Jenkins CI/CD

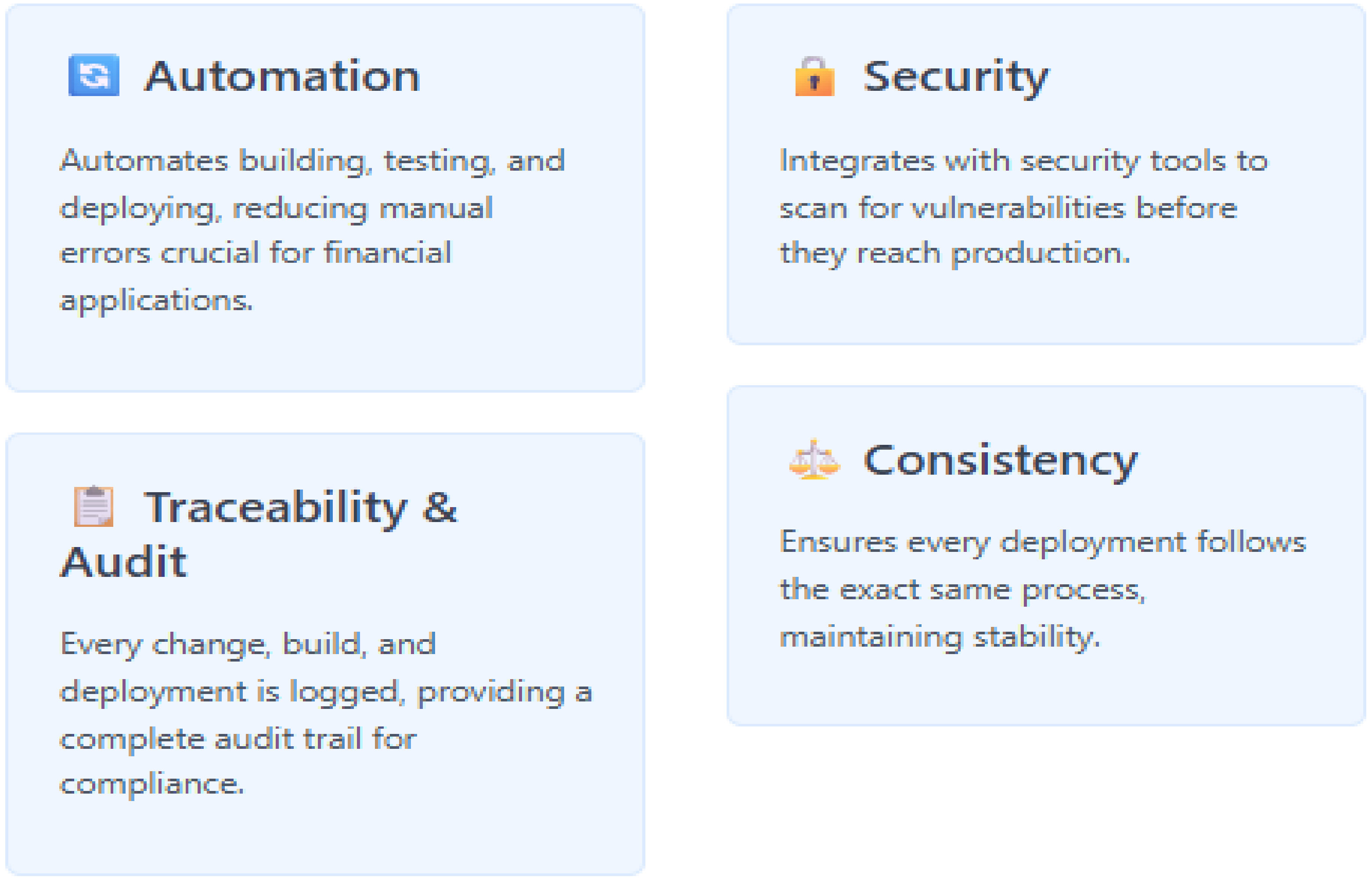
Personal Banking Management Service

A Practical Guide to Automation, Security, and Compliance

What We'll Cover Today

Jenkin-python-project.zip

Why Jenkins?



🔄 Automation

Automates building, testing, and deploying, reducing manual errors crucial for financial applications.

🔒 Security

Integrates with security tools to scan for vulnerabilities before they reach production.

