**all the functions in groovy pipeline in devops with example**

**🧩 All Major Functions in a DevOps Pipeline (Groovy/Jenkins)**

**1. pipeline**

Defines the start of a declarative pipeline.

pipeline {

    agent any

    stages {

        // stages go here

    }

}

**2. agent**

Specifies where the pipeline or stage should run.

agent any // or agent { label 'linux' }

**3. environment**

Defines environment variables.

environment {

    APP\_NAME = 'MyApp'

    ENV = 'staging'

}

**4. stages and stage**

Defines the sequence of steps in the pipeline.

stages {

    stage('Build') {

        steps {

            echo 'Building...'

        }

    }

}

**5. steps**

Contains the actual commands or actions to perform.

steps {

    sh './gradlew build'

}

**6. script**

Allows use of scripted pipeline syntax inside a declarative pipeline.

steps {

    script {

        def version = "1.0.${env.BUILD\_NUMBER}"

        echo "Version: ${version}"

    }

}

**7. input**

Pauses the pipeline for manual approval.

stage('Approval') {

    steps {

        input message: 'Deploy to production?', ok: 'Yes, deploy'

    }

}

**8. post**

Defines actions to run after the pipeline or a stage completes.

post {

    success {

        echo 'Pipeline succeeded!'

    }

    failure {

        echo 'Pipeline failed!'

    }

}

**9. when**

Adds conditional logic to stages.

stage('Deploy to Prod') {

    when {

        branch 'main'

    }

    steps {

        echo 'Deploying to production...'

    }

}

**10. parallel**

Runs multiple branches of execution in parallel.

stage('Parallel Tests') {

    parallel {

        stage('Unit Tests') {

            steps {

                sh './gradlew test'

            }

        }

        stage('Integration Tests') {

            steps {

                sh './gradlew integrationTest'

            }

        }

    }

}

**11. tools**

Specifies tools like JDK, Maven, etc.

tools {

    maven 'Maven 3.8.1'

    jdk 'JDK 11'

}

**12. parameters**

Defines input parameters for the pipeline.

parameters {

    string(name: 'BRANCH\_NAME', defaultValue: 'main', description: 'Git branch to build')

}

**13. timeout**

Sets a timeout for a stage or step.

options {

    timeout(time: 10, unit: 'MINUTES')

}

**14. retry**

Retries a step if it fails.

steps {

    retry(3) {

        sh 'curl -f http://example.com'

    }

}

**15. catchError**

Catches errors and allows the pipeline to continue.

steps {

    catchError(buildResult: 'UNSTABLE', stageResult: 'FAILURE') {

        sh './gradlew test'

    }

}

Would you like a **complete Jenkinsfile** that uses all of these functions in a real-world example? Or maybe a **diagram** to visualize how these functions fit into a pipeline?

pipeline {

    agent any

    environment {

        APP\_NAME = 'MyApp'

        ARTIFACT\_NAME = 'myapp.jar'

    }

    stages {

        stage('Checkout') {

            steps {

                echo 'Cloning repository...'

                git url: 'https://github.com/your-org/your-repo.git', branch: 'main'

            }

        }

        stage('Build') {

            steps {

                echo 'Building the application...'

                sh './gradlew build'

            }

        }

        stage('Unit Test') {

            steps {

                echo 'Running unit tests...'

                sh './gradlew test'

            }

        }

        stage('Static Code Analysis') {

            steps {

                echo 'Running SonarQube analysis...'

                withSonarQubeEnv('SonarQubeServer') {

                    sh './gradlew sonarqube'

                }

            }

        }

        stage('Archive Artifacts') {

            steps {

                echo 'Archiving build artifacts...'

                archiveArtifacts artifacts: 'build/libs/\*.jar', fingerprint: true

            }

        }

        stage('Deploy to Staging') {

            steps {

                echo 'Deploying to staging environment...'

                sh 'scp build/libs/\*.jar user@staging-server:/opt/apps/'

            }

        }

        stage('Acceptance Tests') {

            steps {

                echo 'Running acceptance tests...'

                sh './gradlew acceptanceTest'

            }

        }

        stage('Approval') {

            steps {

                input message: 'Approve deployment to production?', ok: 'Deploy'

            }

        }

        stage('Deploy to Production') {

            steps {

                echo 'Deploying to production...'

                sh 'scp build/libs/\*.jar user@prod-server:/opt/apps/'

            }

        }

        stage('Notify') {

            steps {

                echo 'Sending notification...'

                slackSend channel: '#deployments', message: "Deployment of ${APP\_NAME} completed successfully!"

            }

        }

    }

    post {

        failure {

            echo 'Pipeline failed. Sending alert...'

            slackSend channel: '#alerts', message: "Pipeline failed for ${APP\_NAME}!"

