (Approver's Name)
* **Manager Email:** (Approver's Email)
* **Approval Date:** (DD-MM-YYYY)

# Azure DevOps – Branching Strategy

**Branching Strategy Overview**

The strategy includes:

1. **Main Branch**: Represents the production-ready code.
2. **Development Branch**: Base for active development and feature integration.
3. **Feature Branches**: For individual features or tasks.
4. **Release Branches**: For preparing production releases.
5. **Hotfix Branches**: For urgent bug fixes in production.

**Branch Structure**

1. **main (Production Branch)**:
   * This branch contains stable, production-ready code.
   * Direct commits are not allowed; all changes come via pull requests.
2. **Development (Development Branch)**:
   * The integration branch for all feature branches.
   * Contains code that has passed initial testing but is not yet production ready.
   * Merges into main occur via pull requests after QA validation.
3. **Feature Branches**:
   * Naming Convention: feature/<task-id>-<feature-name>  
     *(e.g., feature/UOX-123-add-login-endpoint)*.
   * Created from Development.
   * Merged back into dev after code review and testing.
4. **Release Branches**:
   * Naming Convention: release/<release-version>or <environment>  
     *(e.g., release/1.0.0 or release/qa)*.
   * Created from Development when preparing for a production release.
   * Used for final testing, bug fixing, and deployment preparations.
   * Merged into both main and Development after release.
5. **Hotfix Branches**:
   * Naming Convention: hotfix/<issue-id>  
     *(e.g., hotfix/UOX-456-critical-bugfix)*.
   * Created from main to address urgent production issues.
   * Merged into both main and Development after testing.

**Workflow**

1. **Feature Development**:
   * Developers create feature branches from Development.
   * Perform unit testing and code linting in the branch.
   * Create a pull request (PR) to merge into dev.
   * Require:
     + At least one reviewer.
     + Passing CI builds (e.g., Azure Pipelines).
2. **Integration and Testing**:
   * Merge feature branches into dev.
   * Run automated tests, including integration and security tests.
3. **Release Preparation**:
   * When ready, create a release branch from Development.
   * Perform QA, bug fixes, and final changes.
   * Merge the release branch into main and Development post-release.
4. **Hotfix Workflow**:
   * Create a hotfix branch from main for urgent fixes.
   * Apply the fix, test it, and merge the branch into main and dev.

**Example Workflow**

1. **New Feature**:
   * Create feature/UOX-123-add-login-endpoint from dev.
   * Complete development, commit changes, and push to the feature branch.
   * Open a PR for feature/UOX-123-add-login-endpoint → Development.
2. **Release**:
   * Create release/qa or release/qa-1.0.0 from Development.
   * Perform final testing and prepare for deployment.
   * Merge release branch into both main and Development.
3. **Hotfix**:
   * Create hotfix/UOX-456-critical-bugfix from main.
   * Apply the fix, test, and merge into main and dev.

**Branch Policies**

1. **main**:
   * Require pull requests for all changes.
   * Require at least one reviewer.
   * Enforce successfully build and test pipelines.
2. **Development**:
   * Require pull requests for all changes.
   * Optional: Allow merging with fewer checks for faster integration.
3. **Feature/Release/Hotfix Branches**:
   * No direct commits, changes are merged through PRs.
   * CI pipelines must pass before merging.

**Benefits**

1. Ensures **code stability** in production (main branch is always deployable).
2. Allows **parallel development** of features.
3. Supports **quick hotfixes** without disrupting ongoing development.
4. Encourages **collaboration and review** through pull requests.
5. Facilitates **structured releases** for better QA and deployment practices.

# Azure DevOps – Merging Strategy

### **Merging Strategy Overview**

1. **Key Branches**:
   * main: Always contains production-ready code.
   * dev: The primary branch for integration and testing.
   * Feature/Release/Hotfix branches: Work in progress until merged.
2. **Merge Rules**:
   * **Feature → Development**: Feature branches are merged into dev after passing code review and testing.
   * **Development → Release**: A release branch is created or merged for final QA and preparation for production.
   * **Release → Main**: After successful testing, release branches are merged into main.
   * **Hotfix → Main and Development**: Hotfixes are merged into both main (for immediate production deployment) and Development (to keep the development branch up to date).