📋 Traceability & Audit

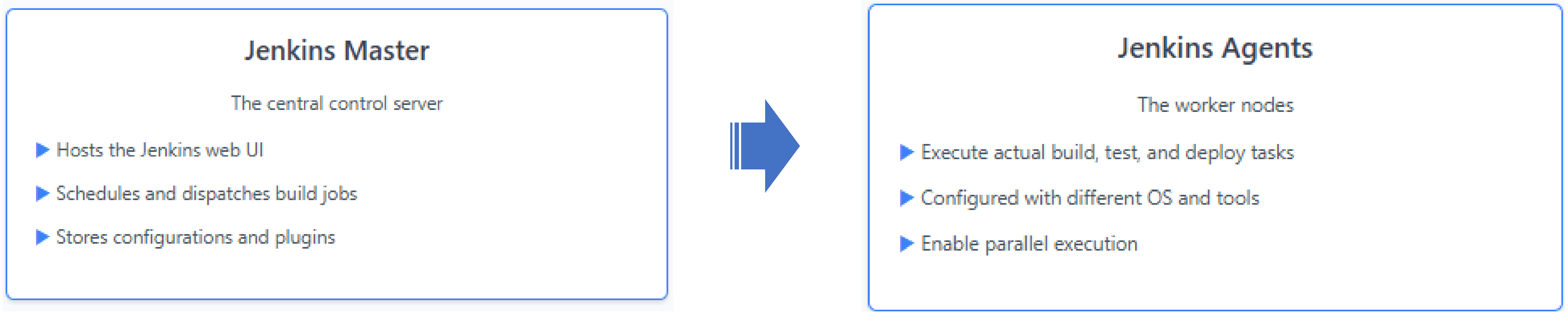
Every change, build, and deployment is logged, providing a complete audit trail for compliance.

⚖️ Consistency

Ensures every deployment follows the exact same process, maintaining stability.

Jenkins Master-Agent Architecture

How Jenkins Operates at FIL?



Jenkins Master

The central control server

- ▶ Hosts the Jenkins web UI
- ▶ Schedules and dispatches build jobs
- ▶ Stores configurations and plugins

Jenkins Agents

The worker nodes

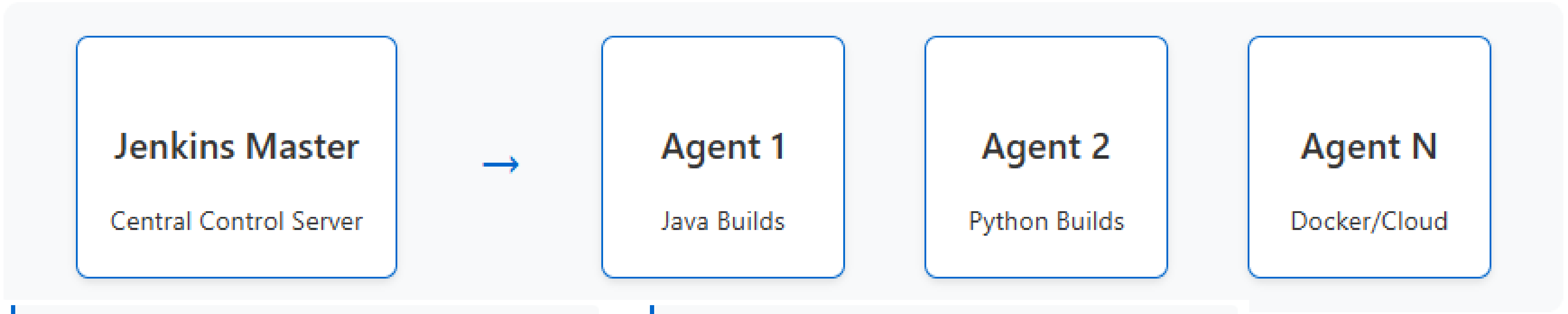
- ▶ Execute actual build, test, and deploy tasks
- ▶ Configured with different OS and tools
- ▶ Enable parallel execution

Pro-Tip: The master should not execute builds itself. Its job is to orchestrate.

Jenkins Master-Agent Architecture

How Jenkins Operates at FIL?

Our setup uses a master-agent (formerly master-slave) architecture.



Jenkins Master

- The central control server
- Hosts the Jenkins web UI
- Schedules and dispatches build jobs for agents
- Stores configurations and plugins

Jenkins Agents

- The worker nodes execute the actual build, test, and deploy tasks
- Can be configured with different operating systems, tools (Java, Python, Docker), and environments
- This allows for parallel execution and specialised build environments

What is a Jenkinsfile?

Pipeline as Code

A Jenkinsfile is a text file that contains the definition of a Jenkins Pipeline and is checked into source control.

