# MonoRepo CICD Pipeline Implementation Guide

## Executive Summary

This document outlines comprehensive approaches for transitioning from individual microservice repositories to MonoRepo-based CICD pipelines while maintaining the existing toolchain (CloudBees CI, SonarQube, Fortify, NexusIQ, uDeploy, PCF).

## Current State Analysis

### Existing Architecture

* **Repository Structure**: Individual repositories per microservice
* **Pipeline Per Service**: Each microservice has dedicated Jenkins pipeline
* **Build Detection**: Automatic build type detection (Maven/Gradle/Node.js)
* **Security Scanning**: Parallel execution of SonarQube, Fortify, NexusIQ
* **Deployment**: uDeploy to PCF platform
* **Infrastructure**: CloudBees CI with multiple controllers and dynamic worker nodes

### Key Challenges in MonoRepo Transition

1. **Change Detection**: Identifying which services need rebuilding
2. **Parallel Execution**: Maintaining efficient build parallelization
3. **Security Scanning**: Managing individual Fortify App IDs per service
4. **Artifact Management**: Handling multiple artifacts from single repository
5. **Deployment Orchestration**: Coordinating deployments across multiple services

## MonoRepo CICD Implementation Approaches

### Approach 5: Your Proposed ChangeSet-Based Orchestration (RECOMMENDED)

#### Implementation Strategy

**Main Orchestration Pipeline**

// Main Pipeline (pipeline-repo/monorepo-orchestrator/Jenkinsfile)  
@Library('monorepo-shared-library') \_  
  
pipeline {  
 agent { label 'orchestrator-node' }  
   
 parameters {  
 booleanParam(  
 name: 'FORCE\_BUILD\_ALL',  
 defaultValue: false,  
 description: 'Force build all services regardless of changes'  
 )  
 string(  
 name: 'SPECIFIC\_SERVICES',  
 defaultValue: '',  
 description: 'Comma-separated list of specific services to build'  
 )  
 }  
   
 environment {  
 PIPELINE\_CONFIG\_REPO = 'https://bitbucket.company.com/pipeline-configs.git'  
 SERVICE\_CONFIG\_FILE = 'config/service-mappings.yml'  
 MAX\_CONCURRENT\_BUILDS = 5  
 }  
   
 stages {  
 stage('Initialize') {  
 steps {  
 script {  
 // Checkout monorepo for analysis  
 checkout scm  
   
 // Checkout pipeline configs  
 dir('pipeline-config') {  
 git url: env.PIPELINE\_CONFIG\_REPO, credentialsId: 'bitbucket-creds'  
 }  
   
 // Load service configurations  
 env.SERVICE\_CONFIG = readFile("pipeline-config/${env.SERVICE\_CONFIG\_FILE}")  
 }  
 }  
 }  
   
 stage('Change Detection') {  
 steps {  
 script {  
 def changedServices = []  
   
 if (params.FORCE\_BUILD\_ALL) {  
 changedServices = getAllServices()  
 } else if (params.SPECIFIC\_SERVICES) {  
 changedServices = params.SPECIFIC\_SERVICES.split(',').collect { it.trim() }  
 } else {  
 changedServices = detectChangedServices()  
 }  
   
 env.CHANGED\_SERVICES = changedServices.join(',')  
 env.BUILD\_SUMMARY = "Building ${changedServices.size()} services: ${changedServices.join(', ')}"  
   
 currentBuild.description = env.BUILD\_SUMMARY  
   
 echo "=== CHANGE DETECTION RESULTS ==="  
 echo "Changed Services: ${env.CHANGED\_SERVICES}"  
 echo "Total Services: ${changedServices.size()}"  
 }  
 }  
 }  
   
 stage('Validate Services') {  
 when {  
 expression { env.CHANGED\_SERVICES != '' }  
 }  
 steps {  
 script {  
 def services = env.CHANGED\_SERVICES.split(',')  
 validateServiceConfigurations(services)  
 }  
 }  
 }  
   
 stage('Trigger Service Builds') {  
 when {  
 expression { env.CHANGED\_SERVICES != '' }  
 }  
 steps {  
 script {  
 def services = env.CHANGED\_SERVICES.split(',').collect { it.trim() }  
 def buildResults = triggerServiceBuilds(services, env.MAX\_CONCURRENT\_BUILDS as Integer)  
   
 // Store results for reporting  
 env.BUILD\_RESULTS = writeJSON returnText: true, json: buildResults  
   
 // Generate status report  
 def overallStatus = generateStatusReport(buildResults)  
   
 // Set build result  
 if (overallStatus != 'SUCCESS') {  
 currentBuild.result = 'FAILURE'  
 error("One or more service builds failed. Check the report above.")  
 }  
 }  
 }  
 }  
 }  
   
 post {  
 always {  
 script {  
 if (env.BUILD\_RESULTS) {  
 archiveArtifacts artifacts: 'build-report.html', fingerprint: true  
 publishHTML([  
 allowMissing: false,  
 alwaysLinkToLastBuild: true,  
 keepAll: true,  
 reportDir: '.',  
 reportFiles: 'build-report.html',  
 reportName: 'MonoRepo Build Report'  
 ])  
 }  
 }  
 }  
   
 failure {  
 emailext (  
 subject: "MonoRepo Build Failed: ${env.JOB\_NAME} - ${env.BUILD\_NUMBER}",  
 body: """  
 Build failed for the following services:  
 ${env.BUILD\_SUMMARY}  
   
 Check the build report: ${env.BUILD\_URL}MonoRepo\_Build\_Report  
 """,  
 to: "${env.CHANGE\_AUTHOR\_EMAIL}"  
 )  
 }  
 }  
}

