## Strategic Approach for Continuous Integration Using GitHub Actions

### Overview of the Recommended CI Process

The following outlines a structured yet flexible approach to organizing and managing changes and releases in a GitHub repository using GitHub Actions. This strategy ensures clean merges, reliable builds, automated versioning, and streamlined deployments, aligning seamlessly with Continuous Delivery (CD) processes managed by ArgoCD, GitOps, Artifactory, and Azure Storage.

### **Key Technologies**

The following technologies will be used in creation of this continuous integration and delivery approach:

* **ArgoCD**: Continuous delivery tool for Kubernetes using GitOps. ArgoCD monitors our Git repositories and automatically syncs the desired state with the actual state in our Kubernetes clusters. It provides a declarative way to define application configurations and handles reconciliation.
* **GitHub Actions**: CI pipeline to build and tag artifacts. Automates the build process, runs tests, and creates versioned artifacts that are ready for deployment. The pipeline is triggered on code changes and ensures consistent build practices.
* **Azure DevOps (ADO)**: Agile tracking and planning. Provides comprehensive project management capabilities, including work item tracking, sprint planning, and integration with our development workflow.
* **SonarQube**: Code quality and security analysis tool. Provides detailed reports on code quality, security vulnerabilities, and bugs. It can be integrated into the CI pipeline to ensure that only high-quality, secure code is deployed.
* **Artifactory**: Artifact repository for storing container images and zip files. Acts as a secure, centralized location for all our build artifacts, supporting version control and access management.
* **Kustomize**: Kubernetes resource customization. Allows us to maintain environment-specific configurations while keeping base configurations DRY (Don’t Repeat Yourself).
* **Helm**: Kubernetes package manager for deploying API containers. Provides templating and packaging capabilities for complex Kubernetes applications, making deployments more manageable.
* **Azure Blob Storage**: Direct deployment of UX zip files. Offers a scalable solution for storing and serving static web content, with built-in CDN capabilities for improved performance.

### Branch and PR Management

* **Branch Protection**: Enable branch protection rules on the main branch requiring pull requests (PRs) to merge. This prevents direct pushes to main.

### Branch Naming and PR Traceability

* Clearly name branches by incorporating relevant information such as issue numbers (e.g., feature/123-improve-ui).

Example branch creation commands:

**For a feature branch:**

# Create and switch to a new feature branch  
git checkout -b feature/123-new-feature main

**For a bug fix branch:**

# Create and switch to a new bug fix branch  
git checkout -b fix/456-login-issue main

Note: Always create branches from the latest main:

git checkout main  
git pull  
git checkout -b feature/123-new-feature

### Automated Workflows

1. **PR Validation Workflow**
   * Runs on PR creation/update
   * Validates code quality, tests, security
   * Ensures build artifacts can be created
   * Blocks merge if checks fail
2. **Version Management Workflow**
   * Runs on merge to main
   * Determines semantic version bump
   * Creates Git tag and GitHub release
   * Updates version in package files
3. **Artifact Publishing Workflow**
   * Runs after version workflow
   * Builds and publishes artifacts
   * Updates deployment manifests
   * Triggers downstream deployments

Example workflow sequence:

The following summarises the 4 workflows that will be added to every repository.

1. **Validate PR Workflow**
   * Trigger: Pull requests created against main
   * Tasks:
     + **Mandatory**: Validate packaging logic (build artifacts, container images).
     + **Mandatory**: Run tests, static analysis, security checks.
2. **Automated Semantic Release Workflow**
   * Trigger: Push to main
   * Tasks:
     + **Mandatory**: Create releases based on semantic commit messages and PR labels.
     + **Optional**: Automatically update version numbers.
3. **Package and Artifact Workflow**
   * Trigger: Push to main
   * Tasks:
     + **Mandatory**: Build and publish artifacts.
     + **Optional**: Update deployment manifests.
4. **Deploy to Development Workflow**
   * Trigger: Push to main
   * Tasks:
     + **Mandatory**: Deploy artifacts directly to the development environment
     + **Optional**: Run tests against the deployed artifacts

#### Workflow 1: Validate PR

Trigger: Pull requests created against main

Tasks: - **Mandatory**: Validate packaging logic (build artifacts, container images). - **Mandatory**: Run tests, static analysis, security checks.

