



What are all concepts regarding jenkinsfile I should know as a 5 years experienced devops engineer? Create complete notes for interview preparation. Include some trending scenario based questions.

Here are complete notes on **Jenkinsfile** for interview preparation, including concepts, syntax, best practices, and scenario-based questions.

Jenkinsfile - Complete Notes for DevOps Interview Preparation

1. Introduction to Jenkinsfile

- **Jenkinsfile** is a text file that defines a Jenkins Pipeline.
- It allows storing pipeline configuration as code, making CI/CD workflows reproducible and version-controlled.
- Supports **Declarative** and **Scripted** pipelines.

2. Types of Jenkins Pipelines

(A) Declarative Pipeline

- High-level syntax with predefined structure.
- Easier to use and maintain.
- Example:

```
groovy

pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building the project...'
            }
        }
        stage('Test') {
            steps {
                echo 'Running tests...'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying the application...'
            }
        }
    }
}
```

(B) Scripted Pipeline

- Uses Groovy scripting for more flexibility.
- Example:

```
groovy

node {
    stage('Build') {
        echo 'Building the project...'
    }
    stage('Test') {
        echo 'Running tests...'
    }
    stage('Deploy') {
        echo 'Deploying the application...'
    }
}
```

3. Important Components of Jenkinsfile

(A) `agent`

- Defines where the pipeline runs.
- Examples:

```
groovy

agent any           // Runs on any available agent
agent none          // Used when defining agents at stage level
agent { label 'docker-node' } // Runs on a specific agent
agent { docker 'maven:3-alpine' } // Runs inside a Docker container
```

(B) `stages` and `steps`

- `stages`: Logical phases of the pipeline.
- `steps`: Actions within a stage.
- Example:

```
groovy

stages {
    stage('Build') {
        steps {
            sh 'mvn clean install'
        }
    }
}
```

(C) `post` Block

- Defines actions to be executed after pipeline execution.
- Example:

```
groovy

post {
    success {
        echo 'Pipeline completed successfully.'
    }
    failure {
        echo 'Pipeline failed!'
    }
}
```

```
}
}
```

(D) `environment` Block

- Defines environment variables.
- Example:

```
groovy

environment {
    AWS_REGION = 'us-east-1'
    DOCKER_IMAGE = 'myapp:latest'
}
```

(E) `parameters`

- Used for user input before pipeline execution.
- Example:

```
groovy

parameters {
    string(name: 'BRANCH', defaultValue: 'main', description: 'Branch to build')
}
```

(F) `tools` Block

- Defines tools required for execution.
- Example:

```
groovy

tools {
    maven 'Maven 3.6.3'
    jdk 'OpenJDK 11'
}
```

4. Advanced Concepts

(A) Parallel Execution

- Runs multiple stages concurrently.
- Example:

```
groovy

stage('Parallel Stages') {
    parallel {
        stage('Test A') {
            steps {
                echo 'Running Test A'
            }
        }
        stage('Test B') {
```

```

        steps {
            echo 'Running Test B'
        }
    }
}

```

(B) Triggering Jobs with `build` Step

- Calls another Jenkins job.
- Example:

```

groovy

steps {
    build job: 'Deploy-Job', parameters: [string(name: 'VERSION', value: '1.0')]
}

```

(C) Using Shared Libraries

- Reuse common logic across multiple pipelines.
- Example:

```

groovy

@Library('my-shared-lib') _
pipeline {
    agent any
    stages {
        stage('Call Shared Function') {
            steps {
                mySharedFunction()
            }
        }
    }
}

```

(D) Error Handling (`try-catch`)

- Used in scripted pipelines.
- Example:

```

groovy

try {
    sh 'exit 1'
} catch (Exception e) {
    echo "Error: ${e}"
}

```

(E) Running Inside Docker

- Example:

```

groovy

agent {
    docker {
        image 'node:14'
    }
}

```

```
args '--network=host'
}
}
```

5. Best Practices

1. **Use Declarative Pipelines** for simplicity and maintainability.
2. **Version Control Jenkinsfile** in the repository.
3. **Modularize using Shared Libraries** to avoid code duplication.
4. **Use Parameters** to make pipelines dynamic.
5. **Implement Proper Error Handling** with ``post`` and ``try-catch``.
6. **Use `parallel` Stages** for faster execution.
7. **Secure Credentials** with Jenkins Credentials Plugin instead of hardcoding.
8. **Limit `agent` any` Usage** to avoid performance issues.