**Service Configuration YAML**

# pipeline-config/config/service-mappings.yml  
services:  
 user-service:  
 fortifyId: "12345"  
 sonarProjectKey: "user-service"  
 deploymentProfile: "user-profile"  
 path: "services/user-service"  
 buildTool: "maven"  
 owner: "team-alpha@company.com"  
   
 order-service:  
 fortifyId: "12346"  
 sonarProjectKey: "order-service"  
 deploymentProfile: "order-profile"  
 path: "services/order-service"  
 buildTool: "gradle"  
 owner: "team-beta@company.com"  
   
 notification-service:  
 fortifyId: "12347"  
 sonarProjectKey: "notification-service"  
 deploymentProfile: "notification-profile"  
 path: "services/notification-service"  
 buildTool: "nodejs"  
 owner: "team-gamma@company.com"  
  
# Global settings  
settings:  
 defaultTimeout: 3600  
 maxConcurrentBuilds: 5  
 notificationChannels:  
 - email  
 - slack

**Enhanced Change Detection**

// vars/detectChangedServices.groovy  
def call() {  
 def serviceConfig = readYaml text: env.SERVICE\_CONFIG  
 def changedServices = [] as Set  
   
 // Method 1: Use Jenkins ChangeSet API  
 try {  
 def changeSets = currentBuild.changeSets  
 if (changeSets && !changeSets.isEmpty()) {  
 echo "Using ChangeSet API for change detection"  
   
 changeSets.each { changeSet ->  
 changeSet.items.each { change ->  
 echo "Processing change: ${change.msg} by ${change.author}"  
   
 change.affectedPaths.each { path ->  
 def serviceName = findServiceForPath(path, serviceConfig.services)  
 if (serviceName) {  
 changedServices.add(serviceName)  
 echo "Found changed service: ${serviceName} (path: ${path})"  
 }  
 }  
 }  
 }  
 }  
 } catch (Exception e) {  
 echo "Warning: ChangeSet API failed: ${e.message}"  
 }  
   
 // Method 2: Fallback to Git diff if ChangeSet is empty or failed  
 if (changedServices.isEmpty()) {  
 echo "Using Git diff as fallback"  
 try {  
 def gitDiff = sh(  
 script: """  
 if git rev-parse HEAD~1 >/dev/null 2>&1; then  
 git diff --name-only HEAD~1 HEAD  
 else  
 git diff --name-only --cached  
 fi  
 """,  
 returnStdout: true  
 ).trim()  
   
 if (gitDiff) {  
 gitDiff.split('\n').each { path ->  
 def serviceName = findServiceForPath(path, serviceConfig.services)  
 if (serviceName) {  
 changedServices.add(serviceName)  
 echo "Found changed service via git diff: ${serviceName} (path: ${path})"  
 }  
 }  
 }  
 } catch (Exception e) {  
 echo "Warning: Git diff failed: ${e.message}"  
 }  
 }  
   
 // Method 3: Check for shared dependencies  
 def sharedPaths = ['shared/', 'common/', 'libs/']  
 def hasSharedChanges = false  
   
 try {  
 def allChangedFiles = sh(  
 script: "git diff --name-only HEAD~1 HEAD 2>/dev/null || echo ''",  
 returnStdout: true  
 ).trim()  
   
 if (allChangedFiles) {  
 hasSharedChanges = allChangedFiles.split('\n').any { file ->  
 sharedPaths.any { sharedPath -> file.startsWith(sharedPath) }  
 }  
 }  
 } catch (Exception e) {  
 echo "Warning: Shared change detection failed: ${e.message}"  
 }  
   
 if (hasSharedChanges) {  
 echo "Shared dependencies changed - including all services"  
 serviceConfig.services.keySet().each { service ->  
 changedServices.add(service)  
 }  
 }  
   
 def result = changedServices.toList()  
 echo "Final changed services: ${result}"  
 return result  
}  
  
def findServiceForPath(String path, Map services) {  
 return services.find { serviceName, config ->  
 path.startsWith(config.path + "/") || path == config.path  
 }?.key  
}

**Individual Service Pipeline**

// pipeline-repo/service-cicd-pipeline/Jenkinsfile  
@Library('monorepo-shared-library') \_  
  
pipeline {  
 agent none  
   
 parameters {  
 string(name: 'SERVICE\_NAME', description: 'Service name to build')  
 string(name: 'COMMIT\_SHA', description: 'Commit SHA')  
 string(name: 'BUILD\_NUMBER', description: 'Parent build number')  
 }  
   
 environment {  
 SERVICE\_PATH = ""  
 FORTIFY\_APP\_ID = ""  
 BUILD\_TOOL = ""  
 }  
   
 stages {  
 stage('Initialize Service Build') {  
 agent { label 'lightweight' }  
 steps {  
 script {  
 // Load service configuration  
 def serviceConfig = loadServiceConfig(params.SERVICE\_NAME)  
   
 env.SERVICE\_PATH = serviceConfig.path  
 env.FORTIFY\_APP\_ID = serviceConfig.fortifyId  
 env.BUILD\_TOOL = serviceConfig.buildTool  
 env.SONAR\_PROJECT\_KEY = serviceConfig.sonarProjectKey  
   
 currentBuild.description = "Building ${params.SERVICE\_NAME} from ${params.COMMIT\_SHA}"  
   
 echo "=== SERVICE BUILD CONFIGURATION ==="  
 echo "Service: ${params.SERVICE\_NAME}"  
 echo "Path: ${env.SERVICE\_PATH}"  
 echo "Build Tool: ${env.BUILD\_TOOL}"  
 echo "Fortify App ID: ${env.FORTIFY\_APP\_ID}"  
 }  
 }  
 }  
   
 stage('Checkout & Build') {  
 agent {   
 label 'build-agent'  
 customWorkspace "workspace/${params.SERVICE\_NAME}-${params.BUILD\_NUMBER}"  
 }  
 steps {  
 checkout scm  
   
 dir(env.SERVICE\_PATH) {  
 script {  
 buildService(env.BUILD\_TOOL, params.SERVICE\_NAME)  
   