Key Benefits

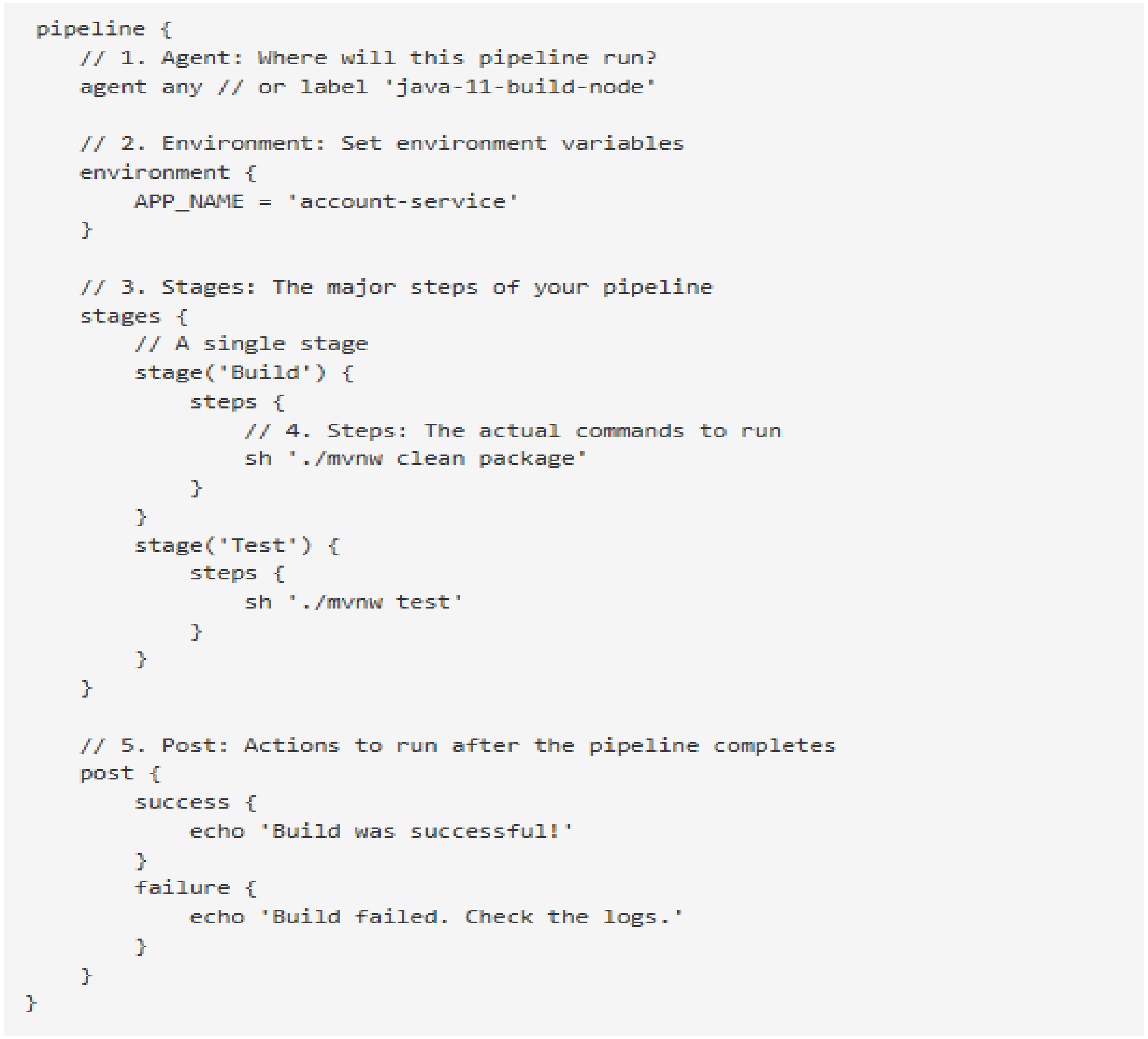
- **Single Source of Truth:** The pipeline is versioned alongside the code it builds
- **Durable & Resumable:** Pipelines can survive Jenkins Master restarts
- **Collaboration:** Allows for code review and iteration on the deployment process itself

Best Practice

We use the modern **Declarative Pipeline** syntax for better structure and readability.

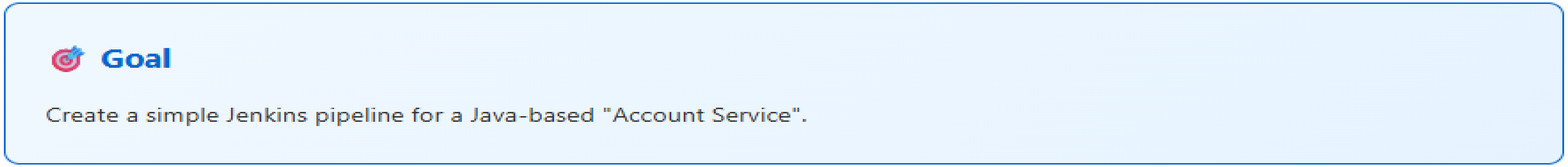
We will use the more modern Declarative Pipeline syntax.