### **Merging Workflow**

#### **1. Feature Branch Merges (Feature → Dev)**

* **Trigger**: When a feature is completed.
* **Steps**:
  1. Developer creates a **pull request (PR)** from feature/<repo-purpose>/<task-id>-<feature-description> to Development.
  2. PR reviewers validate:
     + Code quality.
     + Adherence to coding standards.
     + Link to the relevant task or work item.
  3. Automated pipelines validate:
     + Build success.
     + Unit test results.
     + Static code analysis (e.g., SonarCloud, Checkmarx).
  4. After approvals and successful checks, the feature branch is merged into dev.
* **Policy**:
  1. Require at least one reviewer for PR.
  2. Disallow direct commits to Development.
  3. Merge strategy: **Squash Merge** to keep commit history clean.

#### **2. Dev to Release Merges (Development → Release)**

* **Trigger**: When preparing for a production release.
* **Steps**:
  1. Create a release/<environment>-<version> branch from Development.
  2. Perform:
     + Final integration testing.
     + Bug fixing.
     + Security and performance validation.
  3. After QA and UAT, PR created to merge the release branch into the main.
* **Policy**:
  1. Ensure successful completion of release testing before merging.
  2. Merge strategy: **Merge Commit** to preserve release history.

#### **3. Release to Main Merges (Release → Main)**

* **Trigger**: When a release branch passes all tests.
* **Steps**:
  1. Create a PR from release/<version> to main.
  2. Review and validate the changes.
  3. Deploy code from main to production.
* **Policy**:
  1. Require successful build and production deployment validation before completing the PR.
  2. Merge strategy: **Fast-Forward Merge** to keep a linear production history.

#### **4. Hotfix Merges (Hotfix → Main and Development)**

* **Trigger**: When fixing urgent production issues.
* **Steps**:
  1. Create a hotfix/<issue-id> branch from main.
  2. Apply the fix, test it, and create a PR to merge into main.
  3. Once merged into main, create another PR to merge the hotfix branch into dev.
* **Policy**:
  1. Require expedited reviews for critical fixes.
  2. Merge strategy: **Cherry-Pick** (if only certain changes are needed) or **Squash Merge**.

### **Merge Strategy Policies**

1. **Protected Branches**:
   * Protect main and Development branches from direct commits.
   * Require PRs for all changes.
2. **Pull Request Requirements**:
   * At least one reviewer for feature and hotfix branches.
   * Multiple reviewers (e.g., QA, Security) for release branches.
3. **Automated Checks**:
   * Run CI/CD pipelines on all PRs to validate builds, tests, and scans.
   * Enforce all pipelines to pass before merging.
4. **Conflict Resolution**:
   * Resolve merge conflicts in feature branches before merging to Development.
   * Use rebase (with caution) for clean commit history in non-critical branches.

### **Merge Strategies Summary**

| **Source Branch** | **Target Branch** | **Strategy** | **Purpose** |
| --- | --- | --- | --- |
| feature/<name> | Development | Squash Merge | Clean history, one commit per feature. |
| dev | release/<version> | Merge Commit | Preserve branch history for release traceability. |
| release/<version> | main | Fast-Forward Merge | Maintain a linear production history. |
| hotfix/<name> | main, dev | Squash Merge or Cherry-Pick | Quick fixes while maintaining traceability. |

### **Benefits**

1. Ensures a **stable production branch** (main).
2. Provides a **structured process** for integrating new features and fixing issues.
3. Enforces **quality checks** with mandatory PR reviews and CI/CD pipelines.
4. Maintains a **clean commit history** for easier debugging and audits.

# Azure DevOps – Tagging

### **Tag Naming Convention**

#### **Format:**

<Version>-<Environment>-<Description>

#### **Components:**

1. **<Version>**: Follows [Semantic Versioning (SemVer)](https://semver.org/) :
   * **MAJOR.MINOR.PATCH** (e.g., 1.2.3):
     + MAJOR: Increment for incompatible changes.
     + MINOR: Increment for new functionality (backward-compatible).
     + PATCH: Increment for bug fixes.
2. **<Environment>** (Optional):
   * Indicates the deployment target (e.g., dev, qa, prod).
   * Useful when managing multiple environments.
3. **<Description>** (Optional):
   * Adds clarity to the tag’s purpose (e.g., feature, hotfix).

### **Examples**

1. **Release Tags**:
   * 1.0.0-prod: First production release.
   * 1.1.0-qa: Minor release for QA testing.
   * 2.0.0-prod: Major release with breaking changes.
2. **Hotfix Tags**:
   * 1.0.1-prod-hotfix-UOX-456: Production hotfix for issue UOX-456.
   * 1.0.2-dev-hotfix: Hotfix applied to the development environment.
3. **Feature-Specific Tags** (Optional for traceability):
   * 1.2.0-feature-add-login: Marks the completion of a login feature.