 // Archive build artifacts  
 archiveArtifacts artifacts: getBuildArtifacts(env.BUILD\_TOOL),   
 allowEmptyArchive: true  
 }  
 }  
 }  
 }  
   
 stage('Parallel Security Scans') {  
 parallel {  
 stage('SonarQube Analysis') {  
 agent {   
 label 'sonar-scanner'  
 customWorkspace "workspace/sonar-${params.SERVICE\_NAME}-${params.BUILD\_NUMBER}"  
 }  
 steps {  
 checkout scm  
 dir(env.SERVICE\_PATH) {  
 sonarQubeAnalysis(params.SERVICE\_NAME, env.SONAR\_PROJECT\_KEY, env.BUILD\_TOOL)  
 }  
 }  
 post {  
 always {  
 publishHTML([  
 allowMissing: true,  
 alwaysLinkToLastBuild: false,  
 keepAll: true,  
 reportDir: env.SERVICE\_PATH,  
 reportFiles: 'sonar-report.html',  
 reportName: "SonarQube Report - ${params.SERVICE\_NAME}"  
 ])  
 }  
 }  
 }  
   
 stage('Fortify Analysis') {  
 agent {   
 label 'fortify-scanner'  
 customWorkspace "workspace/fortify-${params.SERVICE\_NAME}-${params.BUILD\_NUMBER}"  
 }  
 steps {  
 checkout scm  
 dir(env.SERVICE\_PATH) {  
 fortifyAnalysis(params.SERVICE\_NAME, env.FORTIFY\_APP\_ID, env.BUILD\_TOOL)  
 }  
 }  
 post {  
 always {  
 publishHTML([  
 allowMissing: true,  
 alwaysLinkToLastBuild: false,  
 keepAll: true,  
 reportDir: env.SERVICE\_PATH,  
 reportFiles: 'fortify-report.html',  
 reportName: "Fortify Report - ${params.SERVICE\_NAME}"  
 ])  
 }  
 }  
 }  
   
 stage('NexusIQ Analysis') {  
 agent {   
 label 'nexus-scanner'  
 customWorkspace "workspace/nexus-${params.SERVICE\_NAME}-${params.BUILD\_NUMBER}"  
 }  
 steps {  
 checkout scm  
 dir(env.SERVICE\_PATH) {  
 nexusIQAnalysis(params.SERVICE\_NAME, env.BUILD\_TOOL)  
 }  
 }  
 post {  
 always {  
 publishHTML([  
 allowMissing: true,  
 alwaysLinkToLastBuild: false,  
 keepAll: true,  
 reportDir: env.SERVICE\_PATH,  
 reportFiles: 'nexus-report.html',  
 reportName: "NexusIQ Report - ${params.SERVICE\_NAME}"  
 ])  
 }  
 }  
 }  
 }  
 }  
   
 stage('Deploy to PCF') {  
 agent { label 'deployment-agent' }  
 steps {  
 checkout scm  
 dir(env.SERVICE\_PATH) {  
 script {  
 def serviceConfig = loadServiceConfig(params.SERVICE\_NAME)  
 deployToPCF(params.SERVICE\_NAME, serviceConfig.deploymentProfile)  
 }  
 }  
 }  
 }  
 }  
   
 post {  
 always {  
 script {  
 // Send status back to main build  
 def status = currentBuild.result ?: 'SUCCESS'  
 echo "=== BUILD COMPLETED ==="  
 echo "Service: ${params.SERVICE\_NAME}"  
 echo "Status: ${status}"  
 echo "Duration: ${currentBuild.durationString}"  
 }  
 }  
   
 failure {  
 emailext (  
 subject: "Service Build Failed: ${params.SERVICE\_NAME}",  
 body: "Build failed for service: ${params.SERVICE\_NAME}\nBuild URL: ${env.BUILD\_URL}",  
 to: getServiceOwner(params.SERVICE\_NAME)  
 )  
 }  
 }  
}

#### Additional Recommendations for Your Approach

1. **Service Discovery Automation**

// vars/autoDiscoverServices.groovy  
def call() {  
 def services = [:]  
   
 // Auto-discover services based on build files  
 def buildFiles = sh(  
 script: """  
 find . -name 'pom.xml' -o -name 'build.gradle' -o -name 'package.json' |   
 grep -E '^\\./services/' |   
 head -50  
 """,  
 returnStdout: true  
 ).trim()  
   
 if (buildFiles) {  
 buildFiles.split('\n').each { buildFile ->  
 def servicePath = new File(buildFile).getParent()  
 def serviceName = servicePath.split('/').last()  
 services[serviceName] = servicePath  
 }  
 }  
   
 return services  
}

1. **Enhanced Status Reporting**

// vars/generateStatusReport.groovy  
def call(buildResults) {  
 def htmlReport = generateHTMLReport(buildResults)  
 def consoleTable = generateConsoleTable(buildResults)  
   