Anatomy of a Declarative Jenkinsfile



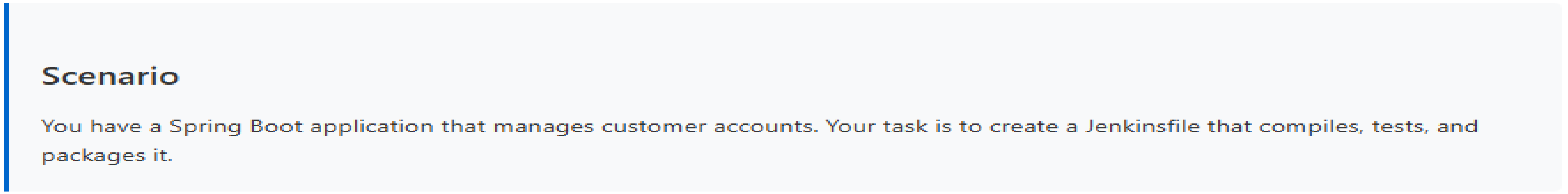
```
pipeline {
    // 1. Agent: Where will this pipeline run?
    agent any // or label 'java-11-build-node'

    // 2. Environment: Set environment variables
    environment {
        APP_NAME = 'account-service'
    }

    // 3. Stages: The major steps of your pipeline
    stages {
        // A single stage
        stage('Build') {
            steps {
                // 4. Steps: The actual commands to run
                sh './mvnw clean package'
            }
        }
        stage('Test') {
            steps {
                sh './mvnw test'
            }
        }
    }

    // 5. Post: Actions to run after the pipeline completes
    post {
        success {
            echo 'Build was successful!'
        }
        failure {
            echo 'Build failed. Check the logs.'
        }
    }
}
```

Exercise 1 - Build a Java Account Service



🎯 **Goal**

Create a simple Jenkins pipeline for a Java-based "Account Service".



Scenario

You have a Spring Boot application that manages customer accounts. Your task is to create a Jenkinsfile that compiles, tests, and packages it.

Setup

- Use a simple Spring Boot project. If you don't have one, use an online initialiser like start.spring.io to create a "Maven" project with "Spring Web"
- Create a file named Jenkinsfile in the root of your project directory

Exercise 1 - Solution Steps

Step 1: Create Your Jenkinsfile

```
 // Jenkinsfile for the Java Account Service
pipeline {
    agent any // Run on any available agent

    stages {
        stage('Clone Repository') {
            steps {
                echo "Cloning the repository..."
                // git url: 'https://your-repo-url/account-service.git', branch: 'main'
            }
        }
        stage('Build Application') {
            steps {
                echo "Building the application..."
                sh './mvnw clean package'
            }
        }
        stage('Run Unit Tests') {
            steps {
                echo "Running unit tests..."
                sh './mvnw test'
            }
        }
        stage('Archive Artifact') {
            steps {
                echo "Archiving the JAR file..."
                archiveArtifacts artifacts: 'target/*.jar', fingerprint: true
            }
        }
    }

    post {
        success {
            echo "Account Service pipeline finished successfully!"
        }
        failure {
            echo "Account Service pipeline failed."
        }
    }
}
```

Jenkins CI/CD Agents

The Right Tool for the Right Job

For our banking service, we'll have multiple microservices (Java, Python). We need different build environments.

Static Agents

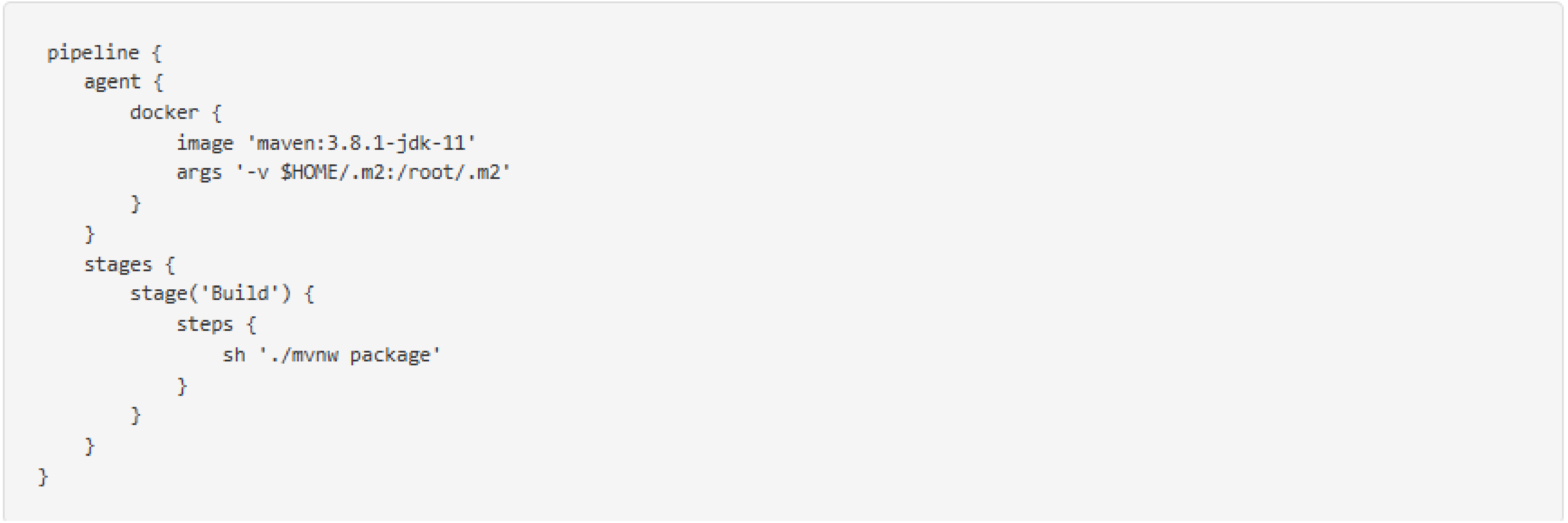
Physical machines or VMs always connected to Jenkins. Good for stable, long-running tasks.

