MonoRepo CICD Pipeline Implementation Guide

Executive Summary

This document outlines comprehensive approaches for implementing MonoRepo-based CICD pipelines while maintaining compatibility with existing infrastructure (CloudBees CI, BitBucket, SonarQube, Fortify, NexusIQ, uDeploy, PCF).

Current State Analysis

Existing Architecture

- **Individual repositories** per microservice
- **Pipeline per microservice** with standardized stages
- **Technology stack**: Maven, Gradle, Node.js (auto-detected)
- **Security scanning**: SonarQube, Fortify, NexusIQ (parallel execution)
- **Deployment**: uDeploy to PCF
- **Shared libraries** for common pipeline logic
- **Individual Fortify App IDs** per microservice

Key Challenges for MonoRepo Transition

1. **Change detection** - Only build/deploy affected services
2. **Fortify App ID management** - Multiple IDs in single repo
3. **Dependency management** - Cross-service dependencies
4. **Build orchestration** - Parallel vs sequential builds
5. **Deployment coordination** - Service interdependencies

Approach 1: Path-Based Trigger Strategy

Overview

Implement change detection to trigger builds only for modified services and their dependents.

Architecture

```
monorepo/
├── services/
│   ├── service-a/
│   │   ├── pom.xml
```

```
│   │   └── src/
│   ├── service-b/
│   │   ├── build.gradle
│   │   └── src/
│   └── service-c/
│       ├── package.json
│       └── src/
├── shared/
│   └── common-libs/
├── Jenkinsfile
└── pipeline-config.yml
```

Implementation Strategy

1. Change Detection Logic

groovy

```groovy
// Shared Library Function
def getChangedServices() {
    def changedFiles = sh(
        script: "git diff --name-only HEAD~1 HEAD",
        returnStdout: true
    ).trim().split('\n')

    def changedServices = []
    changedFiles.each { file ->
        if (file.startsWith('services/')) {
            def serviceName = file.split('/')[1]
            if (!changedServices.contains(serviceName)) {
                changedServices.add(serviceName)
            }
        }
        // If shared components changed, build all services
        if (file.startsWith('shared/')) {
            return getAllServices()
        }
    }
    return changedServices
}
```

2. Service Configuration Management

yaml

```yaml
# pipeline-config.yml
services:
  service-a:
    buildType: maven
    fortifyAppId: "APP-001"
    sonarProjectKey: "service-a"
    deploymentOrder: 1
    dependencies: []

  service-b:
    buildType: gradle
    fortifyAppId: "APP-002"
    sonarProjectKey: "service-b"
    deploymentOrder: 2
    dependencies: ["service-a"]

  service-c:
    buildType: nodejs
    fortifyAppId: "APP-003"
    sonarProjectKey: "service-c"
    deploymentOrder: 3
    dependencies: ["service-a", "service-b"]
```

3. Enhanced Jenkinsfile

groovy

```
@Library('shared-pipeline-library') _

pipeline {
    agent any

    parameters {
        choice(name: 'ENVIRONMENT', choices: ['dev', 'staging', 'prod'])
        booleanParam(name: 'FORCE_BUILD_ALL', defaultValue: false)
        string(name: 'SERVICES_TO_BUILD', defaultValue: '', description: 'Comma-separated service names')
    }

    stages {
        stage('Determine Services to Build') {
            steps {
                script {
                    if (params.FORCE_BUILD_ALL) {
                        env.SERVICES_TO_BUILD = getAllServices().join(',')
                    } else if (params.SERVICES_TO_BUILD) {
                        env.SERVICES_TO_BUILD = params.SERVICES_TO_BUILD
                    } else {
                        def changedServices = getChangedServices()
                        def servicesToBuild = getDependentServices(changedServices)
                        env.SERVICES_TO_BUILD = servicesToBuild.join(',')
                    }
                    echo "Services to build: ${env.SERVICES_TO_BUILD}"
                }
            }
        }

        stage('Parallel Service Builds') {
            steps {
                script {
                    def servicesToBuild = env.SERVICES_TO_BUILD.split(',')
                    def parallelBuilds = [:]

                    servicesToBuild.each { service ->
                        parallelBuilds[service] = {
                            buildService(service)
                        }
                    }

                    parallel parallelBuilds
                }
```

```
            }
        }

        stage('Coordinated Deployment') {
            steps {
                script {
                    deployServicesInOrder(env.SERVICES_TO_BUILD.split(','))
                }
            }
        }
    }
}
```

Pros

- **Efficient builds** - Only changed services
- **Maintains isolation** - Service-specific configurations
- **Gradual migration** - Can migrate services incrementally
- **Familiar workflow** - Similar to current pipeline structure

Cons

- **Complex dependency management**
- **Potential for missed dependencies**
- **Requires sophisticated change detection**

Approach 2: Matrix-Based Pipeline Strategy

Overview

Use Jenkins matrix builds to execute parallel pipelines for each service with dynamic configuration.