6. Trending Scenario-Based Interview Questions

(A) Basic to Advanced Questions

1. How do you set up a Jenkins pipeline to deploy a microservices-based application?
2. Explain the difference between Declarative and Scripted pipelines.
3. How do you trigger a pipeline from GitHub Webhook?
4. What happens if a stage fails? How do you handle failures in Jenkinsfile?
5. How do you pass variables from one stage to another?

(B) Real-World Scenario-Based Questions

1. Your pipeline needs to build multiple projects in parallel but only deploy once they all succeed. How would you achieve this?
 - o **Solution:** Use ``parallel`` execution for builds and add a dependent ``deploy`` stage after successful execution.
2. You need to run your pipeline inside a Kubernetes pod. How would you define this in Jenkinsfile?
 - o **Solution:**

```
groovy

agent {
    kubernetes {
        yaml '''
        apiVersion: v1
        kind: Pod
        spec:
            containers:
            - name: jnlp
              image: jenkins/inbound-agent
            - name: build
              image: maven:3.8.5
            command:
            - cat
            tty: true
        '''
    }
}
```

```
}  
}
```

3. How do you manage secrets like API keys in Jenkinsfile securely?

- **Solution:** Use Jenkins Credentials Plugin and access them using:

```
groovy  
  
withCredentials([string(credentialsId: 'AWS_SECRET', variable: 'AWS_SECRET_KEY')]) {  
    sh 'echo $AWS_SECRET_KEY'  
}
```

4. How do you deploy an application only when the build and tests pass?

- **Solution:** Use a `post` block:

```
groovy  
  
post {  
    success {  
        build job: 'Deploy-Job'  
    }  
}
```

5. How do you implement a rollback mechanism in Jenkinsfile?

- **Solution:** Store the last successful build version and redeploy if needed.

7. Additional Resources

- **Jenkins Pipeline Documentation** - <https://www.jenkins.io/doc/book/pipeline/>
- **Jenkins Shared Libraries** - <https://www.jenkins.io/doc/book/pipeline/shared-libraries/>
- **Jenkins Credentials Management** - <https://www.jenkins.io/doc/book/using/using-credentials/>

This should give you a **solid foundation** for Jenkinsfile-related interview questions. Would you like me to add **mock interview questions with answers**? 🚀



yes

Here are **mock interview questions with answers** to help you prepare for **Jenkinsfile-related interviews**.

Mock Interview: Jenkinsfile & Jenkins Pipelines

Basic Questions

1. What is a Jenkinsfile? Why use it?

✓ **Answer:**

A **Jenkinsfile** is a text file that defines the pipeline for Jenkins in code form. It allows:

- Storing pipeline as code in version control.
- Easy pipeline replication across environments.
- Better collaboration among teams.

There are two types of pipelines in Jenkinsfile:

1. **Declarative Pipeline** (simpler, structured)
2. **Scripted Pipeline** (more flexible, uses Groovy)

2. What are the key components of a Declarative Pipeline?

✓ **Answer:**

A **Declarative Pipeline** consists of:

- `pipeline {}`: The root block.
- `agent {}`: Defines where the pipeline runs.
- `stages {}`: Contains multiple `stage {}` blocks.
- `steps {}`: Defines commands inside stages.
- `post {}`: Specifies actions to run after success/failure.

Example:

```
groovy

pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'mvn clean install'
            }
        }
    }
    post {
        success {
            echo 'Build completed successfully!'
        }
        failure {
            echo 'Build failed!'
        }
    }
}
```

3. What is the difference between Declarative and Scripted Pipelines?

✓ **Answer:**

Feature	Declarative Pipeline	Scripted Pipeline
Syntax	High-level, structured	Groovy-based, flexible
Complexity	Easier to read/write	More complex
Error Handling	Built-in <code>post</code> block	Requires try-catch

Feature	Declarative Pipeline	Scripted Pipeline
Recommended for	Most use cases	Advanced custom logic

Example of **Scripted Pipeline**:

```
groovy

node {
    stage('Build') {
        sh 'mvn clean install'
    }
}
```

4. How do you pass parameters in Jenkinsfile?

✓ **Answer:**

Use the `parameters` block in the pipeline.