Example:

# File: .github/workflows/ci-validate.yaml  
name: CI Validation  
  
on:  
 pull\_request:  
 branches:  
 - main  
  
jobs:  
 build:  
 runs-on: ubuntu-latest  
 steps:  
 - uses: actions/checkout@v4  
 - name: Set up Docker  
 uses: docker/setup-buildx-action@v3  
 - name: Build Docker Image  
 run: docker build -t my-app:${{ github.sha }} .  
 - name: Run Tests  
 run: |  
 npm install  
 npm test

#### Workflow 2: Automated Semantic Release

Trigger: Push to main

Tasks: - **Mandatory**: Create releases based on semantic commit messages and PR labels. - **Optional**: Automatically update version numbers.

Use Release Drafter to automate GitHub Releases: https://github.com/release-drafter/release-drafter

First, create the release drafter configuration:

# File: .github/release-drafter.yml  
  
name-template: 'v$RESOLVED\_VERSION'  
tag-template: 'v$RESOLVED\_VERSION'  
categories:  
 - title: '🚀 Features'  
 labels:  
 - 'feature'  
 - 'enhancement'  
 - 'feat'  
 - title: '🐛 Bug Fixes'  
 labels:  
 - 'fix'  
 - 'bugfix'  
 - 'bug'  
 - title: '🧰 Maintenance'  
 labels:  
 - 'chore'  
 - 'docs'  
 - 'refactor'  
   
change-template: '- $TITLE @$AUTHOR (#$NUMBER)'  
  
version-resolver:  
 major:  
 labels:  
 - 'major'  
 - 'breaking'  
 minor:  
 labels:  
 - 'minor'  
 - 'feature'  
 - 'feat'  
 patch:  
 labels:  
 - 'patch'  
 - 'fix'  
 - 'bugfix'  
 - 'chore'  
 - 'docs'  
 default: patch  
  
template: |  
 ## What's Changed  
 $CHANGES  
   
 \*\*Full Changelog\*\*: https://github.com/$OWNER/$REPOSITORY/compare/$PREVIOUS\_TAG...v$RESOLVED\_VERSION

Then create the workflow:

name: Release Drafter  
  
on:  
 push:  
 branches:  
 - main  
 # Allows manual triggering of the workflow  
 workflow\_dispatch:  
  
permissions:  
 contents: read  
 pull-requests: write  
  
jobs:  
 update\_release\_draft:  
 permissions:  
 contents: write  
 pull-requests: write  
 runs-on: ubuntu-latest  
 steps:  
 - uses: release-drafter/release-drafter@v5  
 env:  
 GITHUB\_TOKEN: ${{ secrets.GITHUB\_TOKEN }}

This pattern provides several benefits: 1. **Automated Release Notes**: Automatically generates changelog based on merged PRs 2. **Smart Versioning**: - Major version bump for breaking changes - Minor version bump for new features - Patch version bump for fixes and maintenance 3. **Categorized Changes**: Groups changes by type (features, fixes, maintenance) 4. **PR Integration**: Works with PR labels to determine version bumps and categories

To use this workflow effectively: 1. Label your PRs appropriately (feature, fix, chore, etc.) 2. Use semantic commit messages 3. PRs will automatically be categorized and included in the next release draft 4. Release notes are automatically generated and kept up-to-date 5. When ready to release, publish the draft release in GitHub

Example PR labels and their effects: - breaking or major: Triggers major version bump (1.0.0 → 2.0.0) - feature or enhancement: Triggers minor version bump (1.0.0 → 1.1.0) - fix or bugfix: Triggers patch version bump (1.0.0 → 1.0.1)

### Commit Message Guidelines

Use semantic commit messages to clearly communicate the nature of changes. The commit message structure should be:

<type>(<scope>): <subject>  
  
[optional body]  
  
[optional footer]