 // Write HTML report  
 writeFile file: 'build-report.html', text: htmlReport  
   
 // Print console table  
 echo consoleTable  
   
 return buildResults.every { k, v -> v.result.result == 'SUCCESS' } ? 'SUCCESS' : 'FAILURE'  
}  
  
def generateHTMLReport(buildResults) {  
 def html = """  
 <!DOCTYPE html>  
 <html>  
 <head>  
 <title>MonoRepo Build Report</title>  
 <style>  
 table { border-collapse: collapse; width: 100%; }  
 th, td { border: 1px solid #ddd; padding: 8px; text-align: left; }  
 th { background-color: #f2f2f2; }  
 .success { background-color: #d4edda; }  
 .failure { background-color: #f8d7da; }  
 .unstable { background-color: #fff3cd; }  
 </style>  
 </head>  
 <body>  
 <h1>MonoRepo Build Report</h1>  
 <p>Build Time: ${new Date()}</p>  
 <table>  
 <thead>  
 <tr>  
 <th>Service</th>  
 <th>Status</th>  
 <th>Duration</th>  
 <th>Build Number</th>  
 <th>Actions</th>  
 </tr>  
 </thead>  
 <tbody>  
 """  
   
 buildResults.each { jobName, result ->  
 def statusClass = result.result.result.toLowerCase()  
 def duration = formatDuration(result.result.duration)  
   
 html += """  
 <tr class="${statusClass}">  
 <td>${result.service}</td>  
 <td>${result.result.result}</td>  
 <td>${duration}</td>  
 <td>${result.result.number}</td>  
 <td>  
 <a href="${result.result.absoluteUrl}" target="\_blank">View Build</a> |  
 <a href="${result.result.absoluteUrl}console" target="\_blank">Console</a>  
 </td>  
 </tr>  
 """  
 }  
   
 html += """  
 </tbody>  
 </table>  
 </body>  
 </html>  
 """  
   
 return html  
}

### Approach 1: Path-Based Change Detection with Matrix Builds

#### Overview

Leverage Jenkins Matrix builds with intelligent path-based change detection to trigger builds only for modified services.

#### Implementation Strategy

**Pipeline Structure:**

// Jenkinsfile (Root Level)  
@Library('shared-library') \_  
  
pipeline {  
 agent none  
   
 stages {  
 stage('Change Detection') {  
 agent { label 'lightweight' }  
 steps {  
 script {  
 def changedServices = detectChangedServices()  
 env.CHANGED\_SERVICES = changedServices.join(',')  
 }  
 }  
 }  
   
 stage('Matrix Build') {  
 when {  
 expression { env.CHANGED\_SERVICES != '' }  
 }  
 matrix {  
 axes {  
 axis {  
 name 'SERVICE'  
 values script {  
 return env.CHANGED\_SERVICES.split(',')  
 }  
 }  
 }  
 stages {  
 stage('Build & Scan') {  
 agent {   
 label 'dynamic-worker'  
 }  
 steps {  
 buildMicroservice(env.SERVICE)  
 }  
 }  
 }  
 }  
 }  
 }  
}

**Shared Library Enhancement:**

// vars/detectChangedServices.groovy  
def call() {  
 def changedFiles = sh(  
 script: "git diff --name-only HEAD~1 HEAD",  
 returnStdout: true  
 ).trim().split('\n')  
   
 def serviceDirectories = [:]  
 def changedServices = [] as Set  
   
 // Map service directories  
 dir('.') {  
 def services = sh(  
 script: "find . -maxdepth 2 -name 'pom.xml' -o -name 'build.gradle' -o -name 'package.json' | xargs dirname | sort -u",  
 returnStdout: true  
 ).trim().split('\n')  
   
 services.each { service ->  
 serviceDirectories[service] = service.replaceAll('^\\.\\/','')  
 }  
 }  
   
 // Detect changed services  
 changedFiles.each { file ->  
 serviceDirectories.each { path, serviceName ->  
 if (file.startsWith(path)) {  
 changedServices.add(serviceName)  
 }  
 }  
 }  
   
 return changedServices.toList()  
}  
  
// vars/buildMicroservice.groovy  
def call(String serviceName) {  
 dir(serviceName) {  
 // Auto-detect build type  
 def buildTool = detectBuildTool()  
   
 // Build stage  
 stage("Build ${serviceName}") {  
 buildWithTool(buildTool, serviceName)  
 }  
   
 // Parallel scanning  
 def scanStages = [:]  
   
 scanStages["SonarQube ${serviceName}"] = {  
 sonarScan(serviceName, buildTool)  
 }  
   
 scanStages["Fortify ${serviceName}"] = {  
 fortifyScan(serviceName, getFortifyAppId(serviceName))  
 }  
   
 scanStages["NexusIQ ${serviceName}"] = {  
 nexusIQScan(serviceName, buildTool)  
 }  
   
 parallel scanStages  
   
 // Deployment  
 stage("Deploy ${serviceName}") {  
 deployToUDeploy(serviceName)  
 }  
 }  
}

**Advantages:**

* Minimal changes to existing shared libraries
* Efficient resource utilization
* Maintains parallel scanning per service
* Scales well with CloudBees CI controllers

**Disadvantages:**

* Complex change detection logic
* Potential for false positives in change detection
* Matrix builds can be resource-intensive

### Approach 2: Multibranch Pipeline with Service-Specific Triggers

#### Overview

Create a sophisticated multibranch pipeline that analyzes changes and creates dynamic pipeline stages for affected services.