Dynamic Agents (Cloud/Containers)

Agents are provisioned on-demand (e.g., a Docker container or an EC2 instance).

💡 **Pro-Tip:** Use dynamic agents for scalability and clean, ephemeral build environments. This is critical for microservices.

Example: Using a Docker container as an agent



```
pipeline {
    agent {
        docker {
            image 'maven:3.8.1-jdk-11'
            args '-v $HOME/.m2:/root/.m2'
        }
    }
    stages {
        stage('Build') {
            steps {
                sh './mvnw package'
            }
        }
    }
}
```

Managing Secrets

⚠️ **Never Hardcode Credentials!**

Banking applications handle extremely sensitive data (API keys, database credentials, tokens).

Jenkins Credentials Types

- Username with password
- Secret text
- Secret file
- Certificate

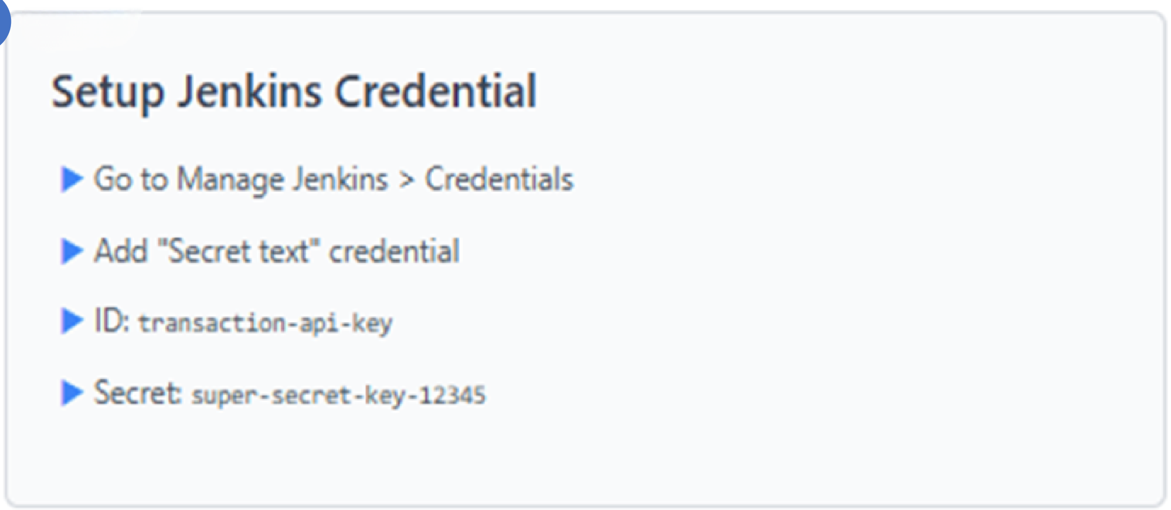
Usage in Jenkinsfile

```
environment {
    // Credential ID from Jenkins Credentials store
    DB_PASSWORD = credentials('prod-db-password-id')
}

// The variable is now available as env.DB_PASSWORD or $DB_PASSWORD
// BUT it will be masked in the logs.
```

Exercise 2 - Python Service with Secrets

Goal: Create a pipeline for a Python "Transaction Service" with secure API key loading

1

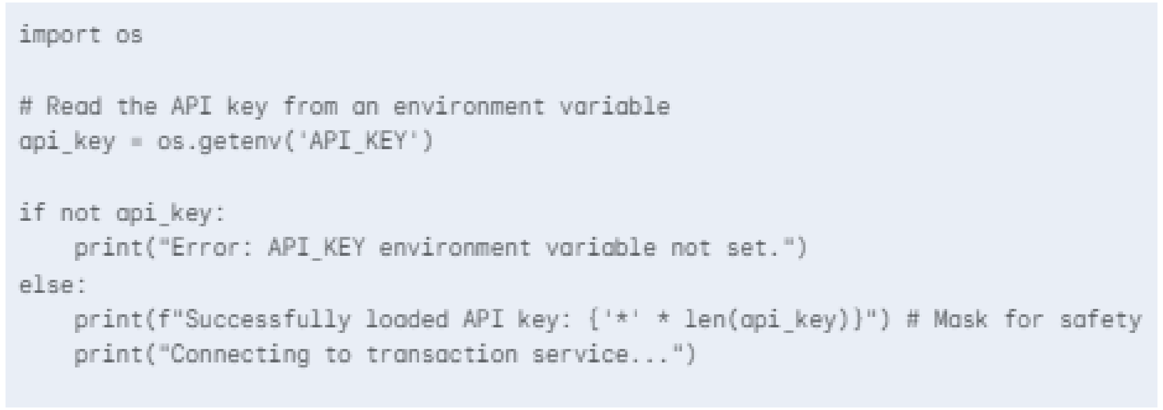


Setup Jenkins Credential

- Go to Manage Jenkins > Credentials
- Add "Secret text" credential
- ID: `transaction-api-key`
- Secret: `super-secret-key-12345`