**Types and Their Impact:** - feat: Introduces a new feature (triggers minor version bump) - fix: Corrects a bug (triggers patch version bump) - docs: Documentation updates (triggers patch version bump) - chore: Maintenance tasks (triggers patch version bump) - refactor: Code improvements (triggers patch version bump) - perf: Performance improvements (triggers patch version bump) - style: Code style changes (triggers patch version bump) - test: Adding or updating tests - ci: Changes to CI configuration files - build: Changes affecting build system

**Breaking Changes:** To indicate a breaking change, add BREAKING CHANGE: in the commit footer or append ! after the type:

# Using footer  
git commit -m "feat: add user authentication API  
BREAKING CHANGE: completely new auth flow"  
  
# Using ! syntax  
git commit -m "feat!: change authentication API structure"

**Examples of Good Commit Messages:**

# Feature with scope  
git commit -m "feat(auth): add OAuth2 login support"  
  
# Bug fix with issue reference  
git commit -m "fix(api): correct rate limiting logic (#123)"  
  
# Breaking change with detailed description  
git commit -m "feat(api)!: change user endpoints  
   
Previous user endpoints are now deprecated.  
New endpoints follow REST principles more closely.  
  
BREAKING CHANGE: /api/v1/user/\* endpoints now require authentication"  
  
# Documentation update  
git commit -m "docs(readme): update deployment instructions"  
  
# Performance improvement  
git commit -m "perf(database): optimize query performance"

**Working with Release Drafter:**

1. **Creating a Feature Branch and Commits:**

# Create feature branch  
git checkout -b feature/user-auth  
  
# Make changes and commit  
git add .  
git commit -m "feat(auth): implement user authentication  
   
- Add login endpoint  
- Implement JWT token generation  
- Add password hashing"  
  
# Push changes  
git push origin feature/user-auth

1. **Creating a PR with Appropriate Labels:**

# Using GitHub CLI  
gh pr create \  
 --title "feat(auth): implement user authentication" \  
 --body "Adds complete user authentication system" \  
 --label "feature" \  
 --label "enhancement"

1. **Updating PR with Additional Changes:**

# Make additional changes  
git add .  
git commit -m "test(auth): add authentication unit tests"  
git push origin feature/user-auth  
  
# Add more changes in separate commit  
git add .  
git commit -m "docs(auth): add API documentation for auth endpoints"  
git push origin feature/user-auth

1. **Squash Merging with Semantic Message:** When merging the PR, use a squash merge with a semantic commit message that summarizes all changes:

# If using GitHub CLI  
gh pr merge feature/user-auth \  
 --squash \  
 --title "feat(auth): implement user authentication system (#123)" \  
 --body "Added complete authentication system including:  
- Login endpoints  
- JWT implementation  
- Password hashing  
- Unit tests  
- API documentation"

**Version Bump Examples:** - A commit with feat: → Minor version bump (1.0.0 → 1.1.0) - A commit with fix: → Patch version bump (1.0.0 → 1.0.1) - A commit with feat!: or BREAKING CHANGE: → Major version bump (1.0.0 → 2.0.0)

**Version Bumping Mechanics:**

1. **Automatic Version Bumping:** The release-drafter determines the version bump based on two factors:

* # 1. PR Labels - Add these when creating the PR  
  gh pr create \  
   --title "Add new authentication system" \  
   --label "feature" # Triggers minor bump  
   --label "breaking" # Triggers major bump  
   --label "fix" # Triggers patch bump  
    
  # 2. Commit Message Conventions  
  # Major bump (2.0.0)  
  git commit -m "feat!: completely new auth system"  
  # or  
  git commit -m "feat: new auth system  
    
  BREAKING CHANGE: This replaces the old auth system"  
    
  # Minor bump (1.1.0)  
  git commit -m "feat: add new login method"  
    
  # Patch bump (1.0.1)  
  git commit -m "fix: correct login validation"

1. **Manual Version Control:** You can manually control version bumping by:

* # Create a PR with specific version intent  
  gh pr create \  
   --title "feat: add new feature" \  
   --label "minor" \  
   --label "feature" \  
   --body "This PR implements feature X  
    