#### Implementation Strategy

**Repository Structure:**

monorepo/  
├── services/  
│ ├── user-service/  
│ │ ├── pom.xml  
│ │ └── src/  
│ ├── order-service/  
│ │ ├── build.gradle  
│ │ └── src/  
│ └── notification-service/  
│ ├── package.json  
│ └── src/  
├── shared/  
│ └── common-libraries/  
├── config/  
│ ├── fortify-app-ids.yml  
│ └── service-config.yml  
└── Jenkinsfile

**Configuration Management:**

# config/service-config.yml  
services:  
 user-service:  
 fortifyAppId: "12345"  
 buildTool: "maven"  
 deploymentProfile: "user-profile"  
 sonarProjectKey: "user-service"  
   
 order-service:  
 fortifyAppId: "12346"  
 buildTool: "gradle"  
 deploymentProfile: "order-profile"  
 sonarProjectKey: "order-service"  
   
 notification-service:  
 fortifyAppId: "12347"  
 buildTool: "nodejs"  
 deploymentProfile: "notification-profile"  
 sonarProjectKey: "notification-service"

**Enhanced Pipeline:**

@Library('monorepo-shared-library') \_  
  
pipeline {  
 agent none  
   
 environment {  
 CHANGED\_SERVICES = ""  
 BUILD\_SERVICES = ""  
 }  
   
 stages {  
 stage('Initialize') {  
 agent { label 'lightweight' }  
 steps {  
 script {  
 def analysis = analyzeChanges()  
 env.CHANGED\_SERVICES = analysis.changed.join(',')  
 env.BUILD\_SERVICES = analysis.buildRequired.join(',')  
   
 // Update build description  
 currentBuild.description = "Building: ${env.BUILD\_SERVICES}"  
 }  
 }  
 }  
   
 stage('Parallel Service Builds') {  
 when {  
 expression { env.BUILD\_SERVICES != '' }  
 }  
 steps {  
 script {  
 def buildStages = [:]  
 def servicesToBuild = env.BUILD\_SERVICES.split(',')  
   
 servicesToBuild.each { service ->  
 buildStages["Build ${service}"] = {  
 buildServicePipeline(service.trim())  
 }  
 }  
   
 parallel buildStages  
 }  
 }  
 }  
   
 stage('Integration Tests') {  
 when {  
 expression { env.BUILD\_SERVICES != '' }  
 }  
 agent { label 'integration-test' }  
 steps {  
 runIntegrationTests(env.BUILD\_SERVICES.split(','))  
 }  
 }  
 }  
   
 post {  
 always {  
 publishTestResults testResultsPattern: '\*\*/target/surefire-reports/\*.xml'  
 publishHTML([  
 allowMissing: false,  
 alwaysLinkToLastBuild: true,  
 keepAll: true,  
 reportDir: 'reports',  
 reportFiles: 'index.html',  
 reportName: 'MonoRepo Build Report'  
 ])  
 }  
 }  
}

**Advanced Shared Library:**

// vars/analyzeChanges.groovy  
def call() {  
 def config = readYaml file: 'config/service-config.yml'  
 def changedFiles = getChangedFiles()  
 def changedServices = [] as Set  
 def buildRequired = [] as Set  
   
 // Analyze changed files  
 changedFiles.each { file ->  
 config.services.each { serviceName, serviceConfig ->  
 if (file.startsWith("services/${serviceName}/")) {  
 changedServices.add(serviceName)  
 buildRequired.add(serviceName)  
 }  
 }  
 }  
   
 // Check for shared library changes  
 def sharedChanged = changedFiles.any { it.startsWith('shared/') }  
 if (sharedChanged) {  
 // If shared code changed, rebuild all services  
 buildRequired.addAll(config.services.keySet())  
 }  
   
 return [  
 changed: changedServices.toList(),  
 buildRequired: buildRequired.toList(),  
 sharedChanged: sharedChanged  
 ]  
}  
  
// vars/buildServicePipeline.groovy  
def call(String serviceName) {  
 def config = readYaml file: 'config/service-config.yml'  
 def serviceConfig = config.services[serviceName]  
   
 node('dynamic-worker') {  
 try {  
 checkout scm  
   
 dir("services/${serviceName}") {  
 // Build stage  
 stage("Build ${serviceName}") {  
 buildService(serviceConfig.buildTool, serviceName)  
 }  
   
 // Parallel security scans  
 def scanTasks = [:]  
   
 scanTasks["SonarQube"] = {  
 node('sonar-scanner') {  
 checkout scm  
 dir("services/${serviceName}") {  
 sonarQubeAnalysis(serviceName, serviceConfig)  
 }  
 }  
 }  
   
 scanTasks["Fortify"] = {  
 node('fortify-scanner') {  
 checkout scm  
 dir("services/${serviceName}") {  
 fortifyAnalysis(serviceName, serviceConfig.fortifyAppId)  
 }  
 }  
 }  
   
 scanTasks["NexusIQ"] = {  
 node('nexus-scanner') {  
 checkout scm  
 dir("services/${serviceName}") {  
 nexusIQAnalysis(serviceName, serviceConfig.buildTool)  
 }  
 }  
 }  
   
 stage("Security Scans ${serviceName}") {  
 parallel scanTasks  
 }  
   
 // Deployment  
 stage("Deploy ${serviceName}") {  
 deployService(serviceName, serviceConfig)  
 }  
 }  
 } catch (Exception e) {  
 currentBuild.result = 'FAILURE'  
 throw e  
 }  
 }  
}

**Advantages:**

* Clean separation of concerns
* Configuration-driven approach
* Better resource management
* Supports complex dependency scenarios

**Disadvantages:**

* Requires significant refactoring of existing libraries
* More complex initial setup
* Learning curve for development teams

### Approach 3: Hybrid Pipeline with Conditional Stages

#### Overview

Maintain existing pipeline structure while adding MonoRepo capabilities through conditional stage execution.