Example:

```
groovy

pipeline {
    agent any
    parameters {
        string(name: 'BRANCH', defaultValue: 'main', description: 'Branch to build')
    }
    stages {
        stage('Checkout') {
            steps {
                git branch: params.BRANCH, url: 'https://github.com/repo.git'
            }
        }
    }
}
```

Intermediate Questions

5. How do you use Jenkins credentials securely in Jenkinsfile?

✓ **Answer:**

Use the **withCredentials** block to access secrets.

Example:

```
groovy

pipeline {
    agent any
    environment {
        AWS_REGION = 'us-east-1'
    }
    stages {
        stage('Use Secrets') {
            steps {
                withCredentials([string(credentialsId: 'AWS_SECRET', variable: 'AWS_SECRET_KEY')]) {
                    sh 'echo "Using AWS Secret: $AWS_SECRET_KEY"'
                }
            }
        }
    }
}
```



```

    }
  }
}

```

✓ This ensures credentials are **not exposed** in logs.

6. How do you run parallel stages in Jenkinsfile?

✓ Answer:

Use the **parallel** block.

Example:

```

groovy

pipeline {
  agent any
  stages {
    stage('Parallel Tasks') {
      parallel {
        stage('Test A') {
          steps {
            echo 'Running Test A'
          }
        }
        stage('Test B') {
          steps {
            echo 'Running Test B'
          }
        }
      }
    }
  }
}

```

✓ Useful for **faster execution**.

7. How do you trigger another Jenkins job inside a pipeline?

✓ Answer:

Use the `build` step.

Example:

```

groovy

pipeline {
  agent any
  stages {
    stage('Trigger Job') {
      steps {
        build job: 'Deploy-Job', parameters: [string(name: 'VERSION', value: '1.0')]
      }
    }
  }
}

```

Advanced Scenario-Based Questions

8. Your build pipeline needs to execute only on changes to `src/` or `pom.xml`. How do you achieve this?

✓ **Answer:**

Use the `when` condition with `changeset`.

Example:

```
groovy

pipeline {
    agent any
    stages {
        stage('Build') {
            when {
                changeset "src/**, pom.xml"
            }
            steps {
                sh 'mvn clean package'
            }
        }
    }
}
```

✓ This prevents unnecessary builds.

9. Your application deployment should run inside a Docker container. How do you define this in Jenkinsfile?

✓ **Answer:**

Use the `agent { docker }` block.

Example:

```
groovy

pipeline {
    agent {
        docker {
            image 'node:14'
            args '--network=host'
        }
    }
    stages {
        stage('Deploy') {
            steps {
                sh 'node app.js'
            }
        }
    }
}
```

✓ Runs the deployment inside a **Node.js 14** container.

10. How do you deploy an application only if the tests pass?

✓ Answer:

Use the `post` block with `success`.

Example:

```
groovy

pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'mvn clean install'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
    }
    post {
        success {
            build job: 'Deploy-Job'
        }
    }
}
```

✓ Ensures **deployment only happens on success**.

11. How do you implement a rollback in case of a failed deployment?

✓ Answer:

Use `post` block with `failure`, store the last successful version, and redeploy it.

Example:

```
groovy

pipeline {
    agent any
    environment {
        LAST_SUCCESSFUL_VERSION = '1.0.0'
    }
    stages {
        stage('Deploy') {
            steps {
                sh 'deploy-app.sh 2.0.0'
            }
        }
    }
    post {
        failure {
            echo "Deployment failed. Rolling back..."
            sh "deploy-app.sh $LAST_SUCCESSFUL_VERSION"
        }
    }
}
```

✓ **Automatic rollback** on failure.

12. Your Jenkins pipeline should build feature branches but deploy only for `main`. How do you achieve this?

✅ Answer:

Use `when` condition.