Implementation Strategy

1. Dynamic Matrix Generation

groovy

```
pipeline {
    agent none

    stages {
        stage('Generate Matrix') {
            agent any
            steps {
                script {
                    def services = getServicesToBuild()
                    def matrixConfig = [:]

                    services.each { service ->
                        def config = getServiceConfig(service)
                        matrixConfig[service] = [
                            buildType: config.buildType,
                            fortifyAppId: config.fortifyAppId,
                            sonarProjectKey: config.sonarProjectKey,
                            servicePath: "services/${service}"
                        ]
                    }

                    env.MATRIX_CONFIG = writeJSON returnText: true, json: matrixConfig
                }
            }
        }

        stage('Matrix Build') {
            matrix {
                axes {
                    axis {
                        name 'SERVICE_NAME'
                        values getServicesToBuild()
                    }
                }
                stages {
                    stage('Build Service') {
                        steps {
                            script {
                                def config = readJSON text: env.MATRIX_CONFIG
                                def serviceConfig = config[env.SERVICE_NAME]

                                buildServiceWithConfig(env.SERVICE_NAME, serviceConfig)
                            }
```

```
                }
            }
        }
    }
  }
}
```

2. Service-Specific Build Function

groovy

```groovy
def buildServiceWithConfig(serviceName, config) {
    dir(config.servicePath) {
        // Auto-detect build type or use config
        def buildType = detectBuildType() ?: config.buildType

        stage("${serviceName} - Checkout & Build") {
            // Checkout already done at pipeline level
            executeBuild(buildType)
        }

        stage("${serviceName} - Security Scans") {
            parallel(
                "SonarQube": {
                    executeSonarScan(config.sonarProjectKey, serviceName)
                },
                "Fortify": {
                    executeFortifyScan(config.fortifyAppId, serviceName)
                },
                "NexusIQ": {
                    executeNexusIQScan(serviceName)
                }
            )
        }

        stage("${serviceName} - Deploy") {
            executeUDeploy(serviceName, config)
        }
    }
}
```

Pros

- **True parallel execution**
- **Clean separation of concerns**
- **Easy to scale**
- **Good for independent services**

Cons

- **Resource intensive**
- **Complex dependency handling**
- **Potential resource conflicts**

Approach 3: Hybrid Multi-Stage Pipeline

Overview

Combine change detection with staged pipeline execution, maintaining service isolation while optimizing for efficiency.

Implementation Strategy

1. Three-Phase Pipeline

```groovy
```

```
pipeline {
    agent any

    stages {
        stage('Analysis Phase') {
            parallel {
                stage('Change Detection') {
                    steps {
                        script {
                            env.CHANGED_SERVICES = getChangedServices().join(',')
                            env.AFFECTED_SERVICES = getDependentServices(getChangedServices()).join(',')
                        }
                    }
                }
                stage('Dependency Graph') {
                    steps {
                        script {
                            generateDependencyGraph()
                            env.BUILD_ORDER = calculateBuildOrder().join(',')
                        }
                    }
                }
            }
        }

        stage('Build Phase') {
            steps {
                script {
                    def buildGroups = groupServicesByDependencyLevel()

                    buildGroups.each { level, services ->
                        echo "Building level ${level}: ${services}"

                        def parallelBuilds = [:]
                        services.each { service ->
                            parallelBuilds[service] = {
                                buildServiceStages(service)
                            }
                        }

                        parallel parallelBuilds
                    }
                }
```

```
            }
        }

        stage('Deployment Phase') {
            steps {
                script {
                    deployServicesInOrder(env.AFFECTED_SERVICES.split(','))
                }
            }
        }
    }
}
```

2. Service Build Stages Function

groovy

```
def buildServiceStages(serviceName) {
    def config = getServiceConfig(serviceName)
    def servicePath = "services/${serviceName}"

    dir(servicePath) {
        // Build
        stage("${serviceName} - Build") {
            def buildType = detectBuildType()
            executeBuild(buildType)
            archiveArtifacts artifacts: getArtifactPattern(buildType)
        }

        // Parallel Security Scans
        stage("${serviceName} - Security Scans") {
            parallel(
                "SonarQube": {
                    executeSonarScan(config.sonarProjectKey, serviceName)
                },
                "Fortify": {
                    executeFortifyScan(config.fortifyAppId, serviceName)
                },
                "NexusIQ": {
                    executeNexusIQScan(serviceName)
                }
            )
        }
    }
}
```

Pros

- **Optimal controller utilization** - Services distributed by specialization
- **Dynamic worker efficiency** - Automatic scaling and resource allocation
- **Dependency-aware builds** - Respects service interdependencies
- **Clear pipeline phases** - Structured approach with proper separation
- **Cost optimization** - Workers created/destroyed as needed

Cons

- **Complex orchestration** - Multi-controller coordination overhead
- **Debugging complexity** - Cross-controller troubleshooting challenges

- **Artifact management** - Distributed artifact collection complexity
- **Network overhead** - Inter-controller communication costsComplex orchestration logic**
- **Debugging complexity**

Approach 4: Micro-Pipeline Orchestration

Overview

Create lightweight service-specific pipelines orchestrated by a master pipeline.