        }

    }

}

complete Jenkinsfile that uses all of these functions in a real-world example

**Copilot said:**

**Copilot**

Here’s a **complete Jenkinsfile** written in **Declarative Groovy syntax** that demonstrates a real-world DevOps pipeline. It includes **all major functions** like agent, environment, parameters, tools, stages, parallel, input, post, retry, timeout, catchError, and more.

pipeline {

    agent any

    tools {

        maven 'Maven 3.8.1'

        jdk 'JDK 11'

    }

    environment {

        APP\_NAME = 'MyApp'

        DEPLOY\_ENV = 'staging'

        ARTIFACT = 'target/myapp.jar'

    }

    parameters {

        string(name: 'BRANCH\_NAME', defaultValue: 'main', description: 'Git branch to build')

    }

    options {

        timeout(time: 30, unit: 'MINUTES')

        buildDiscarder(logRotator(numToKeepStr: '10'))

    }

    stages {

        stage('Checkout') {

            steps {

                echo "Checking out branch: ${params.BRANCH\_NAME}"

                git branch: "${params.BRANCH\_NAME}", url: 'https://github.com/your-org/your-repo.git'

            }

        }

        stage('Build') {

            steps {

                echo 'Building the application...'

                sh 'mvn clean package -DskipTests'

            }

        }

        stage('Static Code Analysis') {

            steps {

                echo 'Running SonarQube analysis...'

                withSonarQubeEnv('SonarQubeServer') {

                    sh 'mvn sonar:sonar'

                }

            }

        }

        stage('Tests') {

            parallel {

                stage('Unit Tests') {

                    steps {

                        echo 'Running unit tests...'

                        sh 'mvn test'

                    }

                }

                stage('Integration Tests') {

                    steps {

                        echo 'Running integration tests...'

                        sh 'mvn verify -Pintegration'

                    }

                }

            }

        }

        stage('Archive Artifacts') {

            steps {

                echo 'Archiving build artifacts...'

                archiveArtifacts artifacts: "${env.ARTIFACT}", fingerprint: true

            }

        }

        stage('Deploy to Staging') {

            steps {

                echo "Deploying ${env.APP\_NAME} to ${env.DEPLOY\_ENV}..."

                catchError(buildResult: 'UNSTABLE', stageResult: 'FAILURE') {

                    retry(2) {

                        sh "scp ${env.ARTIFACT} user@staging-server:/opt/apps/"

                    }

                }

            }

        }

        stage('Acceptance Tests') {

            steps {

                echo 'Running acceptance tests...'

                sh './scripts/run-acceptance-tests.sh'

            }

        }

        stage('Approval') {

            steps {

                input message: 'Approve deployment to production?', ok: 'Deploy'

            }

        }

        stage('Deploy to Production') {

            when {

                branch 'main'

            }

            steps {

                echo 'Deploying to production...'

                sh "scp ${env.ARTIFACT} user@prod-server:/opt/apps/"

            }

        }

        stage('Notify') {

            steps {

                echo 'Sending deployment notification...'

                slackSend channel: '#deployments', message: "✅ ${env.APP\_NAME} deployed successfully!"

            }

        }

    }

    post {

        success {

            echo 'Pipeline completed successfully.'

        }

        failure {

            echo 'Pipeline failed.'

            slackSend channel: '#alerts', message: "❌ Pipeline failed for ${env.APP\_NAME}!"

        }

        always {

            echo 'Cleaning up workspace...'

            cleanWs()

        }

    }

}

**If You Want to Notify Teammates via Email**

You need to use the **mail** step in Jenkins instead. Here's how:

mail to: 'team@example.com',

     subject: "Deployment Notification: ${env.APP\_NAME}",

     body: "✅ ${env.APP\_NAME} has been successfully deployed to ${env.DEPLOY\_ENV}."

**✅ Example: Send Email Notification**

**📧 How to Enable Email Notifications in Jenkins**

1. **Configure SMTP Settings**:
   * Go to **Manage Jenkins → Configure System**.
   * Scroll to **E-mail Notification** or **Extended E-mail Notification**.
   * Enter your SMTP server details (e.g., Gmail, Outlook, etc.).
   * Set the default sender email address.
2. **Test Email Setup**:
   * Use the **"Test configuration by sending test e-mail"** button.
3. **Use mail in Your Pipeline**:
   * Once configured, you can use the mail step in any stage or post block.

**How to Create a Slack Workspace**

1. **Go to the Slack Website**
   * Visit https://slack.com/get-started
2. **Click on “Create a Workspace”**
   * You’ll be prompted to enter your **email address**.
3. **Verify Your Email**
   * Slack will send you a **confirmation code** to your email.
   * Enter the code to proceed.
4. **Name Your Workspace**
   * Choose a name that reflects your team or organization (e.g., MyCompany, DevTeam, etc.).
5. **Create a Project or Team Name**
   * This is usually the first channel (e.g., #general, #project-x).
6. **Invite Team Members**
   * You can invite others by entering their email addresses.
   * Or skip this step and invite them later.
7. **Start Using Slack**
   * You’ll be taken to your new workspace in the Slack web app.
   * You can also download the **Slack desktop or mobile app** for easier access.