Example:

```
groovy

pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'mvn clean install'
            }
        }
        stage('Deploy') {
            when {
                branch 'main'
            }
            steps {
                sh 'deploy-app.sh'
            }
        }
    }
}
```

✅ Ensures deployment **only** for `main` branch.

Bonus: Debugging & Optimization

13. How do you debug a failing Jenkins pipeline?

✅ Answer:

- Check **Jenkins logs** (`Manage Jenkins` → `System Log`).
- Add **debugging logs** inside the pipeline:

```
groovy

steps {
    sh 'set -x' // Enables shell debugging
    sh 'echo "Debug Info: $BUILD_NUMBER"'
}
```

- Run **stages separately** using:

```
groovy

when { expression { env.DEBUG == 'true' } }
```

- Check **workspace and logs** in `/var/lib/jenkins/workspace/`.

14. How do you optimize a slow pipeline?

✓ Answer:

1. Use **parallel execution** (``parallel`` block).
2. **Cache dependencies** using ``stash`/`unstash``.
3. **Minimize unnecessary builds** (``when`` conditions).
4. **Run in lightweight Docker containers**.
5. **Reduce workspace cleanup frequency**.

Conclusion

This **mock interview set** covers all levels of **Jenkinsfile concepts**. Would you like to practice **live coding challenges** for Jenkins pipelines? 🚀



yes

Great! Here are some **live coding challenges** for Jenkinsfile. Try solving them, and I can review your solutions! 🚀

Live Coding Challenges for Jenkinsfile

Challenge 1: Simple Build Pipeline

Task:

Write a Jenkinsfile that:

- Runs on any available agent.
- Has a stage named **"Build"** that prints ``"Building the project..."``.
- Has a stage named **"Test"** that runs a shell command ``"echo Running tests..."``.

♦ Expected Output:

```
sql

Building the project...
Running tests...
```

Challenge 2: Parameterized Pipeline

Task:

Create a Jenkinsfile that:

- Takes a **string parameter** ``BRANCH_NAME`` (default: ``main``).
- Clones the Git repository using ``git`` command.
- Prints ``"Building branch: <BRANCH_NAME>"``.

♦ Example Run:

yaml

Building branch: feature-xyz

Challenge 3: Parallel Test Execution

Task:

Write a pipeline that:

- Runs tests in parallel:
 - **Test A:** Prints ``"Running Test A"`.`
 - **Test B:** Prints ``"Running Test B"`.`

♦ Expected Output (in parallel):

css

```
Running Test A
Running Test B
```

Challenge 4: Deploy Only If Tests Pass

Task:

- Write a Jenkins pipeline with three stages:
 1. **Build** (``echo "Building project"`.`)
 2. **Test** (``exit 1`` if tests fail, ``echo "Tests passed"`.` otherwise)
 3. **Deploy** (Only runs if Build & Test are successful)

♦ Expected Behavior:

- If ``Test`` fails, **Deploy should not run.**
- If ``Test`` passes, **Deploy should run.**

Challenge 5: Secure Credentials Handling

Task:

- Store a secret key in Jenkins credentials (ID: ``SECRET_KEY``).
- Access it securely in the pipeline.
- Print ``"Using secret key: *****`` (masking actual key).

♦ Expected Output:

vbnet

```
Using secret key: ****
```

(Secret should never be printed in logs!)

Challenge 6: Trigger Another Job

Task:

Write a Jenkinsfile that:

- Runs a stage called `"Trigger Deploy Job"`.
- Calls another Jenkins job `"Deploy-App"` with parameter `VERSION = 2.0`.

Expected Behavior:

- The **Deploy-App** job should start after this pipeline runs.
-

Challenge 7: Run in Docker

Task:

- Modify your pipeline to run inside a **Node.js 16** Docker container.
- Print the **Node.js version** using `node -v`.

Expected Output:

```
yaml
```

```
Node.js version: v16.x.x
```

Challenge 8: Build Only for Specific Changes

Task:

- Modify the pipeline so it **only runs when**:
 - `src/**` or `pom.xml` is modified.

