Cracking DevOps Interviews: A Conversational Journey

Complete Interview Preparation Guide
CI/CD • Jenkins • Maven • Kubernetes

📖 Table of Contents

PART 1: CI/CD Fundamentals & Jenkins

1. Introduction to CI/CD
2. Jenkins Fundamentals
3. Jenkins Jobs
4. Jenkins Pipelines

PART 2: Automation & Build Management 5. Groovy for Jenkins 6. CI Automation Workflows 7. Maven Fundamentals 8. Maven in CI/CD

PART 3: Container Orchestration 9. Kubernetes Basics 10. Kubernetes Pods 11. Deployments & Services

---

Chapter 1: Introduction to CI/CD

User: I have DevOps interviews coming up, and every job description mentions CI/CD. What problem does it actually solve?

Expert: Perfect starting question! Let me ask you—have you ever deployed code manually?

User: Yes, I'd SSH into a server, pull from Git, build, and restart the service. It worked but was tedious.

Expert: And what happened when you made a mistake?

User: Once I forgot to run tests. Deployed broken code. The service crashed.

Expert: That's the exact problem CI/CD solves! Now imagine Netflix with 2,000 engineers making 500 changes daily. Manual deployment would be chaos.

```
Problems CI/CD Solves:
═══════════════════════════════════════════════════════════

┌─────────────────────────────────────────────────────────┐
│ WITHOUT CI/CD (Manual)                                  │
├─────────────────────────────────────────────────────────┤
│ 1. Human Error                                          │
│    • Forgot to run tests                                │
│    • Deployed to wrong environment                      │
│                                                         │
│ 2. No Consistency                                       │
│    • Works on my machine ≠ production                   │
│                                                         │
│ 3. Slow Feedback                                        │
│    • Bugs found days/weeks later                        │
│                                                         │
│ 4. Doesn't Scale                                        │
│    • Bottleneck with multiple developers                │
└─────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────┐
│ WITH CI/CD (Automated)                                  │
├─────────────────────────────────────────────────────────┤
│ ✓ Automated testing on every commit                     │
│ ✓ Consistent builds across environments                 │
│ ✓ Fast feedback (minutes, not days)                     │
│ ✓ Scales to hundreds of daily deployments               │
└─────────────────────────────────────────────────────────┘
```

User: What's the difference between CI and CD?

Expert: Critical interview question! Let me break it down:

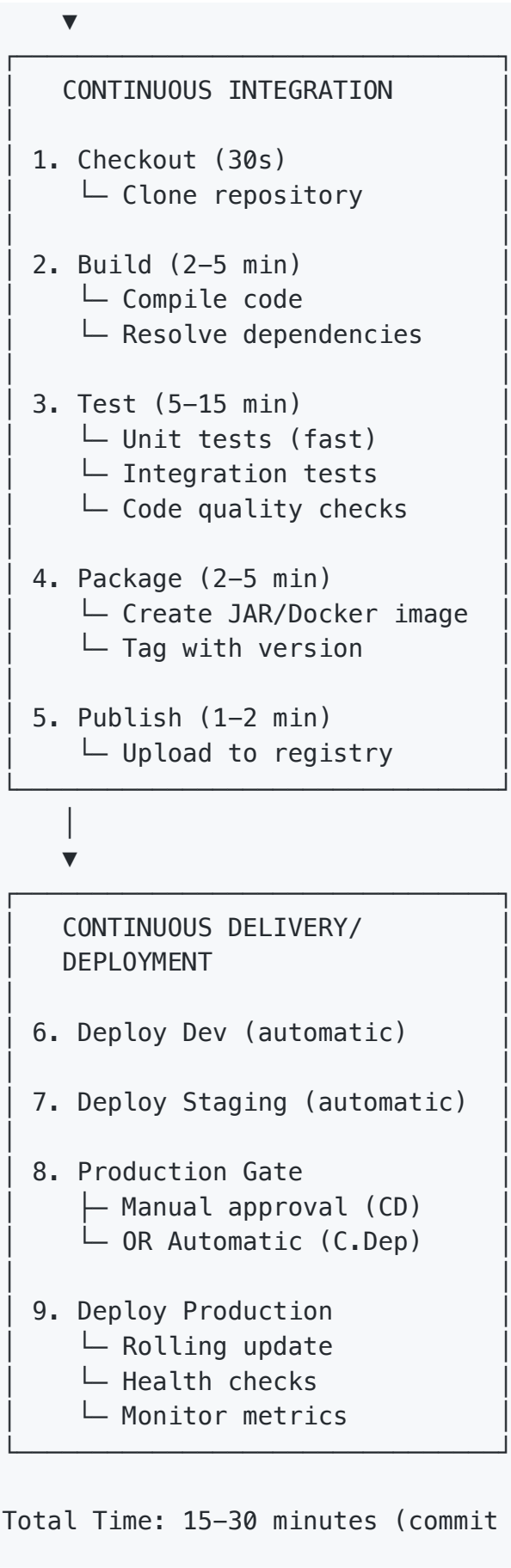
```
CI/CD Components:
═══════════════════════════════════════════════════════════

CONTINUOUS INTEGRATION (CI)
──────────────────────────
What: Automatically validate code on every commit
Flow: Commit → Build → Test → Package
Goal: Fast feedback (5-15 minutes)

Pseudo-code:
ON every_git_commit:
    checkout_code()
```

```
    compile_code()
    run_unit_tests()
    run_integration_tests()
    IF all_tests_pass:
        package_artifact()
        store_in_repository()
    ELSE:
        notify_developer("Build failed!")
        STOP

CONTINUOUS DELIVERY (CD)
─────────────────────────
What: Code always ready to deploy
Deployment: MANUAL approval for production

Pseudo-code:
IF ci_passed:
    deploy_to_staging()  // Automatic
    run_smoke_tests()
    WAIT_FOR human_approval  // Manual gate
    IF approved:
        deploy_to_production()

CONTINUOUS DEPLOYMENT
──────────────────────
What: Every successful build auto-deploys
Deployment: AUTOMATIC to production

Pseudo-code:
IF ci_passed:
    deploy_to_staging()
    IF staging_tests_pass:
        deploy_to_production()  // No human approval
        monitor_metrics()
        IF error_rate_high:
            automatic_rollback()
```

User: So Delivery has manual approval, Deployment is fully automatic?

Expert: Exactly! Here's the complete flow:

```
Complete CI/CD Pipeline Flow:
═══════════════════════════════════════════════════════

Developer
    │
    ├─ git commit
    ├─ git push
    │
```



User: Why is fast feedback important?

Expert: Cost! Finding bugs early is exponentially cheaper:

```
Bug Discovery Cost:
═══════════════════════════════════════════════════════════

Stage           Time to Fix    Cost    Example
─────────────────────────────────────────────────────────
During Coding   1 hour         $       IDE catches syntax error
CI Pipeline     2-4 hours      $$      Test fails immediately
QA Testing      1-2 days       $$$     Manual tester finds bug
Staging         2-5 days       $$$$    Pre-prod issue
Production      1-2 weeks      $$$$$   Customer impact
                                       + Revenue loss
                                       + Reputation damage

Interview Insight:
"CI catches bugs in hours vs weeks in production.
For a team of 20 devs, this saves hundreds of hours monthly."
```

User: How often do real companies deploy?

Expert: Here are real numbers you can cite:

```
Real-World Deployment Frequencies (2024):
═══════════════════════════════════════════════════════════

Company     Deploys/Day    MTTR        Strategy
─────────────────────────────────────────────────────────
Amazon      23,000         <60 min     Microservices
Netflix     4,000          <15 min     Chaos Engineering
Google      25,000         <5 min      Canary + Monitoring
Meta        2,000          <30 min     Feature Flags
Etsy        50             <15 min     Continuous Deployment

MTTR = Mean Time To Recovery

Interview Answer Template:
"The goal isn't maximum frequency. It's deploying as
often as business needs with minimal risk. CI/CD enables
high-frequency deployments through:
 • Small batch sizes
 • Automated testing
 • Fast rollback capability"
```

User: What should I know for interviews?