Create app.py

2

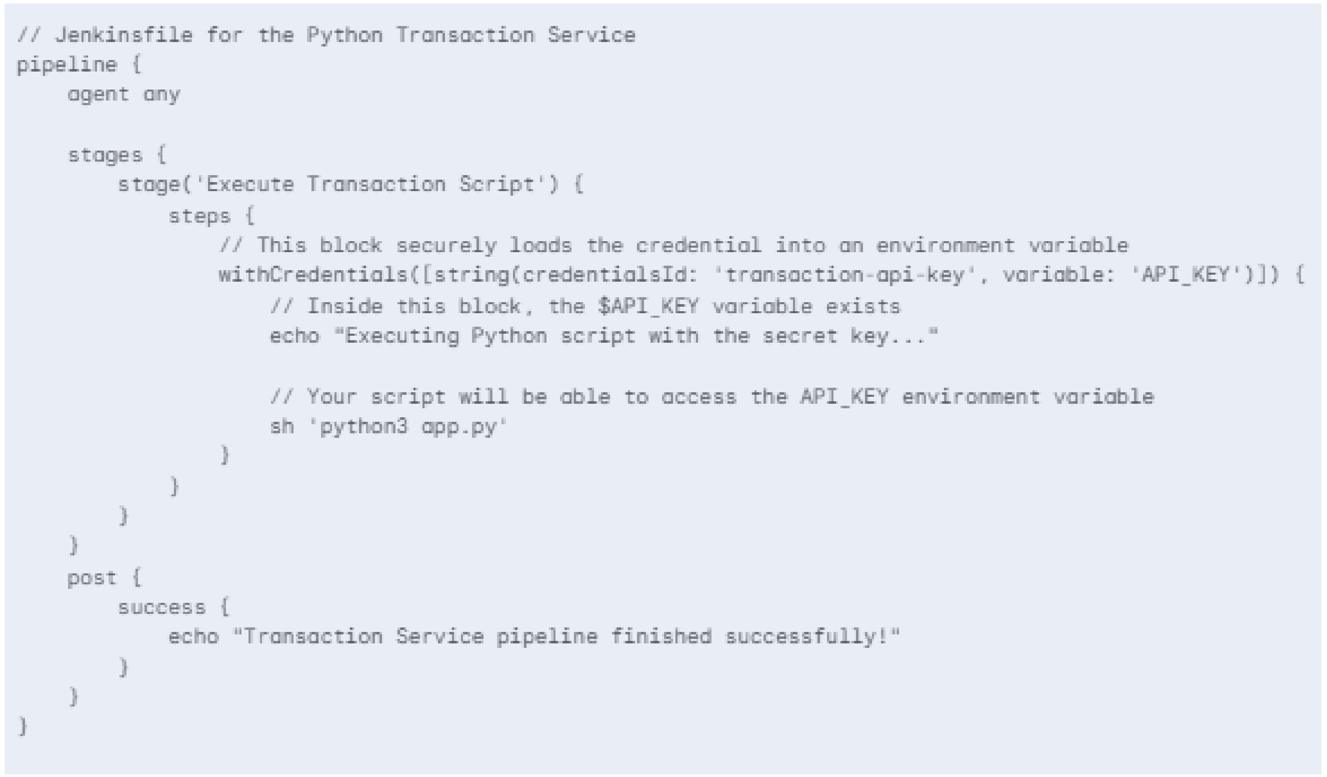


```
import os

# Read the API key from an environment variable
api_key = os.getenv('API_KEY')

if not api_key:
    print("Error: API_KEY environment variable not set.")
else:
    print(f"Successfully loaded API key: {'*' * len(api_key)}") # Mask for safety
    print("Connecting to transaction service...")
```