Implementation Strategy

1. Master Orchestrator Pipeline

groovy

```
@Library('monorepo-pipeline-library') _

pipeline {
    agent any

    stages {
        stage('Service Discovery') {
            steps {
                script {
                    env.SERVICES_TO_BUILD = discoverServicesToBuild().join(',')
                    echo "Discovered services: ${env.SERVICES_TO_BUILD}"
                }
            }
        }

        stage('Trigger Service Pipelines') {
            steps {
                script {
                    def servicesToBuild = env.SERVICES_TO_BUILD.split(',')
                    def serviceJobs = [:]

                    servicesToBuild.each { service ->
                        serviceJobs[service] = {
                            build job: "monorepo-service-pipeline",
                                parameters: [
                                    string(name: 'SERVICE_NAME', value: service),
                                    string(name: 'GIT_COMMIT', value: env.GIT_COMMIT),
                                    string(name: 'ENVIRONMENT', value: params.ENVIRONMENT)
                                ],
                                wait: true,
                                propagate: true
                        }
                    }

                    parallel serviceJobs
                }
            }
        }

        stage('Deployment Orchestration') {
            when {
                allOf {
                    expression { currentBuild.result == null }
```

```
                expression { env.SERVICES_TO_BUILD != '' }
            }
        }
        steps {
            script {
                orchestrateDeployments(env.SERVICES_TO_BUILD.split(','))
            }
        }
    }
  }
}
```

2. Service Pipeline Template

groovy

```
// Job: monorepo-service-pipeline
pipeline {
    agent any

    parameters {
        string(name: 'SERVICE_NAME', description: 'Service to build')
        string(name: 'GIT_COMMIT', description: 'Git commit hash')
        string(name: 'ENVIRONMENT', defaultValue: 'dev')
    }

    stages {
        stage('Checkout') {
            steps {
                checkout([
                    $class: 'GitSCM',
                    branches: [[name: params.GIT_COMMIT]],
                    userRemoteConfigs: [[url: env.GIT_URL]]
                ])
            }
        }

        stage('Build Service') {
            steps {
                script {
                    def serviceName = params.SERVICE_NAME
                    def servicePath = "services/${serviceName}"

                    dir(servicePath) {
                        def config = getServiceConfig(serviceName)
                        def buildType = detectBuildType()

                        // Build
                        executeBuild(buildType)

                        // Parallel Security Scans
                        parallel(
                            "SonarQube": {
                                executeSonarScan(config.sonarProjectKey, serviceName)
                            },
                            "Fortify": {
                                executeFortifyScan(config.fortifyAppId, serviceName)
                            },
                            "NexusIQ": {
```

```
                    executeNexusIQScan(serviceName)
                }
            )

            // Archive artifacts for deployment
            archiveArtifacts artifacts: getArtifactPattern(buildType)
        }
    }
  }
 }
}
```

Pros

- **Clean separation**
- **Easier debugging**
- **Familiar pipeline structure**
- **Easy to extend**

Cons

- **Additional job management**
- **Coordination complexity**
- **Potential performance overhead**

Shared Library Enhancements

Enhanced Service Detection

```groovy
```

```groovy
// vars/detectBuildType.groovy
def call(String servicePath = '.') {
    dir(servicePath) {
        if (fileExists('pom.xml')) {
            return 'maven'
        } else if (fileExists('build.gradle') || fileExists('build.gradle.kts')) {
            return 'gradle'
        } else if (fileExists('package.json')) {
            return 'nodejs'
        } else {
            error "Unable to detect build type for service in ${servicePath}"
        }
    }
}
```

CloudBees-Specific Implementation Considerations

1. Controller Resource Planning

yaml

```yaml
# CloudBees Controller Sizing Guidelines
controller-java-maven:
  recommendedSpecs:
    cpu: "4 cores"
    memory: "8GB RAM"
    storage: "100GB SSD"
  maxConcurrentBuilds: 8
  specializedFor: ["maven", "fortify-java", "sonar-java"]

controller-nodejs:
  recommendedSpecs:
    cpu: "2 cores"
    memory: "4GB RAM"
    storage: "50GB SSD"
  maxConcurrentBuilds: 12
  specializedFor: ["nodejs", "npm-audit", "javascript-security"]

controller-security:
  recommendedSpecs:
    cpu: "6 cores"
    memory: "16GB RAM"
    storage: "200GB SSD"
  maxConcurrentBuilds: 4
  specializedFor: ["fortify-all", "sonar-enterprise", "nexusiq"]
```