Expert: Three key areas:

```
Interview Framework - CI/CD:
```

```
═══════════════════════════════════════════════════════

LEVEL 1: Definitions
────────────────────
Q: "What's CI/CD?"
A: "CI automatically validates code changes. CD ensures
   code is always deployable (Delivery) or automatically
   deploys (Deployment)."

LEVEL 2: Practical Experience
─────────────────────────────
Q: "Describe a pipeline you built"
A: "I built a Jenkins pipeline with:
   • Maven for compilation and testing
   • Docker for containerization
   • Kubernetes for deployment
   • Optimized from 30 to 8 minutes via caching"

LEVEL 3: Design Thinking
────────────────────────
Q: "Design CI/CD for a 10-person team deploying weekly"
A: "I'd analyze current pain points first:
   • What causes deployment failures?
   • How long does manual deployment take?
   • Start with CI for fast feedback
   • Gradually move to automated staging deploys
   • Manual production approval initially
   • Measure MTTR and deployment frequency"

LEVEL 4: Production Knowledge
─────────────────────────────
Q: "How do you handle failed production deployments?"
A: "Implement progressive delivery:
   • Blue-green for instant rollback
   • Canary for gradual rollout (5% → 100%)
   • Automated health checks
   • Automatic rollback on metric degradation
   • Feature flags to disable without redeployment"
```

Chapter 1 Summary:

- CI/CD scales development velocity safely
- CI: Automated validation on every commit
- CD: Always deployable (manual) vs auto-deployed (automatic)
- Bug cost increases exponentially with detection delay
- Top companies deploy thousands of times daily
- Interview focus: definitions, experience, design thinking, production patterns

Chapter 2: Jenkins Fundamentals

User: Why Jenkins? There are newer tools like GitHub Actions.

Expert: Jenkins is ubiquitous. Most companies have it. Migrating thousands of pipelines is expensive. Understanding Jenkins is essential for interviews.

```
Jenkins Architecture:
═══════════════════════════════════════════════════════════════

          ┌──────────────────────────────┐
          │   JENKINS MASTER             │
          │   (Controller)               │
          │                              │
          │  • Schedules jobs            │
          │  • Manages build queue       │
          │  • Stores results            │
          │  • Serves Web UI             │
          └──────────────┬───────────────┘
                         │
          ┌──────────────┼──────────────┐
          │              │              │
          ▼              ▼              ▼
    ┌──────────┐   ┌──────────┐   ┌──────────┐
    │ AGENT1   │   │ AGENT2   │   │ AGENT3   │
    │ Linux    │   │ Windows  │   │ macOS    │
    │ Docker   │   │ .NET     │   │ Xcode    │
    └──────────┘   └──────────┘   └──────────┘
         │              │              │
         ▼              ▼              ▼
      Build          Build          Build
      Java           .NET           iOS

Master = Brain (coordinates)
Agents = Muscle (execute builds)

Interview Insight:
"Master-agent architecture enables scalability.
One master can manage hundreds of agents with
different capabilities (Linux, Windows, Docker)."
```

User: Jobs vs Pipelines—what's the difference?

Expert: Jobs are generic tasks. Pipelines are "Pipeline as Code":

```
Jobs vs Pipelines:
═══════════════════════════════════════════════════════════

FREESTYLE JOB (Old Way)
──────────────────────
Configuration: UI-based (click, click, click)
Storage: Jenkins database
Version Control: ✗ No
Code Review: ✗ No
Replication: ✗ Hard

Use Case: Simple, one-off tasks

PIPELINE JOB (Modern Way)
─────────────────────────
Configuration: Code (Jenkinsfile)
Storage: Git repository
Version Control: ✓ Yes
Code Review: ✓ Pull requests
Replication: ✓ Easy (copy file)

Use Case: All build/test/deploy workflows

Jenkinsfile Location:
my-app/
├── src/
├── pom.xml
└── Jenkinsfile  ← Lives with code
```

User: Show me a basic pipeline.

Expert: Here's declarative syntax (recommended):

```groovy
// Basic Jenkins Pipeline
pipeline {
    // Where to run
    agent any

    // Variables
    environment {
        APP_NAME = 'my-app'
        VERSION = '1.0.0'
    }

    // Build stages
    stages {
        stage('Build') {
            steps {
```

```
                echo "Building ${APP_NAME}"
                sh 'mvn clean compile'
            }
        }

        stage('Test') {
            steps {
                sh 'mvn test'
                junit 'target/surefire-reports/*.xml'
            }
        }

        stage('Package') {
            steps {
                sh 'mvn package -DskipTests'
                archiveArtifacts 'target/*.jar'
            }
        }
    }

    // Post-build actions
    post {
        success {
            echo 'Build succeeded!'
        }
        failure {
            echo 'Build failed!'
        }
    }
}

// Pseudo-code explanation:
DEFINE pipeline:
    SET agent = any
    SET environment variables

    FOR EACH stage IN [Build, Test, Package]:
        EXECUTE stage.steps
        IF stage fails:
            STOP pipeline
            EXECUTE post.failure

    IF all stages pass:
        EXECUTE post.success
```

User: What are agent options?