Exercise 2 - Solution

Step 1: Create Your Jenkinsfile for the Python project

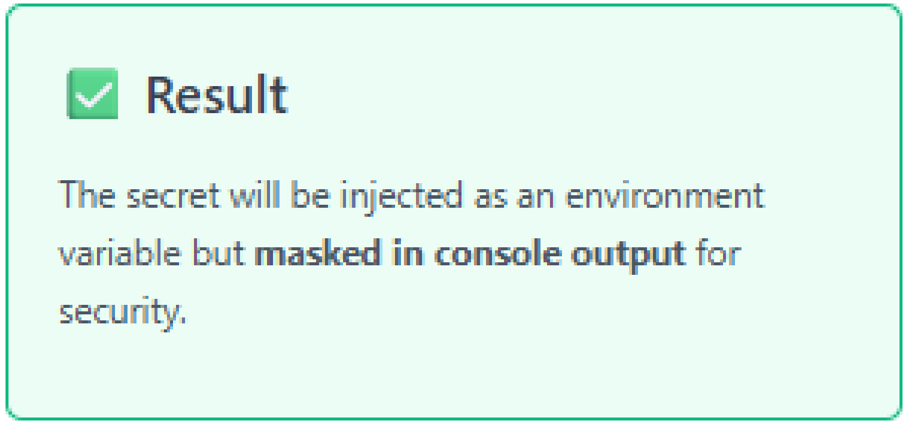
This pipeline uses the Jenkins Credentials plugin to inject the secret.



```
// Jenkinsfile for the Python Transaction Service
pipeline {
    agent any

    stages {
        stage('Execute Transaction Script') {
            steps {
                // This block securely loads the credential into an environment variable
                withCredentials([string(credentialsId: 'transaction-api-key', variable: 'API_KEY')]) {
                    // Inside this block, the $API_KEY variable exists
                    echo "Executing Python script with the secret key..."

                    // Your script will be able to access the API_KEY environment variable
                    sh 'python3 app.py'
                }
            }
        }
    }
    post {
        success {
            echo "Transaction Service pipeline finished successfully!"
        }
    }
}
```

Step 2: Run in Jenkins

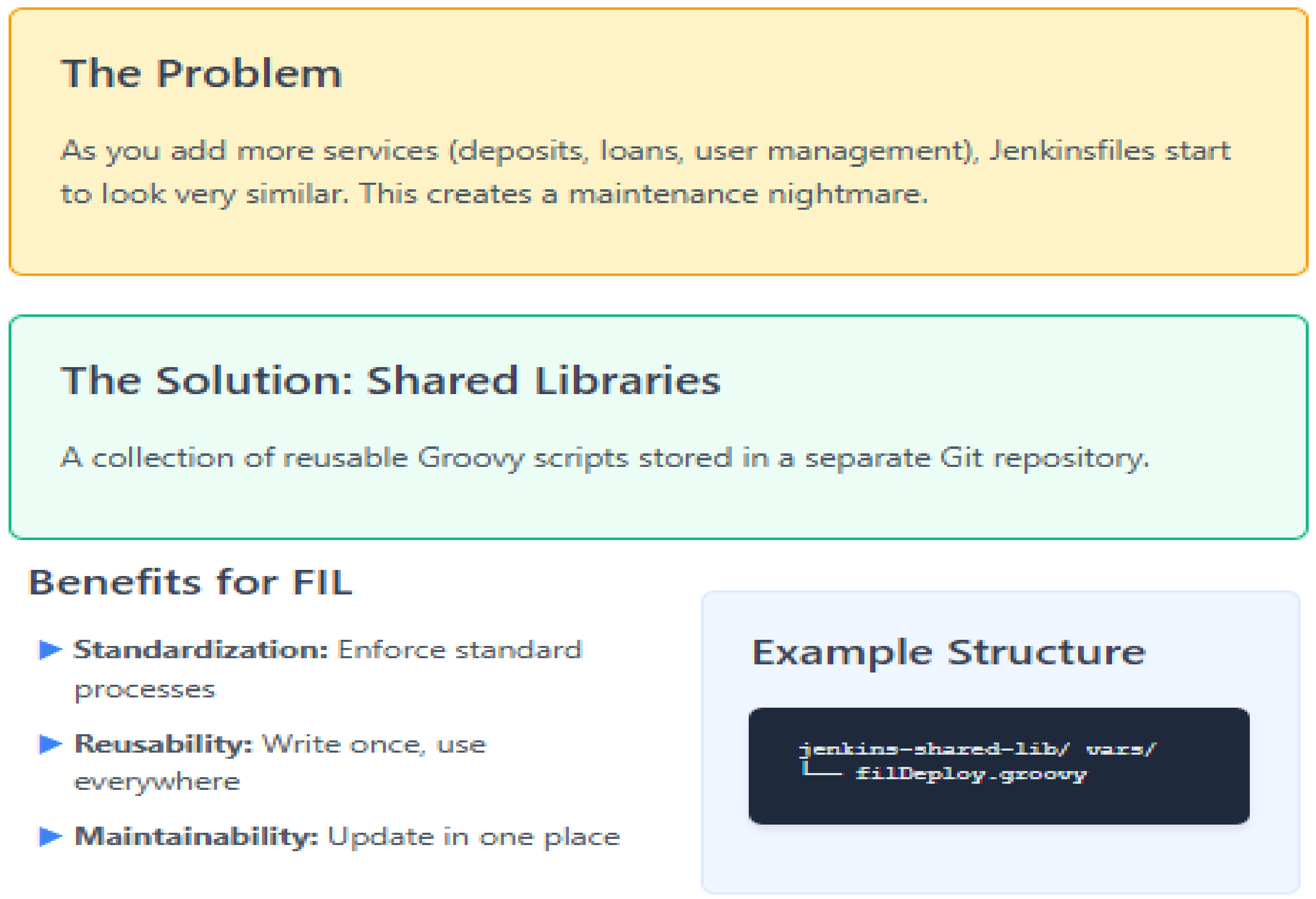
1. Create another "Pipeline" job for this Python service.
2. Configure it to use this new Jenkinsfile from its repository.
3. Click "Build Now".
4. Check the console output. You should see the success message from the Python script, and the secret itself should not be visible.