### **Tagging Workflow**

#### **1. Release Tags**

* **Trigger**: After merging a release/<version> branch into main.
* **Steps**:
  1. Use the git tag command to create a versioned tag.

perl

Copy code

git tag -a 1.0.0-prod -m "Initial production release" git push origin 1.0.0-prod

#### **2. Hotfix Tags**

* **Trigger**: After a hotfix is merged into main.
* **Steps**:
  1. Increment the patch version and add the hotfix suffix.

perl

Copy code

git tag -a 1.0.1-prod-hotfix-UOX-456 -m "Critical production fix for UOX-456" git push origin 1.0.1-prod-hotfix-UOX-456

#### **3. Pre-Release Tags**

* **Trigger**: Before deployment to testing environments (optional).
* **Steps**:
  1. Use a -rc (release candidate) or environment suffix for pre-release versions.

perl

Copy code

git tag -a 1.1.0-qa -m "QA testing for version 1.1.0" git push origin 1.1.0-qa

#### **4. Feature Tags (Optional)**

* **Trigger**: At the completion of a significant feature in dev.
* **Steps**:
  1. Use a descriptive suffix to mark feature milestones.

csharp

Copy code

git tag -a 1.2.0-feature-add-login -m "Feature: Add login functionality" git push origin 1.2.0-feature-add-login

### **Tag Management Policies**

1. **Centralized Tag Creation**:
   * Tags should be created and pushed by release managers or lead developers.
   * Automate tagging in CI/CD pipelines for consistency.
2. **Protection and Governance**:
   * Lock tags in Azure Repos to prevent accidental deletion:
     + Navigate to **Repos > Tags > Settings**.
     + Enable tag protection policies.
3. **Automation**:
   * Use CI/CD pipelines to create and push tags for releases and deployments.
   * Example YAML snippet for tagging:

yaml

Copy code

steps: - script: | git tag -a $(Build.BuildId)-prod -m "Automated tag for production release" git push origin $(Build.BuildId)-prod displayName: "Tag Release"

1. **Documentation**:
   * Maintain a CHANGELOG.md or release notes with each tag for clarity.

### **Benefits of This Tagging Strategy**

1. **Version Traceability**:
   * Links tags to specific features, hotfixes, or releases.
2. **Deployment Clarity**:
   * Ensures environments are tagged appropriately for audit and rollback.
3. **Standardization**:
   * Consistent naming conventions reduce confusion and streamline CI/CD.

# Azure DevOps – Pipeline Naming Conventions

## **Naming Convention Format**

<CompanyCode>-<ProjectCode>-<SystemCode>-<PipelineType>-<PipelinePurpose>

## **Key Components Explained**

### **Company Code (WGLH)**

* Identifies the organization (e.g., WashGas).

### **Project Code (UOX)**

* Represents the specific project within the organization.

### **System Code (OG360)**

* Indicates the system or application associated with the pipeline.

### **Pipeline Type**

* Defines the category of the pipeline, such as:
  + **CI**: Continuous Integration pipeline.
  + **CD**: Continuous Deployment pipeline.
  + **CI-CD**: Combined CI/CD pipeline.
  + **Build**: Build pipeline.
  + **Release**: Release pipeline.
  + **Test**: Testing pipeline.

### **Pipeline Purpose**

* Describes the specific function or purpose of the pipeline, such as:
  + **Deploy**: Deployment pipeline.
  + **Validate**: Validation or testing pipeline.
  + **Scan**: Security or compliance scanning.
  + **Infrastructure**: Infrastructure provisioning pipeline.
  + **Artifacts**: Artifact generation and publishing pipeline.

## **Examples**

| **Pipeline Name** | **Description** |
| --- | --- |
| WGLH-UOX-ARM-CI-Build | CI pipeline for building the ARM system. |
| WGLH-UOX-ARM-CD-Deploy | CD pipeline for deploying the ARM system. |
| WGLH-UOX-ARM-CI-CD-FullWorkflow | Full CI/CD pipeline for the ARM system. |
| WGLH-UOX-ARM-Test-Validate | Testing pipeline for validating ARM components. |
| WGLH-UOX-ARM-Release-Deploy | Release pipeline for deploying ARM artifacts. |
| WGLH-UOX-ARM-CI-Scan | Security scanning pipeline for ARM. |
| WGLH-UOX-ARM-Infrastructure-Setup | Infrastructure provisioning pipeline. |

## **Additional Best Practices**

### **1. Consistency**

* Ensure all pipelines follow the same naming format to enable easy identification and organization.