2. Dynamic Worker Templates

yaml

```yaml
# CloudBees Worker Node Templates
maven-builder-template:
  baseImage: "cloudbees/java-build-tools:maven-3.8-jdk11"
  resources:
    requests:
      cpu: "1000m"
      memory: "2Gi"
    limits:
      cpu: "2000m"
      memory: "4Gi"
  nodeSelector:
    workload-type: "maven-build"
  tolerations:
    - key: "build-workload"
      operator: "Equal"
      value: "true"
      effect: "NoSchedule"

gradle-builder-template:
  baseImage: "cloudbees/java-build-tools:gradle-7-jdk11"
  resources:
    requests:
      cpu: "1000m"
      memory: "2Gi"
    limits:
      cpu: "2000m"
      memory: "4Gi"
  nodeSelector:
    workload-type: "gradle-build"

nodejs-builder-template:
  baseImage: "cloudbees/nodejs-build-tools:16-alpine"
  resources:
    requests:
      cpu: "500m"
      memory: "1Gi"
    limits:
      cpu: "1000m"
      memory: "2Gi"
  nodeSelector:
    workload-type: "nodejs-build"

fortify-scanner-template:
```

```yaml
  baseImage: "cloudbees/security-tools:fortify-22.1"
  resources:
    requests:
      cpu: "2000m"
      memory: "4Gi"
    limits:
      cpu: "4000m"
      memory: "8Gi"
  nodeSelector:
    workload-type: "security-scanning"
  tolerations:
    - key: "security-workload"
      operator: "Equal"
      value: "true"
      effect: "NoSchedule"
```

3. Network and Connectivity Optimization

```groovy
// vars/optimizeNetworkConnectivity.groovy
def call() {
    // Configure CloudBees network optimizations
    def networkConfig = [
        artifactTransfer: [
            compression: true,
            chunkSize: "10MB",
            parallelStreams: 3
        ],
        interControllerComm: [
            keepAliveInterval: 60,
            timeoutSeconds: 300,
            retryAttempts: 3
        ]
    ]

    env.NETWORK_CONFIG = writeJSON returnText: true, json: networkConfig
}
```

Migration Strategy for CloudBees Environment

Phase 1: CloudBees Infrastructure Preparation (3-4 weeks)

Week 1-2: Controller Setup and Configuration

1. **Controller Provisioning**
   - Deploy specialized controllers (Java, Node.js, Security)
   - Configure controller-specific plugins and tools
   - Set up inter-controller networking and authentication
2. **Dynamic Worker Templates**
   - Create optimized worker node templates
   - Configure resource limits and node selectors
   - Test worker provisioning and scaling

Week 3-4: Shared Library Development

1. **CloudBees-Optimized Shared Libraries**
   - Develop controller affinity functions
   - Implement dynamic worker management
   - Create distributed artifact handling
2. **Monitoring and Metrics Setup**
   - Configure CloudBees monitoring dashboards
   - Set up performance metrics collection
   - Implement alerting for resource issues

Phase 2: Pilot Implementation (4-5 weeks)

Week 1-2: Monorepo Structure and Pilot Services

1. **Repository Setup**
   - Create monorepo structure in BitBucket
   - Migrate 2-3 pilot services with different build types
   - Configure service-specific pipeline configurations
2. **Pipeline Development**
   - Implement chosen approach (recommended: Hybrid Multi-Stage)
   - Create controller-aware pipeline logic

- Test change detection and dependency resolution

Week 3-4: Security Integration Testing

1. **Security Scanning Validation**
   - Test Fortify scanning on dedicated controllers
   - Validate SonarQube integration with multiple controllers
   - Test NexusIQ scanning with dynamic workers
2. **Deployment Integration**
   - Test uDeploy integration with distributed builds
   - Validate PCF deployment coordination
   - Test rollback scenarios

Week 5: Performance Validation

1. **Load Testing**
   - Simulate concurrent builds across controllers
   - Test dynamic worker scaling
   - Validate network performance between controllers

Phase 3: Gradual Migration (8-10 weeks)

Week 1-3: Service Group Migration

1. **Low-Risk Services** (Independent services, minimal dependencies)
   - Migrate services with no dependencies first
   - Monitor controller utilization and performance
   - Collect metrics and optimize configurations

Week 4-6: Medium Complexity Services

1. **Services with Dependencies**
   - Migrate services with well-defined dependencies
   - Test dependency-aware build ordering
   - Validate cross-service deployment coordination

Week 7-8: Complex Service Migration

1. **High-Dependency Services**

- Migrate remaining complex services
- Handle circular dependencies and complex build graphs
- Optimize for performance and reliability

Week 9-10: Cleanup and Optimization

1. **Infrastructure Cleanup**
   - Decommission old individual pipelines
   - Clean up unused repositories
   - Optimize controller resource allocation

Performance Optimization Recommendations

CloudBees Resource Optimization

```groovy
// vars/optimizeControllerResources.groovy
def call() {
    def currentTime = new Date().format('HH')
    def isBusinessHours = (currentTime.toInteger() >= 8 && currentTime.toInteger() <= 18)

    if (isBusinessHours) {
        // Scale up during business hours
        env.MAX_CONCURRENT_BUILDS_PER_CONTROLLER = "8"
        env.WORKER_SCALE_FACTOR = "1.5"
    } else {
        // Scale down during off-hours
        env.MAX_CONCURRENT_BUILDS_PER_CONTROLLER = "4"
        env.WORKER_SCALE_FACTOR = "0.8"
    }

    echo "Resource optimization applied for time: ${currentTime} (Business hours: ${isBusinessHours})"
}
```