#### Implementation Strategy

**Pipeline Framework:**

@Library('hybrid-monorepo-library') \_  
  
pipeline {  
 agent none  
   
 parameters {  
 choice(  
 name: 'EXECUTION\_MODE',  
 choices: ['AUTO\_DETECT', 'ALL\_SERVICES', 'SPECIFIC\_SERVICES'],  
 description: 'Pipeline execution mode'  
 )  
 string(  
 name: 'SPECIFIC\_SERVICES',  
 defaultValue: '',  
 description: 'Comma-separated list of services (when SPECIFIC\_SERVICES mode)'  
 )  
 booleanParam(  
 name: 'FORCE\_BUILD\_ALL',  
 defaultValue: false,  
 description: 'Force build all services regardless of changes'  
 )  
 }  
   
 stages {  
 stage('Repository Analysis') {  
 agent { label 'analysis-node' }  
 steps {  
 script {  
 def analyzer = new MonoRepoAnalyzer()  
 def analysisResult = analyzer.analyze(params)  
   
 env.TARGET\_SERVICES = analysisResult.targetServices.join(',')  
 env.EXECUTION\_PLAN = analysisResult.executionPlan  
   
 // Store analysis results  
 writeJSON file: 'analysis-result.json', json: analysisResult  
 stash includes: 'analysis-result.json', name: 'analysis'  
 }  
 }  
 }  
   
 stage('Service Discovery & Validation') {  
 agent { label 'lightweight' }  
 steps {  
 script {  
 validateServiceConfiguration(env.TARGET\_SERVICES.split(','))  
 }  
 }  
 }  
   
 stage('Parallel Service Processing') {  
 steps {  
 script {  
 executeServiceBuilds()  
 }  
 }  
 }  
   
 stage('Cross-Service Integration') {  
 when {  
 expression {   
 def services = env.TARGET\_SERVICES.split(',')  
 return services.length > 1  
 }  
 }  
 agent { label 'integration-node' }  
 steps {  
 runCrossServiceTests()  
 }  
 }  
 }  
}

**MonoRepo Analyzer Class:**

// src/com/company/MonoRepoAnalyzer.groovy  
package com.company  
  
class MonoRepoAnalyzer {  
   
 def analyze(params) {  
 def result = [  
 targetServices: [],  
 executionPlan: [:],  
 changeAnalysis: [:]  
 ]  
   
 switch(params.EXECUTION\_MODE) {  
 case 'AUTO\_DETECT':  
 result = autoDetectServices(params.FORCE\_BUILD\_ALL)  
 break  
 case 'ALL\_SERVICES':  
 result = getAllServices()  
 break  
 case 'SPECIFIC\_SERVICES':  
 result = getSpecificServices(params.SPECIFIC\_SERVICES)  
 break  
 }  
   
 return result  
 }  
   
 private def autoDetectServices(forceAll) {  
 if (forceAll) {  
 return getAllServices()  
 }  
   
 def changedFiles = sh(  
 script: "git diff --name-only HEAD~1 HEAD || echo ''",  
 returnStdout: true  
 ).trim().split('\n').findAll { it.trim() }  
   
 def services = discoverServices()  
 def affectedServices = [] as Set  
   
 changedFiles.each { file ->  
 services.each { service, path ->  
 if (file.startsWith(path)) {  
 affectedServices.add(service)  
 }  
 }  
 }  
   
 // Check for infrastructure changes  
 def infraChanged = changedFiles.any {   
 it.startsWith('shared/') ||   
 it.startsWith('config/') ||   
 it == 'Jenkinsfile'  
 }  
   
 if (infraChanged && !forceAll) {  
 // Infrastructure changes affect all services  
 affectedServices.addAll(services.keySet())  
 }  
   
 return [  
 targetServices: affectedServices.toList(),  
 executionPlan: createExecutionPlan(affectedServices.toList()),  
 changeAnalysis: [  
 changedFiles: changedFiles,  
 infraChanged: infraChanged  
 ]  
 ]  
 }  
   
 private def createExecutionPlan(services) {  
 def plan = [:]  
   
 services.each { service ->  
 plan[service] = [  
 buildType: detectBuildType(service),  
 fortifyAppId: getFortifyAppId(service),  
 dependencies: getServiceDependencies(service),  
 deploymentConfig: getDeploymentConfig(service)  
 ]  
 }  
   
 return plan  
 }  
}

**Service Build Orchestrator:**

// vars/executeServiceBuilds.groovy  
def call() {  
 def services = env.TARGET\_SERVICES.split(',').findAll { it.trim() }  
   
 if (services.isEmpty()) {  
 echo "No services to build"  
 return  
 }  
   
 def buildGroups = organizeBuildGroups(services)  
   
 buildGroups.each { groupName, groupServices ->  
 stage("Build Group: ${groupName}") {  
 def parallelBuilds = [:]  
   
 groupServices.each { service ->  
 parallelBuilds["${service}"] = {  
 buildServiceWorkflow(service)  
 }  
 }  
   
 parallel parallelBuilds  
 }  
 }  
}  
  
def buildServiceWorkflow(serviceName) {  
 node('dynamic-worker') {  
 def stagePrefix = "[${serviceName}]"  
   
 try {  
 stage("${stagePrefix} Checkout") {  
 checkout scm  
 unstash 'analysis'  
 }  
   
 stage("${stagePrefix} Build") {  
 dir("services/${serviceName}") {  
 def analysisResult = readJSON file: '../analysis-result.json'  
 def serviceConfig = analysisResult.executionPlan[serviceName]  
   
 buildWithConfig(serviceName, serviceConfig)  
 }  
 }  
   
 // Parallel security scanning  
 def scanJobs = [:]  
   