Expected Behavior:

- If changes are in `README.md`, pipeline **should not run**.
 - If `src/main.py` is changed, pipeline **should run**.
-

Challenge 9: Rollback on Failure

Task:

- Deploy an app (`deploy-app.sh`).
- If deployment **fails**, roll back to the last successful version (`rollback.sh`).

Expected Behavior:

- On success: **No rollback**.
 - On failure: **Rollback should trigger**.
-

Challenge 10: Scheduled Nightly Builds

Task:

- Schedule your Jenkins pipeline to run **daily at 2 AM** using ``cron`` syntax.

♦ Expected Behavior:

- Pipeline should **automatically run every night at 2 AM**.

Ready to Solve?

Try solving these challenges! 🏆 Paste your solutions here, and I'll review them for **best practices, security, and efficiency!** 🚀 🔥



write down the solutions

Here are the **solutions** for all **10 Jenkinsfile challenges**, written using **Declarative Pipeline syntax** for clarity. 🚀

✅ Challenge 1: Simple Build Pipeline

```
groovy

pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building the project...'
            }
        }
        stage('Test') {
            steps {
                sh 'echo Running tests...'
            }
        }
    }
}
```

✅ Challenge 2: Parameterized Pipeline

```
groovy

pipeline {
    agent any
    parameters {
        string(name: 'BRANCH_NAME', defaultValue: 'main', description: 'Branch to build')
    }
    stages {
        stage('Checkout') {
            steps {
                git branch: params.BRANCH_NAME, url: 'https://github.com/repo.git'
            }
        }
    }
}
```



```

    stage('Build') {
        steps {
            echo "Building branch: ${params.BRANCH_NAME}"
        }
    }
}

```

✓ Challenge 3: Parallel Test Execution

```

groovy

pipeline {
    agent any
    stages {
        stage('Run Tests') {
            parallel {
                stage('Test A') {
                    steps {
                        echo 'Running Test A'
                    }
                }
                stage('Test B') {
                    steps {
                        echo 'Running Test B'
                    }
                }
            }
        }
    }
}

```

✓ Challenge 4: Deploy Only If Tests Pass

```

groovy

pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building project'
            }
        }
        stage('Test') {
            steps {
                script {
                    def testResult = sh(script: 'echo "Running Tests" && exit 0', returnStatus: true)
                    if (testResult != 0) {
                        error 'Tests failed, stopping pipeline.'
                    }
                }
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying application...'
            }
        }
    }
}

```

✓ Challenge 5: Secure Credentials Handling

```
groovy

pipeline {
  agent any
  stages {
    stage('Use Secret') {
      steps {
        withCredentials([string(credentialsId: 'SECRET_KEY', variable: 'SECRET')]) {
          sh 'echo "Using secret key: ****"'
        }
      }
    }
  }
}
```

✓ Secret key is masked!

✓ Challenge 6: Trigger Another Job

```
groovy

pipeline {
  agent any
  stages {
    stage('Trigger Deploy Job') {
      steps {
        build job: 'Deploy-App', parameters: [string(name: 'VERSION', value: '2.0')]
      }
    }
  }
}
```

✓ Challenge 7: Run in Docker

```
groovy

pipeline {
  agent {
    docker {
      image 'node:16'
    }
  }
  stages {
    stage('Check Node.js Version') {
      steps {
        sh 'node -v'
      }
    }
  }
}
```

♦ Runs inside a Node.js 16 Docker container.

✓ Challenge 8: Build Only for Specific Changes

```
groovy

pipeline {
  agent any
  stages {
    stage('Build') {
      when {
        changeset "src/**, pom.xml"
      }
      steps {
        echo 'Code changes detected, running build...'
        sh 'mvn clean install'
      }
    }
  }
}
```

✓ Prevents unnecessary builds.

✓ Challenge 9: Rollback on Failure

```
groovy

pipeline {
  agent any
  environment {
    LAST_SUCCESSFUL_VERSION = '1.0.0'
  }
  stages {
    stage('Deploy') {
      steps {
        script {
          def deployStatus = sh(script: './deploy-app.sh 2.0.0', returnStatus: true)
          if (deployStatus != 0) {
            echo "Deployment failed. Rolling back..."
            sh "./rollback.sh ${env.LAST_SUCCESSFUL_VERSION}"
          }
        }
      }
    }
  }
}
```

✓ Automatic rollback on failure.