   Version bump: minor"  
    
  # Force a specific version through release UI  
  gh release create v1.2.0 \  
   --title "v1.2.0" \  
   --notes "Release notes..." \  
   --target main

1. **Version Bump Hierarchy:** When multiple changes are present, the highest-impact change determines the version:

* # These changes in one PR:  
  git commit -m "fix: update error handling"  
  git commit -m "feat: add new endpoint"  
  git commit -m "docs: update README"  
  # Results in minor bump (1.1.0) because feat > fix > docs  
    
  # If any breaking change exists, it takes precedence:  
  git commit -m "fix: update error handling"  
  git commit -m "feat!: breaking API change"  
  # Results in major bump (2.0.0)

1. **Release Workflow:**

* # 1. Create feature branch  
  git checkout -b feature/new-auth  
    
  # 2. Make changes and commit with semantic messages  
  git commit -m "feat(auth): add OAuth support"  
    
  # 3. Create PR with appropriate labels  
  gh pr create \  
   --title "feat(auth): add OAuth support" \  
   --label "feature" \  
   --body "Adds OAuth authentication support"  
    
  # 4. After PR is merged, release-drafter will:  
  # - Update draft release  
  # - Determine version bump  
  # - Generate changelog  
    
  # 5. Publish release (manual or automated)  
  gh release create v1.1.0 \  
   --draft=false \  
   --title "v1.1.0" \  
   --notes-file CHANGELOG.md

**Best Practices for Version Management:** 1. Always use semantic commit messages 2. Label PRs consistently with version intent 3. Include breaking change notices in commit messages when applicable 4. Review draft releases before publishing 5. Keep one significant change per PR for clear version bumping 6. Document version bumps in PR descriptions

### Workflow 3: Publish to Artifactory

#### 3.1 UX Artifact Publishing

# File: .github/workflows/publish-ux-to-artifactory.yaml  
name: Publish UX to Artifactory  
  
on:  
 release:  
 types: [published]  
  
jobs:  
 publish-ux:  
 runs-on: ubuntu-latest  
 steps:  
 - uses: actions/checkout@v4  
   
 # Setup Node.js for UX build  
 - name: Setup Node.js  
 uses: actions/setup-node@v4  
 with:  
 node-version: '18'  
 cache: 'npm'  
 cache-dependency-path: 'package-lock.json'  
   
 # Build UX application  
 - name: Build UX  
 run: |  
 npm ci  
 npm run build  
   
 # Package UX into versioned zip  
 - name: Package UX  
 run: |  
 cd dist  
 zip -r ../ux-${{ github.event.release.tag\_name }}.zip .  
 cd ..  
   
 # Setup JFrog CLI  
 - name: Setup JFrog CLI  
 uses: jfrog/setup-jfrog-cli@v4  
 env:  
 JF\_URL: ${{ secrets.JF\_URL }}  
 JF\_USER: ${{ secrets.JF\_USER }}  
 JF\_PASSWORD: ${{ secrets.JF\_PASSWORD }}  
   
 # Publish UX zip to Artifactory  
 - name: Publish UX to Artifactory  
 run: |  
 # Create artifact metadata  
 cat << EOF > artifact-props.json  
 {  
 "version": "${{ github.event.release.tag\_name }}",  
 "build.number": "${{ github.run\_number }}",  
 "vcs.revision": "${{ github.sha }}",  
 "build.timestamp": "$(date -u +"%Y-%m-%dT%H:%M:%SZ")",  
 "build.url": "${{ github.server\_url }}/${{ github.repository }}/actions/runs/${{ github.run\_id }}"  
 }  
 EOF  
   
 # Upload UX zip with properties  
 jf rt upload \  
 --props-file=artifact-props.json \  
 "ux-${{ github.event.release.tag\_name }}.zip" \  
 "frontend-local/ux/release-${{ github.event.release.tag\_name }}/"  
   