 scanJobs["${stagePrefix} SonarQube"] = {  
 node('sonar-node') {  
 checkout scm  
 dir("services/${serviceName}") {  
 sonarQubeAnalysis(serviceName)  
 }  
 }  
 }  
   
 scanJobs["${stagePrefix} Fortify"] = {  
 node('fortify-node') {  
 checkout scm  
 unstash 'analysis'  
 dir("services/${serviceName}") {  
 def analysisResult = readJSON file: '../analysis-result.json'  
 def appId = analysisResult.executionPlan[serviceName].fortifyAppId  
 fortifyAnalysis(serviceName, appId)  
 }  
 }  
 }  
   
 scanJobs["${stagePrefix} NexusIQ"] = {  
 node('nexus-node') {  
 checkout scm  
 dir("services/${serviceName}") {  
 nexusIQAnalysis(serviceName)  
 }  
 }  
 }  
   
 stage("${stagePrefix} Security Scans") {  
 parallel scanJobs  
 }  
   
 stage("${stagePrefix} Deploy") {  
 deployToEnvironment(serviceName)  
 }  
   
 } catch (Exception e) {  
 currentBuild.result = 'FAILURE'  
 error("Build failed for service: ${serviceName} - ${e.message}")  
 }  
 }  
}

**Advantages:**

* Gradual migration path
* Backward compatibility with existing processes
* Flexible execution modes
* Maintains existing tool integrations

**Disadvantages:**

* Code complexity increases
* Maintenance overhead
* Potential performance impact

### Approach 4: Event-Driven Pipeline with Webhook Integration

#### Overview

Implement an event-driven architecture using Bitbucket webhooks and CloudBees CI API to trigger selective builds based on changed paths.

#### Implementation Strategy

**Webhook Handler:**

// Webhook Pipeline (webhook-handler/Jenkinsfile)  
@Library('webhook-monorepo-library') \_  
  
pipeline {  
 agent { label 'webhook-processor' }  
   
 triggers {  
 bitbucketPush()  
 }  
   
 stages {  
 stage('Process Webhook') {  
 steps {  
 script {  
 def webhookPayload = parseWebhookPayload()  
 def affectedServices = analyzeChangedPaths(webhookPayload)  
   
 if (affectedServices.isEmpty()) {  
 echo "No services affected by this change"  
 return  
 }  
   
 // Trigger individual service builds  
 triggerServiceBuilds(affectedServices, webhookPayload)  
 }  
 }  
 }  
 }  
}  
  
// vars/triggerServiceBuilds.groovy  
def call(affectedServices, webhookData) {  
 def buildJobs = [:]  
   
 affectedServices.each { service ->  
 buildJobs["Trigger ${service}"] = {  
 build job: "monorepo-service-builder",  
 parameters: [  
 string(name: 'SERVICE\_NAME', value: service),  
 string(name: 'COMMIT\_SHA', value: webhookData.commitSha),  
 string(name: 'BRANCH\_NAME', value: webhookData.branchName),  
 booleanParam(name: 'TRIGGERED\_BY\_WEBHOOK', value: true)  
 ],  
 wait: false  
 }  
 }  
   
 parallel buildJobs  
}

**Service Builder Pipeline:**

// Service Builder (monorepo-service-builder/Jenkinsfile)  
@Library('monorepo-shared-library') \_  
  
pipeline {  
 agent none  
   
 parameters {  
 string(name: 'SERVICE\_NAME', description: 'Name of the service to build')  
 string(name: 'COMMIT\_SHA', description: 'Commit SHA to build')  
 string(name: 'BRANCH\_NAME', description: 'Branch name')  
 booleanParam(name: 'TRIGGERED\_BY\_WEBHOOK', defaultValue: false)  
 }  
   
 environment {  
 SERVICE\_PATH = "services/${params.SERVICE\_NAME}"  
 BUILD\_NUMBER\_SUFFIX = "${params.COMMIT\_SHA.take(8)}"  
 }  
   
 stages {  
 stage('Validate Service') {  
 agent { label 'lightweight' }  
 steps {  
 script {  
 validateServiceExists(params.SERVICE\_NAME)  
 }  
 }  
 }  
   
 stage('Build Service') {  
 agent {   
 label 'dynamic-worker'  
 customWorkspace "workspace/monorepo-${params.SERVICE\_NAME}-${env.BUILD\_NUMBER}"  
 }  
 steps {  
 checkoutAtCommit(params.COMMIT\_SHA, params.BRANCH\_NAME)  
   
 dir(env.SERVICE\_PATH) {  
 buildService(params.SERVICE\_NAME)  
 }  
 }  
 }  
   
 stage('Security Scans') {  
 parallel {  
 stage('SonarQube') {  
 agent {   
 label 'sonar-scanner'  
 customWorkspace "workspace/sonar-${params.SERVICE\_NAME}-${env.BUILD\_NUMBER}"  
 }  
 steps {  
 checkoutAtCommit(params.COMMIT\_SHA, params.BRANCH\_NAME)  
 dir(env.SERVICE\_PATH) {  
 sonarAnalysis(params.SERVICE\_NAME)  
 }  
 }  
 }  
   
 stage('Fortify') {  
 agent {   
 label 'fortify-scanner'  
 customWorkspace "workspace/fortify-${params.SERVICE\_NAME}-${env.BUILD\_NUMBER}"  
 }  
 steps {  
 checkoutAtCommit(params.COMMIT\_SHA, params.BRANCH\_NAME)  
 dir(env.SERVICE\_PATH) {  
 fortifyAnalysis(params.SERVICE\_NAME)  
 }  
 }  
 }  
   
 stage('NexusIQ') {  
 agent {   
 label 'nexus-scanner'  
 customWorkspace "workspace/nexus-${params.SERVICE\_NAME}-${env.BUILD\_NUMBER}"  
 }  
 steps {  
 checkoutAtCommit(params.COMMIT\_SHA, params.BRANCH\_NAME)  
 dir(env.SERVICE\_PATH) {  
 nexusAnalysis(params.SERVICE\_NAME)  
 }  
 }  
 }  
 }  
 }  
   
 stage('Deploy') {  
 agent { label 'deployment-agent' }  
 steps {  
 checkoutAtCommit(params.COMMIT\_SHA, params.BRANCH\_NAME)  
 dir(env.SERVICE\_PATH) {  
 deployService(params.SERVICE\_NAME)  
 }  
 }  
 }  
 }  
}