Build Cache Optimization

```groovy
```

```groovy
// vars/optimizeBuildCache.groovy
def call(String serviceName, String buildType) {
    def cacheConfig = [
        maven: [
            cacheDir: "/opt/maven-cache/${serviceName}",
            cacheKey: "maven-${serviceName}-${env.GIT_COMMIT}",
            fallbackKeys: ["maven-${serviceName}-", "maven-global-"]
        ],
        gradle: [
            cacheDir: "/opt/gradle-cache/${serviceName}",
            cacheKey: "gradle-${serviceName}-${env.GIT_COMMIT}",
            fallbackKeys: ["gradle-${serviceName}-", "gradle-global-"]
        ],
        nodejs: [
            cacheDir: "/opt/npm-cache/${serviceName}",
            cacheKey: "npm-${serviceName}-${env.GIT_COMMIT}",
            fallbackKeys: ["npm-${serviceName}-", "npm-global-"]
        ]
    ]

    return cacheConfig[buildType]
}
```

Recommendations

Primary Recommendation: CloudBees Hybrid Multi-Stage Pipeline (Enhanced Approach 3)

Rationale for CloudBees Environment:

- **Optimal controller utilization** - Services distributed based on specialization and load
- **Dynamic worker efficiency** - Automatic scaling with cost optimization
- **Dependency-aware execution** - Respects service interdependencies across controllers
- **Resource isolation** - Security scans on dedicated controllers prevent resource conflicts
- **Monitoring and observability** - Built-in CloudBees metrics and dashboards
- **Enterprise features** - Leverages CloudBees Enterprise capabilities like durability hints

Secondary Recommendation: CloudBees Multi-Controller Matrix Strategy (Enhanced Approach 2)

Rationale:

- **True parallel execution** - Maximum throughput with controller specialization
- **Simplified debugging** - Clear controller-service mapping
- **Easy scaling** - Add controllers as needed for capacity
- **Technology specialization** - Controllers optimized for specific build types

Implementation Priority Order

1. **Phase 1 Priority**: Infrastructure setup and controller optimization
2. **Phase 2 Priority**: Pilot with Hybrid Multi-Stage approach
3. **Phase 3 Priority**: Gradual service migration with continuous monitoring
4. **Phase 4 Priority**: Performance optimization and cost management

Critical Success Factors for CloudBees

1. **Controller Sizing** - Proper resource allocation based on service types
2. **Network Optimization** - Fast inter-controller communication
3. **Worker Management** - Efficient dynamic worker provisioning and cleanup
4. **Monitoring** - Comprehensive visibility into controller and worker utilization
5. **Cost Management** - Dynamic scaling to optimize CloudBees licensing costs

Troubleshooting Guide for CloudBees MonoRepo Implementation

Common Issues and Solutions

1. Controller Overload Issues

Symptoms:

- Builds queuing for extended periods
- Dynamic workers failing to provision
- Inter-controller communication timeouts

Solutions:

```groovy
```

```
// Emergency load balancing function
def handleControllerOverload() {
    def overloadedControllers = getOverloadedControllers()

    overloadedControllers.each { controller ->
        echo "Controller ${controller} is overloaded, redistributing services..."

        // Temporarily redistribute services to other controllers
        redistributeServices(controller)

        // Scale down non-critical builds
        postponeNonCriticalBuilds(controller)

        // Alert operations team
        sendControllerAlert(controller, 'OVERLOAD')
    }
}

def redistributeServices(String overloadedController) {
    def availableControllers = getAvailableControllers().findAll {
        name, info -> name != overloadedController && info.currentLoad < info.maxCapacity * 0.8
    }

    if (availableControllers.isEmpty()) {
        error "No available controllers for redistribution from ${overloadedController}"
    }

    // Implement round-robin redistribution
    def redistributionPlan = createRedistributionPlan(overloadedController, availableControllers)
    applyRedistributionPlan(redistributionPlan)
}
```

2. Dynamic Worker Provisioning Failures

Symptoms:

- Builds failing with "No suitable worker available"
- Worker templates not being instantiated
- Resource quota exceeded errors

Solutions:

```groovy
def handleWorkerProvisioningFailure(String serviceName) {
    echo "Worker provisioning failed for ${serviceName}, attempting recovery..."

    // Try alternative worker templates
    def alternativeTemplates = getAlternativeWorkerTemplates(serviceName)

    alternativeTemplates.each { template ->
        try {
            provisionWorkerWithTemplate(template)
            return true
        } catch (Exception e) {
            echo "Failed to provision worker with template ${template}: ${e.message}"
        }
    }

    // Fallback to general-purpose workers
    echo "Falling back to general-purpose worker for ${serviceName}"
    return provisionGeneralPurposeWorker(serviceName)
}

def getAlternativeWorkerTemplates(String serviceName) {
    def serviceConfig = getServiceConfig(serviceName)
    def buildType = serviceConfig.buildType

    def alternatives = [
        'maven': ['maven-builder-template', 'java-builder-template', 'general-builder-template'],
        'gradle': ['gradle-builder-template', 'java-builder-template', 'general-builder-template'],
        'nodejs': ['nodejs-builder-template', 'node-builder-template', 'general-builder-template']
    ]

    return alternatives[buildType] ?: ['general-builder-template']
}
```

3. Artifact Transfer Issues Between Controllers

Symptoms:

- Stash/unstash operations failing
- Deployment phase missing artifacts
- Inter-controller network timeouts