✓ Challenge 10: Scheduled Nightly Builds

```
groovy

pipeline {
  agent any
  triggers {
    cron('0 2 * * *') // Runs at 2 AM daily
  }
  stages {
    stage('Nightly Build') {
      steps {
        echo 'Running nightly build...'
      }
    }
  }
}
```

✓ Runs every day at 2 AM.

🚀 Next Steps

Would you like **more advanced** Jenkins challenges? Or do you need **custom scenarios** for your **real-world project**? Let me know! 😊🔥



yes provide custom scenarios with solution and also write down rollback.sh

🚀 Custom Jenkinsfile Scenarios with Solutions

Here are some **advanced real-world Jenkins pipeline scenarios** with solutions, plus the ``rollback.sh`` script! 🔥

✓ Scenario 1: Multi-Branch Build with Conditional Deployment

📌 Scenario:

- The pipeline should build **any branch**.
- Deployment should **only happen for** ``main`` **or** ``release`` **branches**.
- Notify the team if the deployment succeeds.

Jenkinsfile Solution

```
groovy

pipeline {
    agent any
    parameters {
        string(name: 'BRANCH_NAME', defaultValue: 'main', description: 'Branch to build')
    }
    stages {
        stage('Checkout') {
            steps {
                git branch: params.BRANCH_NAME, url: 'https://github.com/repo.git'
            }
        }
        stage('Build') {
            steps {
                echo "Building branch: ${params.BRANCH_NAME}"
                sh 'mvn clean install'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Deploy') {
            when {
                anyOf {
                    branch 'main'
                    branch 'release'
                }
            }
            steps {
                echo 'Deploying application...'
            }
        }
    }
}
```

```

        sh './deploy-app.sh'
    }
}
}
post {
    success {
        mail to: 'devops-team@example.com',
            subject: "Deployment Successful",
            body: "Deployment completed successfully for branch: ${params.BRANCH_NAME}"
        }
    }
}
}

```

✓ Ensures only `main` or `release` branches are deployed.

✓ Notifies the team on success.

✓ Scenario 2: Rolling Deployment with Health Check

🚩 Scenario:

- Deploy the new version.
- Run a **health check** to verify the deployment.
- If the health check **fails**, **rollback to the previous version** using `rollback.sh`.

Jenkinsfile Solution

```

groovy

pipeline {
    agent any
    environment {
        LAST_SUCCESSFUL_VERSION = '1.0.0'
    }
    stages {
        stage('Deploy New Version') {
            steps {
                script {
                    def deployStatus = sh(script: './deploy-app.sh 2.0.0', returnStatus: true)
                    if (deployStatus != 0) {
                        error "Deployment failed!"
                    }
                }
            }
        }
        stage('Health Check') {
            steps {
                script {
                    def healthStatus = sh(script: 'curl -s -o /dev/null -w "%{http_code}" http://api', returnStatus: true)
                    if (healthStatus != '200') {
                        error "Health check failed!"
                    }
                }
            }
        }
    }
    post {
        failure {
            echo "Deployment or Health Check failed. Rolling back..."
            sh "./rollback.sh ${env.LAST_SUCCESSFUL_VERSION}"
        }
    }
}

```

- ✓ Automatically rolls back on failure.
- ✓ Ensures only healthy deployments are active.

`rollback.sh` Script

```
bash

#!/bin/bash

# Capture the version to roll back to
VERSION=$1

if [ -z "$VERSION" ]; then
    echo "Error: No rollback version specified!"
    exit 1
fi

echo "Rolling back to version: $VERSION"

# Stop the current application
echo "Stopping current application..."
systemctl stop myapp || docker stop myapp-container

# Deploy the previous stable version
echo "Deploying version $VERSION..."
./deploy-app.sh $VERSION

# Start the application
echo "Starting application..."
systemctl start myapp || docker start myapp-container

# Verify rollback success
HEALTH_CHECK=$(curl -s -o /dev/null -w "%{http_code}" http://app-url.com)

if [ "$HEALTH_CHECK" == "200" ]; then
    echo "Rollback successful! Version $VERSION is now running."
else
    echo "Rollback failed! Manual intervention required."
    exit 1
fi
```

- ✓ Stops faulty deployment & rolls back to last stable version.
- ✓ Works with both Systemd (`systemctl`) and Docker.
- ✓ Performs a health check after rollback.