Expert: Critical for interviews:

```
Agent Configuration:
═══════════════════════════════════════════════════════════

// Option 1: Any available agent
agent any

// Option 2: Specific labeled agent
agent {
    label 'linux && docker'
}

// Option 3: Docker container (Best for consistency!)
agent {
    docker {
        image 'maven:3.8.1-jdk-11'
        args '-v /var/maven/.m2:/root/.m2'  // Cache!
    }
}

// Option 4: Different agents per stage
pipeline {
    agent none  // No default
    stages {
        stage('Build') {
            agent { label 'linux' }
            steps { sh 'mvn compile' }
        }
        stage('Deploy') {
            agent { label 'production' }  // Special access
            steps { sh './deploy.sh' }
        }
    }
}

Docker Agent Benefits:
├─ Guaranteed tool versions (Maven 3.8.1, JDK 11)
├─ Consistent across all builds
├─ No agent configuration needed
└─ Agent just needs Docker installed

Interview Tip:
"Docker agents ensure build reproducibility.
Same environment every time, no 'works on my machine' issues."
```

User: How do credentials work?

Expert: Never hardcode! Use Jenkins credentials store:

```
// WRONG - Never do this!
sh 'aws s3 cp file.txt --access-key AKIAEXAMPLE'

// RIGHT - Use credentials
pipeline {
    environment {
        // Jenkins auto-creates USER and PSW variables
        AWS_CREDS = credentials('aws-creds-id')
        DOCKER_CREDS = credentials('docker-hub-id')
    }

    stages {
        stage('Deploy') {
            steps {
                // Credentials auto-injected
                sh '''
                    aws s3 cp file.txt s3://bucket
                    # Uses $AWS_CREDS_USR and $AWS_CREDS_PSW
                '''
            }
        }

        stage('Docker Push') {
            steps {
                sh '''
                    echo $DOCKER_CREDS_PSW | \
                    docker login -u $DOCKER_CREDS_USR --password-stdin
                '''
            }
        }
    }
}

// Pseudo-code:
credentials('id') returns:
    - USERNAME as ${ID}_USR
    - PASSWORD as ${ID}_PSW
    - Masked in console output (shows ****)
    - Injected as environment variables

Security:
✓ Stored encrypted in Jenkins
✓ Masked in logs
✓ Access controlled by permissions
✗ Not visible in Jenkinsfile
```

Chapter 2 Summary:

- Jenkins: master-agent architecture
- Master coordinates, agents execute
- Pipelines (code) better than freestyle jobs (UI)
- Agent options: any, labeled, Docker (best for consistency)
- Credentials: use credentials() function, never hardcode
- Jenkinsfile lives in Git with application code

Chapter 3: Jenkins Jobs

User: Why learn freestyle jobs if pipelines are better?

Expert: Two reasons: legacy systems have thousands of them, and interviews test your knowledge of evolution.

```
Freestyle Job Structure:
═══════════════════════════════════════════════════════

1. SOURCE CODE MANAGEMENT
   ├─ Git repository URL
   ├─ Branch: */main
   └─ Credentials

2. BUILD TRIGGERS
   ├─ Poll SCM: H/5 * * * * (every 5 min)
   ├─ GitHub webhook (better!)
   └─ Build periodically: 0 2 * * * (2 AM daily)

3. BUILD ENVIRONMENT
   ├─ Delete workspace before build
   ├─ Inject credentials
   └─ Timeout: 60 minutes

4. BUILD STEPS
   ├─ Execute shell: mvn clean compile
   ├─ Execute shell: mvn test
   └─ Execute shell: mvn package

5. POST-BUILD ACTIONS
   ├─ Archive artifacts: target/*.jar
   ├─ Publish JUnit results
   └─ Email notification
```

User: What about build triggers?