Solutions:

groovy

```
def handleArtifactTransferFailure(String serviceName, String sourceController) {
    echo "Artifact transfer failed for ${serviceName} from ${sourceController}"

    // Implement retry with exponential backoff
    def maxRetries = 3
    def retryDelay = 30 // seconds

    for (int i = 0; i < maxRetries; i++) {
        try {
            echo "Retry attempt ${i + 1} for artifact transfer: ${serviceName}"

            // Use alternative transfer method
            transferArtifactsViaSharedStorage(serviceName, sourceController)
            return true

        } catch (Exception e) {
            if (i == maxRetries - 1) {
                echo "Final retry failed, triggering rebuild on deployment controller"
                triggerRebuildOnDeploymentController(serviceName)
                return false
            }

            echo "Transfer attempt ${i + 1} failed, retrying in ${retryDelay} seconds..."
            sleep retryDelay
            retryDelay *= 2 // Exponential backoff
        }
    }
}

def transferArtifactsViaSharedStorage(String serviceName, String sourceController) {
    // Use shared network storage as fallback
    sh """
        # Copy artifacts to shared storage
        rsync -avz /var/jenkins_home/jobs/*/builds/${env.BUILD_NUMBER}/archive/ \\
            ${env.SHARED_STORAGE_PATH}/${serviceName}-${env.BUILD_NUMBER}/

        # Verify transfer
        if [ ! -d "${env.SHARED_STORAGE_PATH}/${serviceName}-${env.BUILD_NUMBER}" ]; then
            echo "Shared storage transfer failed"
            exit 1
        fi
```

```
    """
}
```

Monitoring and Alerting Setup

CloudBees Dashboard Configuration

groovy

```groovy
def setupMonitoringDashboards() {
    def dashboardConfig = [
        controllerHealth: [
            metrics: ['cpu_usage', 'memory_usage', 'build_queue_length', 'active_workers'],
            thresholds: [
                cpu_warning: 70,
                cpu_critical: 85,
                memory_warning: 75,
                memory_critical: 90,
                queue_warning: 10,
                queue_critical: 20
            ]
        ],
        serviceMetrics: [
            metrics: ['build_duration', 'success_rate', 'deployment_frequency'],
            timeWindows: ['1h', '24h', '7d', '30d']
        ],
        costOptimization: [
            metrics: ['worker_utilization', 'idle_time', 'resource_efficiency'],
            targets: [
                worker_utilization: 75,
                idle_time_max: 300, // seconds
                cost_per_build: 'track'
            ]
        ]
    ]

    return dashboardConfig
}
```

Alerting Rules

yaml

```yaml
# CloudBees Alerting Configuration
alerts:
  controller_overload:
    condition: "controller.queue_length > 15 OR controller.cpu_usage > 90"
    severity: "critical"
    actions:
      - redistribute_load
      - scale_up_workers
      - notify_ops_team

  build_failure_spike:
    condition: "service.failure_rate > 25% over 1h"
    severity: "warning"
    actions:
      - analyze_failure_patterns
      - check_dependency_health
      - notify_dev_team

  cost_anomaly:
    condition: "cost_per_build > baseline * 1.5"
    severity: "warning"
    actions:
      - analyze_resource_usage
      - check_worker_efficiency
      - notify_finops_team

  security_scan_delays:
    condition: "fortify.avg_duration > 30m OR sonar.avg_duration > 15m"
    severity: "warning"
    actions:
      - check_scanner_resources
      - verify_scan_queues
      - notify_security_team
```

Performance Tuning Guidelines

Controller-Specific Optimizations

groovy

```groovy
def optimizeControllerPerformance(String controllerName) {
    def optimizations = [
        'controller-java-maven': {
            // Optimize Maven settings
            env.MAVEN_OPTS = "-Xmx2g -XX:+UseG1GC -XX:+UseStringDeduplication"
            env.MAVEN_CONFIG = "-Dmaven.repo.local=/opt/maven-cache -T 2C"
        },
        'controller-nodejs': {
            // Optimize Node.js settings
            env.NODE_OPTIONS = "--max-old-space-size=2048"
            env.NPM_CONFIG_CACHE = "/opt/npm-cache"
            env.NPM_CONFIG_PREFER_OFFLINE = "true"
        },
        'controller-security': {
            // Optimize security scanner settings
            env.FORTIFY_MAX_MEMORY = "8g"
            env.SONAR_SCANNER_OPTS = "-Xmx4g"
            env.NEXUS_IQ_TIMEOUT = "900" // 15 minutes
        }
    ]

    def optimization = optimizations[controllerName]
    if (optimization) {
        optimization.call()
        echo "Applied performance optimizations for ${controllerName}"
    }
}
```

Build Cache Strategy

groovy

```
def implementBuildCacheStrategy() {
    def cacheStrategy = [
        maven: [
            local: "/opt/maven-cache",
            remote: "https://nexus.company.com/repository/maven-cache/",
            ttl: "7d"
        ],
        gradle: [
            local: "/opt/gradle-cache",
            remote: "https://gradle-cache.company.com/",
            ttl: "7d"
        ],
        nodejs: [
            local: "/opt/npm-cache",
            remote: "https://npm-cache.company.com/",
            ttl: "3d"
        ]
    ]

    // Configure build tool cache settings
    cacheStrategy.each { buildType, config ->
        configureToolCache(buildType, config)
    }
}
```