✓ Scenario 3: Blue-Green Deployment

Scenario:

- Deploy new version (`Blue`).
- If successful, **switch traffic to it** while keeping the old version (`Green`) as a backup.
- If failure occurs, **switch traffic back to `Green`**.

Jenkinsfile Solution

```
groovy

pipeline {
    agent any
    environment {
        ACTIVE_VERSION = 'green'
    }
}
```

```

        NEW_VERSION = 'blue'
    }
    stages {
        stage('Deploy New Version') {
            steps {
                sh "./deploy-app.sh ${env.NEW_VERSION}"
            }
        }
        stage('Health Check') {
            steps {
                script {
                    def healthStatus = sh(script: 'curl -s -o /dev/null -w "%{http_code}" http://api', returnStatus: true)
                    if (healthStatus != '200') {
                        error "Health check failed!"
                    }
                }
            }
        }
        stage('Switch Traffic') {
            steps {
                sh "echo 'Switching traffic to ${env.NEW_VERSION}'"
                sh "./switch-traffic.sh ${env.NEW_VERSION}"
            }
        }
    }
    post {
        failure {
            echo "Deployment failed! Reverting traffic back to ${env.ACTIVE_VERSION}"
            sh "./switch-traffic.sh ${env.ACTIVE_VERSION}"
        }
    }
}

```

- ✓ Minimizes downtime by keeping a backup version (`Green`).
- ✓ Automatically reverts traffic in case of failure.

`switch-traffic.sh` Script

```

bash

#!/bin/bash

NEW_VERSION=$1

if [ "$NEW_VERSION" == "blue" ]; then
    echo "Switching traffic to Blue environment..."
    # Example: Update Nginx config or load balancer
    sudo ln -sf /etc/nginx/sites-available/blue /etc/nginx/sites-enabled/default
elif [ "$NEW_VERSION" == "green" ]; then
    echo "Switching traffic to Green environment..."
    sudo ln -sf /etc/nginx/sites-available/green /etc/nginx/sites-enabled/default
else
    echo "Invalid environment specified!"
    exit 1
fi

# Reload Nginx to apply changes
sudo systemctl reload nginx
echo "Traffic successfully switched to $NEW_VERSION"

```

- ✓ Handles traffic switch seamlessly.
- ✓ Uses Nginx but can be modified for AWS ALB, Kubernetes, etc.

✓ Scenario 4: Jenkins Pipeline with Slack Notifications

🚀 Scenario:

- Notify the **Slack team** when:
 - Build starts
 - Build succeeds
 - Build fails

Jenkinsfile Solution

```

groovy

pipeline {
    agent any
    environment {
        SLACK_CHANNEL = '#devops-alerts'
    }
    stages {
        stage('Build') {
            steps {
                slackSend(channel: env.SLACK_CHANNEL, message: "🚀 Build Started: ${env.BUILD_NUMBER}")
                sh 'mvn clean install'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Deploy') {
            steps {
                sh './deploy-app.sh'
            }
        }
    }
    post {
        success {
            slackSend(channel: env.SLACK_CHANNEL, message: "✅ Build Successful: ${env.BUILD_NUMBER}")
        }
        failure {
            slackSend(channel: env.SLACK_CHANNEL, message: "❌ Build Failed: ${env.BUILD_NUMBER}")
        }
    }
}

```

- ✅ Sends real-time Slack alerts.
- ✅ Enhances DevOps monitoring.

🚀 Summary

These advanced **real-world Jenkins pipelines** help you implement:

- ✅ Multi-Branch Builds
- ✅ Rolling Deployments & Rollbacks
- ✅ Blue-Green Deployments
- ✅ Slack Notifications

Would you like **custom scenarios based on your real project**? Let me know! 🚀🔥

ChatGPT can make mistakes. Check important info.