 # Create deployment manifest  
 - name: Generate Deployment Manifest  
 run: |  
 cat << EOF > ux-version.yaml  
 version: ${{ github.event.release.tag\_name }}  
 artifactPath: frontend-local/ux/release-${{ github.event.release.tag\_name }}/ux-${{ github.event.release.tag\_name }}.zip  
 buildNumber: ${{ github.run\_number }}  
 gitCommit: ${{ github.sha }}  
 buildTimestamp: $(date -u +"%Y-%m-%dT%H:%M:%SZ")  
 EOF  
   
 # Upload version manifest  
 jf rt upload \  
 ux-version.yaml \  
 "frontend-local/ux/release-${{ github.event.release.tag\_name }}/manifest.yaml"

#### 3.2 API Artifact Publishing

# File: .github/workflows/publish-api-to-artifactory.yaml  
name: Publish API to Artifactory  
  
on:  
 release:  
 types: [published]  
  
jobs:  
 publish-api:  
 runs-on: ubuntu-latest  
 steps:  
 - uses: actions/checkout@v4  
   
 # Login to Docker registry  
 - name: Login to Artifactory Docker Registry  
 uses: docker/login-action@v3  
 with:  
 registry: ${{ secrets.DOCKER\_REGISTRY }}  
 username: ${{ secrets.JF\_USER }}  
 password: ${{ secrets.JF\_PASSWORD }}  
   
 # Set up Docker Buildx  
 - name: Set up Docker Buildx  
 uses: docker/setup-buildx-action@v3  
   
 # Build and push API Docker image  
 - name: Build and Push API Image  
 uses: docker/build-push-action@v5  
 with:  
 context: .  
 push: true  
 tags: |  
 ${{ secrets.DOCKER\_REGISTRY }}/api:${{ github.event.release.tag\_name }}  
 ${{ secrets.DOCKER\_REGISTRY }}/api:latest  
 cache-from: type=registry,ref=${{ secrets.DOCKER\_REGISTRY }}/api:buildcache  
 cache-to: type=registry,ref=${{ secrets.DOCKER\_REGISTRY }}/api:buildcache,mode=max  
   
 # Setup JFrog CLI  
 - name: Setup JFrog CLI  
 uses: jfrog/setup-jfrog-cli@v4  
 env:  
 JF\_URL: ${{ secrets.JF\_URL }}  
 JF\_USER: ${{ secrets.JF\_USER }}  
 JF\_PASSWORD: ${{ secrets.JF\_PASSWORD }}  
   
 # Create API version manifest  
 - name: Generate API Version Manifest  
 run: |  
 cat << EOF > api-version.yaml  
 version: ${{ github.event.release.tag\_name }}  
 image: ${{ secrets.DOCKER\_REGISTRY }}/api:${{ github.event.release.tag\_name }}  
 buildNumber: ${{ github.run\_number }}  
 gitCommit: ${{ github.sha }}  
 buildTimestamp: $(date -u +"%Y-%m-%dT%H:%M:%SZ")  
 EOF  
   
 # Upload version manifest  
 jf rt upload \  
 api-version.yaml \  
 "docker-local/api/release-${{ github.event.release.tag\_name }}/manifest.yaml"

#### Directory Structure in Artifactory

artifactory/  
├── docker-local/ # API artifacts  
│ └── api/  
│ ├── release-v1.2.3/  
│ │ ├── manifest.yaml # API version metadata  
│ │ └── image-manifest.json # Docker image manifest  
│ └── latest/  
└── frontend-local/ # UX artifacts  
 └── ux/  
 └── release-v1.2.3/  
 ├── ux-v1.2.3.zip # UX bundle  
 └── manifest.yaml # UX version metadata

#### Retrieving Artifacts

For UX deployments:

# Download UX manifest to check version info  
jf rt download "frontend-local/ux/release-v1.2.3/manifest.yaml" ./  
  
# Download UX bundle  
jf rt download "frontend-local/ux/release-v1.2.3/ux-v1.2.3.zip" ./

For API deployments:

# Download API manifest to check version info  
jf rt download "docker-local/api/release-v1.2.3/manifest.yaml" ./  
  
# Pull API Docker image  
docker pull $DOCKER\_REGISTRY/api:v1.2.3

#### Benefits of Separation

1. **Independent Versioning**
   * UX and API can be versioned independently
   * Allows for different release cycles
   * Supports independent rollbacks
2. **Simplified CI/CD**
   * Smaller, focused workflows
   * Faster builds and deployments
   * Reduced pipeline complexity
3. **Clear Separation of Concerns**
   * Each repository has its own workflow
   * Separate teams can manage their own releases
   * Independent scaling of build resources
4. **Enhanced Traceability**
   * Separate manifests for each component
   * Clear artifact lineage
   * Independent audit trails
5. **Flexible Deployment**
   * Deploy UX changes without API updates
   * Roll back API without affecting UX
   * Mix and match versions as needed

#### Workflow 3: Deploy to Development

Trigger: Push to main

Tasks: - **Mandatory**: Deploy artifacts directly to the development environment - **Optional**: Run tests against the deployed artifacts

Use GitHub Actions to automate the deployment:

name: Deploy to Development  
  
on:  
 push:  
 branches:  
 - main  
  
jobs:  
 deploy:  
 runs-on: ubuntu-latest  
 steps:  
 - name: Checkout code  
 uses: actions/checkout@v2  
  
 - name: Deploy to Development  
 run: |  
 # Add deployment commands here  
 # Example: kubectl apply -f deployment.yaml  
 # Example: az storage blob upload --account-name your-storage-account --container-name your-container-name --name destination-filename.zip --file /path/to/your/source.zip

This workflow provides fast continuous integration for engineering. GitOps will only be used for moving code from development to QA environment.

To deploy a UX zip file to a storage account, use the following command:

az storage blob upload --account-name your-storage-account --container-name your-container-name --name destination-filename.zip --file /path/to/your/source.zip

Replace your-storage-account, your-container-name, destination-filename.zip, and /path/to/your/source.zip with your actual storage account, container, file name, and local file path.

> \*\*Note\*\*: This can be discussed as a team if we add in GitOps here, my suggestion for tactical delivery is we deploy straing to dev for speed of testing once the code is merged to main, code is only move through environments through git ops

### Integration with Continuous Delivery (CD)

These CI workflows integrate directly into the Continuous Delivery process using ArgoCD and GitOps principles. After artifacts are published to Artifactory, ArgoCD automatically monitors repository changes defined in your GitOps repository structure, triggering deployments to Kubernetes-based environments.

#### GitOps Repository Structure

gitops-repo/  
├── base/ # Base configurations  
│ ├── api/ # API component base  
│ │ ├── deployment.yaml  
│ │ ├── service.yaml  
│ │ └── kustomization.yaml  
│ └── ui/ # UI component base  
│ ├── deployment.yaml  
│ ├── service.yaml  
│ └── kustomization.yaml  
│  
├── environments/  
│ ├── dev/ # Development environment  
│ │ ├── api/  
│ │ │ ├── kustomization.yaml  
│ │ │ └── env-values.yaml  
│ │ └── ui/  
│ │ ├── kustomization.yaml  
│ │ └── env-values.yaml  
│ │  
│ ├── qa/ # QA environment  
│ │ ├── api/  
│ │ │ ├── kustomization.yaml  
│ │ │ └── env-values.yaml  
│ │ └── ui/  
│ │ ├── kustomization.yaml  
│ │ └── env-values.yaml  
│ │  
│ ├── uat/ # UAT environment  
│ │ ├── api/  
│ │ │ ├── kustomization.yaml  
│ │ │ ├── env-values.yaml  
│ │ │ └── scaling.yaml  
│ │ └── ui/  
│ │ ├── kustomization.yaml  
│ │ ├── env-values.yaml  
│ │ └── cdn-config.yaml  
│ │  
│ └── prod/ # Production environment  
│ ├── api/  
│ │ ├── kustomization.yaml  
│ │ ├── env-values.yaml  
│ │ ├── scaling.yaml  
│ │ └── hpa.yaml  
│ └── ui/  
│ ├── kustomization.yaml  
│ ├── env-values.yaml  
│ ├── cdn-config.yaml  
│ └── ssl-config.yaml  
│  
└── argocd/ # ArgoCD ApplicationSets  
 ├── api-appset.yaml # API deployment configuration  
 ├── ui-appset.yaml # UI deployment configuration  
 └── promotion-workflow.yaml # Promotion workflow definition