Cost Optimization Strategies

Dynamic Resource Scaling

groovy

```groovy
def implementCostOptimization() {
    // Time-based scaling
    def currentHour = new Date().format('HH').toInteger()
    def isBusinessHours = (currentHour >= 8 && currentHour <= 18)
    def isDevelopmentDay = !(new Date().format('E') in ['Sat', 'Sun'])

    if (isBusinessHours && isDevelopmentDay) {
        // Scale up during development hours
        scaleControllers('UP', 1.5)
        enablePreWarmingForHighPriorityServices()
    } else {
        // Scale down during off-hours
        scaleControllers('DOWN', 0.5)
        disablePreWarming()

        // Run only critical builds during off-hours
        filterBuildsByPriority(['critical', 'high'])
    }

    // Weekend maintenance mode
    if (!isDevelopmentDay) {
        enableMaintenanceMode()
    }
}

def scaleControllers(String direction, Double factor) {
    def controllers = getAvailableControllers()

    controllers.each { name, info ->
        def newCapacity = direction == 'UP' ?
            Math.ceil(info.maxCapacity * factor) :
            Math.floor(info.maxCapacity * factor)

        updateControllerCapacity(name, newCapacity.intValue())
    }
}
```

Resource Usage Analytics

```groovy
```

```
def generateCostAnalysisReport() {
    def report = [
        period: "Last 30 days",
        controllerUtilization: [:],
        workerEfficiency: [:],
        costBreakdown: [:],
        recommendations: []
    ]

    // Analyze controller utilization
    def controllers = getAvailableControllers()
    controllers.each { name, info ->
        def utilizationData = getControllerUtilizationData(name, 30)
        report.controllerUtilization[name] = [
            avgUtilization: utilizationData.avgCpuUsage,
            peakUtilization: utilizationData.maxCpuUsage,
            idleTime: utilizationData.idleTime,
            cost: calculateControllerCost(name, utilizationData)
        ]

        // Generate recommendations
        if (utilizationData.avgCpuUsage < 30) {
            report.recommendations.add("Consider downsizing controller ${name} - low utilization")
        }
        if (utilizationData.idleTime > 50) {
            report.recommendations.add("Controller ${name} has high idle time - optimize scheduling")
        }
    }

    // Archive report
    writeJSON file: 'cost-analysis-report.json', json: report
    archiveArtifacts artifacts: 'cost-analysis-report.json'

    return report
}
```

Final Implementation Checklist

Pre-Migration Checklist

- [ ] CloudBees controllers provisioned and configured
- [ ] Dynamic worker templates created and tested
- [ ] Inter-controller networking and security configured

- [ ] Shared libraries developed and tested
- [ ] Monitoring dashboards and alerting configured
- [ ] Backup and rollback procedures documented
- [ ] Team training completed

Migration Execution Checklist

- [ ] Pilot services identified and migrated
- [ ] Security scanning integration validated
- [ ] Deployment pipelines tested end-to-end
- [ ] Performance benchmarks established
- [ ] Rollback procedures tested
- [ ] Documentation updated

Post-Migration Checklist

- [ ] All services successfully migrated
- [ ] Individual repository pipelines decommissioned
- [ ] Performance metrics within acceptable ranges
- [ ] Cost optimization measures implemented
- [ ] Team feedback collected and addressed
- [ ] Lessons learned documented
- [ ] Continuous improvement plan established

Conclusion

The CloudBees CI multi-controller architecture provides unique advantages for MonoRepo implementation that can significantly improve build efficiency, resource utilization, and operational management. The recommended Hybrid Multi-Stage Pipeline approach, enhanced with CloudBees-specific optimizations, offers the best balance of performance, maintainability, and cost-effectiveness.

Key success factors include:

1. **Proper controller specialization** based on service types and requirements
2. **Efficient dynamic worker management** with appropriate resource allocation
3. **Robust monitoring and alerting** to ensure optimal performance
4. **Comprehensive cost optimization** strategies to maximize ROI
5. **Thorough testing and validation** throughout the migration process

With careful planning and execution, this MonoRepo implementation will provide your organization with a scalable, efficient, and cost-effective CICD solution that leverages the full potential of your CloudBees CI

```
investment.gedServices.contains(dependency)) {
dependentServices.add(service)
}
}
}
}

    return dependentServices.toList()

}
```

Security Scanning Adaptations

Fortify App ID Management
```groovy
// vars/executeFortifyScan.groovy
def call(String fortifyAppId, String serviceName) {
    def scanResults = sh(
        script: """
            fortify -appId ${fortifyAppId} \
                -serviceName ${serviceName} \
                -buildId ${env.BUILD_NUMBER} \
                -sourceDir . \
                -resultsFile fortify-results-${serviceName}.xml
        """,
        returnStatus: true
    )

    if (scanResults != 0) {
        error "Fortify scan failed for service: ${serviceName}"
    }

    archiveArtifacts artifacts: "fortify-results-${serviceName}.xml"
    publishHTML([
        allowMissing: false,
        alwaysLinkToLastBuild: true,
        keepAll: true,
        reportDir: '.',
        reportFiles: "fortify-results-${serviceName}.xml",
        reportName: "Fortify Report - ${serviceName}"
    ])
}
```