**Advantages:**

* True event-driven architecture
* Immediate response to changes
* Optimal resource utilization
* Clear separation of concerns

**Disadvantages:**

* Complex setup and maintenance
* Requires webhook infrastructure
* Debugging can be challenging

## Migration Strategy & Implementation Plan

### Phase 1: Preparation (Weeks 1-2)

1. **Repository Consolidation**
   * Create monorepo structure
   * Migrate existing repositories
   * Set up shared configuration
2. **Shared Library Enhancement**
   * Extend existing libraries for monorepo support
   * Add change detection capabilities
   * Implement service discovery
3. **Testing Environment Setup**
   * Configure test CloudBees CI controller
   * Set up dynamic worker nodes
   * Test basic monorepo functionality

### Phase 2: Pilot Implementation (Weeks 3-4)

1. **Select Pilot Services**
   * Choose 2-3 non-critical services
   * Implement chosen approach
   * Run parallel builds (old vs new)
2. **Validation & Tuning**
   * Performance testing
   * Security scan validation
   * Deployment verification

### Phase 3: Gradual Rollout (Weeks 5-8)

1. **Batch Migration**
   * Migrate services in groups of 5-10
   * Monitor performance and stability
   * Gather feedback from development teams
2. **Documentation & Training**
   * Create developer guidelines
   * Conduct training sessions
   * Update operational procedures

### Phase 4: Full Migration (Weeks 9-12)

1. **Complete Migration**
   * Migrate remaining services
   * Decommission old pipelines
   * Performance optimization
2. **Monitoring & Support**
   * Set up monitoring dashboards
   * Establish support procedures
   * Create troubleshooting guides

## Best Practices & Recommendations

### Repository Organization

monorepo/  
├── services/ # Individual microservices  
│ ├── service-a/  
│ ├── service-b/  
│ └── service-c/  
├── shared/ # Shared libraries and utilities  
│ ├── common/  
│ ├── testing/  
│ └── infrastructure/  
├── config/ # Configuration files  
│ ├── service-registry.yml  
│ ├── fortify-mappings.yml  
│ └── deployment-configs/  
├── scripts/ # Build and deployment scripts  
├── .jenkins/ # Jenkins-specific configurations  
└── docs/ # Documentation

### Configuration Management

1. **Centralized Configuration**
   * Use YAML files for service metadata
   * Version control all configurations
   * Environment-specific overrides
2. **Security Configuration**
   * Secure storage of Fortify App IDs
   * Encrypted secrets management
   * Access control policies

### Performance Optimization

1. **Build Caching**
   * Implement artifact caching
   * Use Docker layer caching
   * Leverage Nexus for dependency caching
2. **Resource Management**
   * Right-size worker nodes
   * Implement build queuing strategies
   * Monitor resource utilization

### Monitoring & Observability

1. **Build Metrics**
   * Build duration tracking
   * Success/failure rates
   * Resource utilization
2. **Alerting**
   * Failed build notifications
   * Performance degradation alerts
   * Security scan failures

## Tool-Specific Considerations

### CloudBees CI

* **Controller Distribution**: Distribute monorepo builds across multiple controllers
* **Dynamic Workers**: Configure appropriate worker templates for different workloads
* **Pipeline Optimization**: Use pipeline caching and parallel execution

### Bitbucket Integration

* **Webhook Configuration**: Set up path-based webhooks
* **Branch Policies**: Configure merge requirements
* **Permission Management**: Service-specific access controls

### Security Tools Integration

* **SonarQube**: Project keys mapping for individual services
* **Fortify**: App ID management and result aggregation
* **NexusIQ**: Component analysis per service

### Deployment (uDeploy)

* **Application Mapping**: Service to application mapping
* **Environment Management**: Coordinate multi-service deployments
* **Rollback Strategies**: Service-specific rollback capabilities

## Risk Mitigation

### Technical Risks

1. **Build Performance**: Implement incremental builds and caching
2. **Resource Contention**: Monitor and scale infrastructure
3. **Dependency Conflicts**: Use dependency management tools

### Operational Risks

1. **Team Adoption**: Provide training and support
2. **Process Changes**: Gradual migration approach
3. **Rollback Plan**: Maintain parallel systems during transition

### Security Risks

1. **Access Control**: Implement fine-grained permissions
2. **Audit Trail**: Maintain comprehensive logging
3. **Compliance**: Ensure regulatory requirements are met

## Conclusion

The recommended approach is **Approach 2: Multibranch Pipeline with Service-Specific Triggers** for the following reasons:

1. **Scalability**: Handles complex scenarios with multiple services
2. **Maintainability**: Clean, configuration-driven approach
3. **Performance**: Optimal resource utilization
4. **Flexibility**: Supports various build and deployment patterns
5. **Integration**: Works well with existing CloudBees CI infrastructure

This approach provides the best balance of functionality, maintainability, and performance while minimizing risks during the migration process.