# Monorepo CI/CD Orchestrator — Objective, Approach & Flow

## Objective

Enable CI/CD for a **monorepo** where multiple microservices are stored in a single repository, ensuring that: - Only the services that are changed are built, scanned, and deployed. - Each service retains its own independent lifecycle (build → quality/security scans → artifact publish → deployment). - No Jenkinsfiles are stored inside application service code; instead, all CI/CD logic is managed in a **central CI/CD repository** (used by the DevOps team).

## Solution Approach

We will implement an **Orchestrator + Worker Pipelines** model:

1. **Orchestrator Pipeline**
   * Triggered whenever a commit or pull request is made to the monorepo.
   * Detects which services are impacted by the change.
   * Delegates work to corresponding **service worker pipelines**.
2. **Service Worker Pipelines**
   * Pre-created, parameterized Jenkins jobs (one per service, e.g., service\_payments\_job).
   * Reused on each change — new build numbers represent new runs.
   * Executes the lifecycle:
     + Code checkout
     + Build (Maven/Gradle/NodeJS auto-detected)
     + Quality/security scans (SonarQube, Fortify, NexusIQ)
     + Artifact publish to Nexus
     + Deployment via uDeploy to PCF
3. **Metadata Mapping File**
   * A central file (e.g., services.yml) stored in the monorepo.
   * Maintains mappings for each service:
     + Service folder path
     + Fortify App ID
     + Sonar Project Key
     + Nexus coordinates
     + uDeploy application name
4. **Central CI/CD Repository**
   * Holds all Jenkinsfiles (orchestrator + worker templates).
   * Shared libraries for utility methods (e.g., detect changes, parse services.yml).
   * Ensures application teams don’t need to manage Jenkinsfiles.

## Flow Diagram

Below is a **visual diagram** showing how the orchestrator and worker pipelines interact:

|  |
| --- |
| Monorepo CI/CD Orchestrator Flow |

Monorepo CI/CD Orchestrator Flow

**Step-by-step flow:**

1. **Code Commit/PR in Monorepo**  
   ↓
2. **Orchestrator Pipeline runs**
   * Checks out code
   * Identifies changed services using Git diff or Bitbucket API
   * Reads services.yml for metadata ↓
3. **For each changed service:**
   * Triggers the corresponding **Service Worker Pipeline** (e.g., service\_payments\_job)
   * Worker job runs as a new build (Build #1, #2, etc.) ↓
4. **Worker Pipeline executes stages:**
   * Build → Sonar → Fortify → NexusIQ (in parallel where possible) → Publish → Deploy ↓
5. **Results**
   * Artifacts published to Nexus
   * Quality/security dashboards updated per service
   * Deployments triggered via uDeploy

## Example Scenario

**Change in Payments service** 1. Developer commits changes under services/payments/. 2. Orchestrator detects the change → identifies Payments service. 3. Orchestrator triggers **existing job** service\_payments\_job. - If it’s the first time, this job runs Build #1. - Next time there’s another Payments change, it runs Build #2, and so on. 4. Worker pipeline executes: - Build Payments code (Maven) - Run Sonar scan for Payments (Sonar project key = org:payments) - Run Fortify scan (App ID = PAYMENTS-FORTIFY) - Run NexusIQ scan (App ID = PAYMENTS-IQ) - Publish artifact → Nexus - Deploy Payments → PCF via uDeploy app Payments-App

**Change in Payments + Orders services** 1. Developer commits changes in both services/payments/ and services/orders/. 2. Orchestrator detects both changes. 3. Orchestrator triggers service\_payments\_job and service\_orders\_job **in parallel**. 4. Each job runs independently with its own build number.

## Files to be Created

1. **In Monorepo (application code repo)**
   * services.yml → Metadata file for all services.

* Example structure:
* services:  
   payments:  
   path: "services/payments"  
   buildType: "maven"  
   sonarProjectKey: "org:payments"  
   fortifyAppId: "PAYMENTS-FORTIFY"  
   nexusCoordinates: "com.company:payments"  
   udeployApp: "Payments-App"  
    
   orders:  
   path: "services/orders"  
   buildType: "gradle"  
   sonarProjectKey: "org:orders"  
   fortifyAppId: "ORDERS-FORTIFY"  
   nexusCoordinates: "com.company:orders"  
   udeployApp: "Orders-App"  
    
   inventory:  
   path: "services/inventory"  
   buildType: "nodejs"  
   sonarProjectKey: "org:inventory"  
   fortifyAppId: "INVENTORY-FORTIFY"  
   nexusCoordinates: "com.company:inventory"  
   udeployApp: "Inventory-App"

1. **In Central CI/CD Repository (DevOps-owned)**
   * orchestrator.Jenkinsfile → Main orchestrator pipeline logic.
   * worker-template.Jenkinsfile → Standardized pipeline template for service jobs.
   * Shared Library files:
     + scmUtils.groovy → Change detection logic.
     + metaUtils.groovy → Read and provide metadata from services.yml.
     + scanUtils.groovy → Common methods to run Sonar, Fortify, NexusIQ.
     + deployUtils.groovy → Common methods to deploy via uDeploy.

## How Orchestrator Uses services.yml

1. **Detect Changes**
   * Orchestrator checks Git commit differences.
   * Identifies modified folders under services/.
2. **Read Metadata**
   * For each changed service, orchestrator looks up the corresponding section in services.yml.
   * Extracts details like buildType, sonarProjectKey, fortifyAppId, etc.
3. **Trigger Worker Jobs**
   * Passes metadata as parameters to the appropriate worker job (e.g., service\_payments\_job).
   * Example: buildType=maven, fortifyAppId=PAYMENTS-FORTIFY, udeployApp=Payments-App.
4. **Worker Pipeline Execution**
   * Worker pipeline uses passed parameters to:
     + Run the correct build tool (Maven/Gradle/NodeJS).
     + Trigger the right Sonar, Fortify, NexusIQ scans.
     + Publish artifacts to Nexus with correct coordinates.
     + Deploy to the mapped uDeploy app.
5. **Result Aggregation**
   * Orchestrator collects status of all triggered worker jobs.
   * Marks overall pipeline success/failure.

## Key Points to Remember

* **Worker jobs are pre-created once**, then reused on every trigger.
* Each trigger = new build number in that job.
* No Jenkinsfiles live in the application service code.
* Central metadata (services.yml) ensures mapping consistency.
* Orchestrator decides *what to build*; workers execute *how to build*.