✅ Result

The secret will be injected as an environment variable but **masked in console output** for security.

Don't Repeat Yourself (DRY)



The Problem

As you add more services (deposits, loans, user management), Jenkinsfiles start to look very similar. This creates a maintenance nightmare.

The Solution: Shared Libraries

A collection of reusable Groovy scripts stored in a separate Git repository.

Benefits for FIL

- ▶ **Standardization:** Enforce standard processes
- ▶ **Reusability:** Write once, use everywhere
- ▶ **Maintainability:** Update in one place

Example Structure

```
jenkins-shared-lib/ vars/
└── filDeploy.groovy
```

Common Troubleshooting Steps

What to do when your pipeline turns RED

1. Check Console Output

Your primary source of information. Look for exact error messages.

2. Blue Ocean Plugin

Modern UI for visualizing pipelines. Clearly shows which stage failed.

Pro-Tip: Replay Feature
Modify your Jenkinsfile directly in the UI and re-run without new commits. Perfect for quick fixes!

3. Workspace Issues

Check build workspace on agent. Use `cleanWs()` to wipe workspace before builds.

Example: Shared Library for FIL

Step 1: Create the Library Repo (jenkins-shared-lib)
Directory structure:

```
vars/
└── filDeploy.groovy   // This defines a global custom step named 'filDeploy'
```

`filDeploy.groovy`:

```
// A simple custom step implementation
def call(String serviceName, String environment) {
    echo "Starting standard FIL deployment for '${serviceName}' to '${environment}'..."
    sh "echo 'Deploying to ${environment}...' > deployment.log"
    // In a real scenario, this would contain kubectl apply, Ansible playbook, etc.
    echo "Deployment completed."
}
```

Step 2: Configure in Jenkins Go to **Manage Jenkins > Configure System > Global Pipeline Libraries** and add your new library.

Step 3: Use it in a Jenkinsfile

```
// Jenkinsfile for any service
@Library('jenkins-shared-lib') _ // Import the library

pipeline {
    agent any
    stages {
        stage('Build') { /* ... build steps ... */ }
        stage('Deploy to Staging') {
            steps {
                // Call our custom step from the shared library!
                filDeploy serviceName: 'account-service', environment: 'staging'
            }
        }
    }
}
```

Compliance as Code

In banking, compliance isn't optional

🔍 Static Code Analysis (SAST)

Use SonarQube to scan for vulnerabilities. Fail builds that don't meet quality gates.

📋 Change Management

Integrate with Jira/ServiceNow. Check for approved change requests before production deployment.

📦 Dependency Scanning (SCA)

Use OWASP Dependency-Check or Snyk for vulnerable libraries.

Best Practice: Fail fast with security scans early in the pipeline.

Example: Enforcing a Quality Gate

```
// ... inside a Jenkinsfile stage
stage('Security and Quality Scan') {
    steps {
        withSonarQubeEnv('Our-SonarQube-Server') {
            sh './mvnw sonar:sonar'
        }
    }
}

stage('Quality Gate Check') {
    steps {
        // This step will wait for SonarQube analysis to complete
        // and will fail the pipeline if the Quality Gate is not 'PASSED'.
        timeout(time: 10, unit: 'MINUTES') {
            waitForQualityGate abortPipeline: true
        }
    }
}

stage('Production Deploy') {
    // This stage will only run if the quality gate passed
    steps {
        echo "All checks passed. Deploying to production."
        // ... deployment logic ...
    }
}
```

Best Practices, Pro-Tips and Key takeaways

✅ Do This

- Keep Master Lean - offload to agents
- Use Declarative Pipelines
- Store Jenkinsfile in Git
- Embrace Dynamic Agents

✅ Security First

- Use Credentials plugin for all secrets
- Run security scans early
- Implement quality gates
- Centralize logic with Shared Libraries

Key Takeaways

- **Jenkins provides** the automation, security, and auditability required to build and deploy Personal Banking Services safely.
- **Jenkinsfiles and Shared Libraries** allow us to treat CI/CD as code - versioned, collaborative, and maintainable.
- **Shift Security Left** by integrating compliance and security checks into the pipeline for more robust applications.

Questions?

Thank you for your attention