Here’s a **Groovy script** that demonstrates both **arrays (lists)** and **associative arrays (maps)** with common operations:

pipeline {

    agent any

    stages {

        stage('Working with Arrays and Maps') {

            steps {

                script {

                    // ---------- Array (List) ----------

                    def fruits = ['apple', 'banana', 'cherry']

                    echo "Fruits List: ${fruits}"

                    echo "First fruit: ${fruits[0]}"

                    // Add an item

                    fruits << 'orange'

                    echo "Updated Fruits List: ${fruits}"

                    // Loop through list

                    fruits.each { fruit ->

                        echo "Fruit: ${fruit}"

                    }

                    // ---------- Associative Array (Map) ----------

                    def person = [

                        name: 'Alice',

                        age : 28,

                        city: 'Chennai'

                    ]

                    echo "Person Map: ${person}"

                    echo "Name: ${person.name}"

                    echo "City: ${person['city']}"

                    // Add a new key-value pair

                    person.country = 'India'

                    echo "Updated Person Map: ${person}"

                    // Loop through map

                    person.each { key, value ->

                        echo "${key}: ${value}"

                    }

                }

            }

        }

    }

}

**In Jenkins Pipeline**

Jenkins pipelines are designed to run shell commands easily using the sh step (on Unix/Linux agents).

**✅ Example:**

pipeline {

    agent any

    stages {

        stage('Run Bash') {

            steps {

                sh '''

                    #!/bin/bash

                    echo "Running from Jenkins Pipeline"

                    uname -a

                '''

            }

        }

    }

}

**Example: When scm is NOT Default**

If you define your Git repo **inside the pipeline**, like this:

pipeline {

    agent any

    stages {

        stage('Checkout') {

            steps {

                checkout([$class: 'GitSCM',

                    branches: [[name: '\*/main']],

                    userRemoteConfigs: [[url: 'https://github.com/user/repo.git']]

                ])

            }

        }

    }

}

**If we configured git in jenkins**

pipeline {

    agent any

    stages {

        stage('Checkout Code') {

            steps {

                // This pulls the code from the GitHub repo configured in the Jenkins job

                checkout scm

            }

        }

        stage('Build') {

            steps {

                // Run a build command, assuming it's a Maven project

                sh 'mvn clean install'

            }

        }

    }

}

So other jobs (like deployment jobs) can use them give in jenkins pipeline

**Copilot said:**

**Copilot**

To allow **other jobs (like deployment jobs)** to use artifacts archived in a Jenkins pipeline, you typically use a **multi-job pipeline** or **pipeline with downstream jobs**. Jenkins provides a way to **pass artifacts between jobs** using the copyArtifacts plugin.

Here’s a **complete example** using two pipelines:

**✅ Job 1: Build and Archive Artifacts**

This job compiles the code and archives the .jar file.

pipeline {

    agent any

    stages {

        stage('Build') {

            steps {

                sh 'mkdir -p build && echo "dummy jar content" > build/app.jar'

            }

        }

    }

    post {

        success {

            archiveArtifacts artifacts: 'build/\*.jar', fingerprint: true

        }

    }

}

**Job 2: Deploy Using Artifacts from Job 1**

This job **copies the artifact** from Job 1 and uses it for deployment.

pipeline {

    agent any

    stages {

        stage('Fetch Artifact') {

            steps {

                // Copy artifact from Job 1

                copyArtifacts(

                    projectName: 'job-1-name',

                    selector: specific('lastSuccessfulBuild'),

                    filter: 'build/\*.jar',

                    fingerprintArtifacts: true

                )

            }

        }

        stage('Deploy') {

            steps {

                // Simulate deployment

                sh 'echo "Deploying app.jar..." && ls build'

            }

        }

    }

}

**🔧 Requirements**

* Install the **Copy Artifact Plugin** in Jenkins.
* Replace 'job-1-name' with the actual name of your build job.
* Ensure both jobs are configured to allow artifact sharing (e.g., permissions, workspace settings).

**What it does:**

* **copyArtifacts**: This step copies files (artifacts) from another Jenkins job.
* **projectName: 'job-1-name'**: This is the name of the job that originally built and archived the .jar file.
* **selector: specific('lastSuccessfulBuild')**: This tells Jenkins to copy artifacts from the **last successful build** of that job.
* **filter: 'build/\*.jar'**: This specifies which files to copy — in this case, all .jar files inside the build/ directory.
* **fingerprintArtifacts: true**: This enables **tracking** of the artifact across jobs using a unique fingerprint (hash).
* **sh**: Runs a shell command.
* **echo "Deploying app.jar..."**: Simulates a deployment message.
* **ls build**: Lists the contents of the build/ directory to confirm the .jar file was copied successfully.