Service-Specific SonarQube Projects

groovy

```groovy
// vars/executeSonarScan.groovy
def call(String sonarProjectKey, String serviceName) {
    def buildType = detectBuildType()

    switch(buildType) {
        case 'maven':
            sh """
                mvn sonar:sonar \
                    -Dsonar.projectKey=${sonarProjectKey} \
                    -Dsonar.projectName="${serviceName}" \
                    -Dsonar.sources=src/main \
                    -Dsonar.tests=src/test
            """
            break
        case 'gradle':
            sh """
                ./gradlew sonarqube \
                    -Dsonar.projectKey=${sonarProjectKey} \
                    -Dsonar.projectName="${serviceName}"
            """
            break
        case 'nodejs':
            sh """
                sonar-scanner \
                    -Dsonar.projectKey=${sonarProjectKey} \
                    -Dsonar.projectName="${serviceName}" \
                    -Dsonar.sources=src \
                    -Dsonar.tests=test
            """
            break
    }
}
```

Deployment Strategies

Coordinated Deployment with uDeploy

```groovy
```

```groovy
// vars/orchestrateDeployments.groovy
def call(List<String> servicesToDeploy) {
    def config = readYaml file: 'pipeline-config.yml'

    // Group services by deployment order
    def deploymentGroups = servicesToDeploy.groupBy { service ->
        config.services[service].deploymentOrder ?: 999
    }.sort { it.key }

    deploymentGroups.each { order, services ->
        echo "Deploying services with order ${order}: ${services}"

        def deploymentJobs = [:]
        services.each { service ->
            deploymentJobs[service] = {
                deployService(service)
            }
        }

        parallel deploymentJobs

        // Wait for deployment validation
        validateDeployments(services)
    }
}

def deployService(String serviceName) {
    def config = getServiceConfig(serviceName)

    sh """
        udeploy-client \
            -application ${serviceName} \
            -environment ${params.ENVIRONMENT} \
            -version ${env.BUILD_NUMBER} \
            -process "Deploy ${serviceName}"
    """
}
```

Migration Strategy

Phase 1: Preparation (2-3 weeks)

1. **Infrastructure Setup**

- Create monorepo structure in BitBucket
- Develop shared library enhancements
- Create pipeline configuration templates

2. **Pilot Service Selection**
   - Choose 2-3 independent services
   - Services with minimal dependencies
   - Different build types (Maven, Gradle, Node.js)

Phase 2: Pilot Implementation (3-4 weeks)

1. **Migrate Pilot Services**
   - Move selected services to monorepo
   - Implement chosen pipeline approach
   - Test all pipeline stages thoroughly

2. **Validation & Refinement**
   - Compare build times and reliability
   - Validate security scanning integration
   - Test deployment coordination

Phase 3: Gradual Migration (6-8 weeks)

1. **Service Group Migration**
   - Migrate services in logical groups
   - Maintain parallel pipelines during transition
   - Monitor and optimize performance

2. **Complete Migration**
   - Migrate remaining services
   - Decommission individual repositories
   - Update documentation and training

Recommendations

Primary Recommendation: Hybrid Multi-Stage Pipeline (Approach 3)

Rationale:

- Best balance of efficiency and maintainability

- Respects service dependencies
- Optimal resource utilization
- Familiar concepts for teams

Secondary Recommendation: Path-Based Trigger Strategy (Approach 1)

Rationale:

- Easiest migration from current state
- Maintains service isolation
- Can be implemented incrementally
- Lower learning curve

Implementation Considerations

1. Resource Management

- **Build agents**: Ensure sufficient capacity for parallel builds
- **Security scanning**: Configure scan queues to prevent conflicts
- **Nexus repository**: Optimize for concurrent access

2. Monitoring and Observability

- **Pipeline metrics**: Build times, success rates, resource usage
- **Service health**: Deployment success, runtime monitoring
- **Cost tracking**: Resource consumption, scan usage

3. Team Training

- **Pipeline concepts**: MonoRepo vs individual repo workflows
- **Change impact**: Understanding service dependencies
- **Troubleshooting**: Debugging complex pipeline issues

Conclusion

The transition to MonoRepo CICD requires careful planning and phased implementation. The hybrid multi-stage approach provides the best balance of efficiency and maintainability while preserving your existing tool investments and team expertise. Success depends on thorough testing, comprehensive monitoring, and gradual migration with continuous feedback and optimization.

Next Steps

1. **Stakeholder Alignment**: Review approaches with development teams
2. **Tool Validation**: Test integrations with existing infrastructure
3. **Pilot Planning**: Select services and timeline for pilot implementation
4. **Resource Planning**: Ensure adequate infrastructure capacity
5. **Documentation**: Create detailed implementation guides and runbooks