### **2. Avoid Special Characters**

* Use only alphanumeric characters and dashes (-) to avoid compatibility issues with tools and scripts.

### **3. Documentation**

* Include a clear description of each pipeline’s purpose in the Azure DevOps pipeline settings.

### **4. Versioning (Optional)**

* Append version numbers for major changes if needed.
  + Example: WGLH-UOX-ARM-CI-Build-v2

### **5. Team-Specific Prefixes (Optional)**

* If multiple teams contribute, add a team prefix.
  + Example: TeamA-WGLH-UOX-ARM-CI-Build

# Azure DevOps – CICD Pipeline (Azure Functions)

To implement a **CI/CD pipeline for Azure Functions**, you can use tools like Azure DevOps to automate the build, testing, and deployment process. Here's a step-by-step guide:

### **1. Set Up Your Development Environment**

* Develop your Azure Function using **Visual Studio**, **Visual Studio Code**, or other IDEs.
* Ensure you have the **Azure Functions Core Tools** installed for local development and debugging.

### **2. Push Your Code to a Repository**

* Store your Azure Functions code in a version control system like **GitHub**, **Azure Repos**, or **Bitbucket**.

### **3. Choose a CI/CD Tool**

* **Azure DevOps**

### **4. Define the CI/CD Pipeline**

#### **4.1 Continuous Integration (CI)**

1. **Build Pipeline**
   * Trigger the pipeline on a new commit or pull request.
   * Tasks in the pipeline:
     + Restore dependencies (e.g., npm install, dotnet restore).
     + Build the function app (e.g., dotnet build, npm run build).
     + Run unit tests.
2. **Publish Artifacts**
   * Publish the build output as a pipeline artifact.

#### **4.2 Continuous Deployment (CD)**

1. **Release Pipeline**
   * Deploy the function app to Azure using one of the following:
     + Azure CLI (az functionapp deployment source config-zip)
     + Azure Functions App Deployment task (Azure DevOps)
     + GitHub Actions deployment workflow.
2. **Environment Management**
   * Use **staging** and **production** environments.
   * Store environment-specific settings in **App Settings** or use **Azure Key Vault**.

### **5. Implement CI/CD with Azure DevOps**

#### **5.1 Create a Build Pipeline**

1. Trigger the pipeline on code pushes.
2. Steps in the pipeline:
   * Checkout code.
   * Restore dependencies.
   * Build the function app.
   * Run tests.
   * Publish artifacts.

#### **5.2 Create a Release Pipeline**

1. Use the artifact from the build pipeline.
2. Deploy to Azure using the **Azure App Service Deploy** task.
3. Example deployment step:

### **6. Best Practices**

1. **Parameterize Deployments**
   * Use variables for app name, resource group, and environment settings.
2. **Secure Secrets**
   * Store sensitive information like connection strings and keys in **Azure Key Vault** or pipeline secrets.
3. **Use Slots for Staging**
   * Deploy to a staging slot and swap with production after validation.
4. **Automated Testing**
   * Add unit tests, integration tests, and performance tests to the pipeline.
5. **Monitor Deployed Functions**
   * Use Application Insights for monitoring.

# Azure DevOps – CICD Pipeline (Azure LogicApps)

Implementing a CI/CD pipeline for **Azure Logic Apps** involves automating the deployment of Logic App resources from development to production environments. This can be achieved using Azure DevOps

### **1. Set Up Your Development Environment**

* Use **Visual Studio Code** with the Azure Logic Apps (Standard) extension to develop your Logic Apps.
* For Logic Apps (Consumption), export the ARM template directly from the Azure portal.

### **2. Organize Your Logic Apps Code**

* Save the Logic App definition (workflow.json) and related files in a version control system Azure Repos.
* Use **parameterization** for environment-specific settings (e.g., connection strings, API keys).

### **3. Choose a CI/CD Tool**

* **Azure DevOps**

### **4. Define the CI/CD Pipeline**

#### **4.1 Continuous Integration (CI)**

1. **Build Pipeline (Optional)**
   * For Logic Apps (Standard): Build the project using a Node.js or .NET runtime if applicable.
   * Validate workflows using schema validation or unit tests.
2. **Store Artifacts**
   * Package the Logic App definition and configuration files as artifacts.

#### **4.2 Continuous Deployment (CD)**

1. **Release Pipeline**
   * Use the ARM templates or Bicep templates for deployment.
   * Deploy to staging/production environments in Azure using tasks like **Azure Resource Manager Deployment** (Azure DevOps) or **Azure CLI Actions** (GitHub).
2. **Connection Management**
   * Use **Service Principal** or **Managed Identity** for secure authentication of Logic App connections.