#### Environment-Specific Configurations

1. **Development (Dev)**
   * Automatic deployments from main branch
   * Minimal resource requests/limits
   * Debug logging enabled
   * No SSL requirement

* # environments/dev/api/env-values.yaml  
  resources:  
   requests:  
   memory: "256Mi"  
   cpu: "100m"  
   limits:  
   memory: "512Mi"  
   cpu: "200m"  
  logging:  
   level: debug

1. **Quality Assurance (QA)**
   * Deployment on release candidates
   * Moderate resource allocation
   * Test data integration

* # environments/qa/api/env-values.yaml  
  resources:  
   requests:  
   memory: "512Mi"  
   cpu: "200m"  
   limits:  
   memory: "1Gi"  
   cpu: "500m"  
  testing:  
   dataSet: "qa-dataset"

1. **User Acceptance Testing (UAT)**
   * Production-like environment
   * Manual promotion required
   * Enhanced monitoring

* # environments/uat/api/env-values.yaml  
  resources:  
   requests:  
   memory: "1Gi"  
   cpu: "500m"  
   limits:  
   memory: "2Gi"  
   cpu: "1000m"  
  monitoring:  
   enabled: true  
   detailedMetrics: true

1. **Production (Prod)**
   * Manual promotion required
   * High availability configuration
   * Auto-scaling enabled
   * Enhanced security measures

* # environments/prod/api/env-values.yaml  
  resources:  
   requests:  
   memory: "2Gi"  
   cpu: "1000m"  
   limits:  
   memory: "4Gi"  
   cpu: "2000m"  
  security:  
   networkPolicies: true  
   podSecurityPolicies: true  
  highAvailability:  
   enabled: true  
   minReplicas: 3

#### Promotion Workflow

# argocd/promotion-workflow.yaml  
apiVersion: argoproj.io/v1alpha1  
kind: Workflow  
metadata:  
 name: promotion-workflow  
spec:  
 templates:  
 - name: promote-to-uat  
 steps:  
 - name: validate-qa  
 template: validate-environment  
 arguments:  
 parameters:  
 - name: env  
 value: qa  
 - name: deploy-uat  
 template: deploy-environment  
 arguments:  
 parameters:  
 - name: env  
 value: uat  
  
 - name: promote-to-prod  
 steps:  
 - name: validate-uat  
 template: validate-environment  
 arguments:  
 parameters:  
 - name: env  
 value: uat  
 - name: deploy-prod  
 template: deploy-environment  
 arguments:  
 parameters:  
 - name: env  
 value: prod

#### Deployment Strategy

1. **Development**
   * Automatic deployment from main
   * Rolling updates
   * No approval required
2. **QA**
   * Deploys from release candidates
   * Automated testing gates
   * QA team approval required
3. **UAT**
   * Manual promotion from QA
   * Full regression testing
   * Business stakeholder approval required
4. **Production**
   * Manual promotion from UAT
   * Change advisory board approval
   * Scheduled deployment windows
   * Blue-green deployment strategy

#### Best Practices for Multi-Environment GitOps

1. **Configuration Management**
   * Use Kustomize for environment-specific changes
   * Maintain secrets in HashiCorp Vault or similar
   * Version all configuration changes
2. **Promotion Process**
   * Implement clear promotion criteria
   * Automate validation checks
   * Maintain audit trails for promotions
3. **Security Measures**
   * Increase security controls progressively
   * Implement different service accounts per environment
   * Use network policies to isolate environments
4. **Monitoring and Observability**
   * Configure graduated monitoring levels
   * Implement environment-specific alerts
   * Maintain separate logging streams

Following this strategic integration of GitHub Actions with ArgoCD, GitOps, Artifactory, and Azure ensures a robust, scalable, and efficient CI/CD pipeline.