### **5. Implement CI/CD with Azure DevOps**

1. **Create a Build Pipeline**
   * Trigger on code commits or pull requests.
   * Tasks:
     + Linting/Validation of JSON files.
     + Packing the Logic App definition.
   * Publish as a build artifact.
2. **Create a Release Pipeline**
   * Deploy artifacts using Azure Resource Manager (ARM) templates.
   * Parameters should replace environment-specific values dynamically.

### **6. Best Practices**

1. **Use Parameter Files**
   * Separate environment-specific configurations.
2. **Secure Secrets**
   * Use Azure Key Vault to store sensitive information.
3. **Rollback Plans**
   * Implement versioning and rollback mechanisms.
4. **Testing**
   * Validate Logic Apps in a testing environment before promoting to production.

# Azure DevOps – SecOps

GitHub Advanced Security for Azure DevOps integrates the suite of security features from GitHub Advanced Security into Azure Repos and includes the following features:

* Secret scanning for push protection: verify if code pushes contain commits that reveal secrets like credentials.
* Secret Scanning Repository: Scan your repository to find accidentally committed exposed secrets.
* Dependency scanning involves searching for known vulnerabilities in both direct and transitive open-source dependencies.
* Code Scanning: Use the CodeQL static analysis engine to identify code-level application vulnerabilities, such as SQL injection and authentication bypass.

GitHub Advanced Security for Azure DevOps works with Azure Repos.

Enable Advanced Security at the organization, project, or repository level. To access all scanning tools and results, you must first enable Advanced Security. Once enabled, any future pushes containing secrets to a repository with this policy activated will be blocked, and repository secret scanning will occur in the background.

A diagram of a security system

AI-generated content may be incorrect.

## **Enable GitHub Advanced Security**

**Where:** Azure DevOps > Repos > Advanced Security

**Action:**

* Turn on Advanced Security at the repository level.
* Automatically enables:
  + Secret Scanning (repo & push protection)
  + Code Scanning
  + Dependency Scanning (if configured in pipelines)

## **Enable Secret Scanning**

#### Features

* Scans for sensitive data in historical commits & ongoing pushes.
* Alerts for credentials, tokens, and keys (only if a full pair is present).

#### Setup

* No additional config is needed beyond enabling Advanced Security.
* To **enable/disable push protection**:  
  Azure DevOps → Repos → [Repository Name] → Settings → Advanced Security → Secret Scanning

#### Monitoring

* View alerts: **Repos → Advanced Security → Secrets tab**
* Filter by alert state and secret type.

## **Enable Dependency Scanning**

#### Features

* Detects vulnerabilities in open-source components.
* Alerts raised at the repo level.

#### Setup

* Integrate into the CI pipeline using a **Dependency Scanning task:**

|  |
| --- |
| - task: DependencyScanning@1  inputs:  scanType: 'full' |

* Add this task in all pipelines you want to be scanned.

#### Notes

* Results are **aggregated per repository.**
* Triggers annotations on PRs if the policy is set.

## **Enable Code Scanning with CodeQL**

#### Features

* Static code analysis using CodeQL engine.
* Detects security vulnerabilities and coding errors.

#### Setup

* Integrate CodeQL tasks in the pipeline:

|  |
| --- |
| - task: InitializeCodeQL@0  inputs:  languages: 'csharp, javascript'  enableAutomaticCodeQLInstall: true  - task: CodeQLAnalysis@0  - task: PublishCodeQLResults@0 |

* Supported languages: csharp, cpp, go, java, javascript, python, ruby, swift

#### Notes

* Configure languages in the InitializeCodeQL step.
* Use enableAutomaticCodeQLInstall: true for self-hosted agents.

## **Enable Pull Request Annotations**

#### Features

* Automatically adds annotations to PRs with alerts from:
  + Dependency Scanning
  + Code scanning

#### Requirements

* A **build validation policy** was applied to PRs with scanning tasks included.
* A scan must have been completed on:
  + **Default branch**
  + **Target the branch** before scanning the PR source branch.

#### Tips

* Alerts from the default and target branches influence the PR annotations.
* Refer to alert dashboards for guidance.

## **Enable Monitoring & Alerting**

#### Centralized View

* Go to: **Repos → Advanced Security**
* Tabs: Secrets, Dependencies, Code Scanning
* View:
  + Active alerts
  + State (Open, Resolved, Dismissed)
  + Secret types or vulnerability details
  + Suggested remediation steps