Expert: Critical difference between polling and webhooks:

```
Build Triggers:
═══════════════════════════════════════════════════════

POLL SCM (Inefficient)
──────────────────────
Cron: H/5 * * * *  (every 5 minutes)

Pseudo-code:
EVERY 5 minutes:
    check_git_for_new_commits()
    IF new_commits_found:
        trigger_build()

Problem: Wastes resources checking when nothing changed

GITHUB WEBHOOK (Efficient)
──────────────────────────
Setup: GitHub → Settings → Webhooks
URL: http://jenkins.company.com/github-webhook/

Pseudo-code:
ON git_push_event:
    github_sends_notification_to_jenkins()
    jenkins_receives_webhook()
    jenkins_triggers_build_immediately()

Benefits:
✓ Instant (no delay)
✓ No wasted polling
✓ Scalable to thousands of repos

Interview Answer:
"Webhooks are superior to polling. They provide instant
feedback and don't waste resources. Polling is only used
when Jenkins is behind a firewall and can't receive webhooks."
```

User: What's parameterized build?

Expert: Accepts inputs when triggered:

```
// Parameterized Build Example
Parameters:
├─ String: ENVIRONMENT (default: staging)
├─ Choice: VERSION (1.0.0, 1.1.0, 1.2.0)
└─ Boolean: RUN_TESTS (default: true)
```

```
Build Script:
#!/bin/bash
echo "Deploying version $VERSION to $ENVIRONMENT"

if [ "$RUN_TESTS" = "true" ]; then
    mvn test
fi

./deploy.sh --env=$ENVIRONMENT --version=$VERSION

// When triggered, shows form:
┌─────────────────────────────────┐
|  Build with Parameters          |
├─────────────────────────────────┤
|  Environment: [staging     ▼]   |
|  Version:     [1.0.0       ▼]   |
|  Run Tests:   [✓] Yes            |
|                                 |
|  [Build]  [Cancel]              |
└─────────────────────────────────┘
```

Chapter 3 Summary:

- Freestyle jobs: UI-configured, hard to replicate
- Triggers: webhooks > polling (instant, efficient)
- Parameterized builds accept user inputs
- Migration strategy: gradual, not big-bang
- Modern practice: use pipelines for all new work

Chapter 4: Jenkins Pipelines

User: Let's dive deep into pipeline syntax.

Expert: Two types: Declarative (easier) and Scripted (flexible). Focus on Declarative:

```
// Complete Declarative Pipeline
pipeline {
    // 1. AGENT: Where to run
    agent {
        docker {
            image 'maven:3.8.1-jdk-11'
```

```
            args '-v maven-cache:/root/.m2'
        }
    }

    // 2. ENVIRONMENT: Variables
    environment {
        APP = 'myapp'
        VERSION = "${env.GIT_COMMIT.take(7)}"
        REGISTRY = 'registry.company.com'
    }

    // 3. STAGES: Workflow
    stages {
        stage('Build') {
            steps {
                sh 'mvn clean compile'
            }
        }

        stage('Test') {
            steps {
                sh 'mvn test'
                junit 'target/surefire-reports/*.xml'
            }
        }

        stage('Package') {
            steps {
                sh 'mvn package -DskipTests'
                archiveArtifacts 'target/*.jar'
            }
        }

        stage('Deploy to Staging') {
            when {
                branch 'main'  // Only on main branch
            }
            steps {
                sh './deploy-staging.sh'
            }
        }
    }

    // 4. POST: After pipeline
    post {
        success {
            slackSend color: 'good',
                      message: "${APP} ${VERSION} succeeded"
```

```
        }
        failure {
            slackSend color: 'danger',
                      message: "${APP} ${VERSION} failed"
        }
        always {
            cleanWs()  // Clean workspace
        }
    }
}
```

User: What's the `when` directive?

Expert: Conditional stage execution:

```
// When Directive Examples
═══════════════════════════════════════════════════

// 1. Branch-based
stage('Deploy Production') {
    when {
        branch 'main'
    }
    steps {
        sh './deploy-prod.sh'
    }
}

// 2. Environment variable
stage('Run Performance Tests') {
    when {
        environment name: 'RUN_PERF_TESTS', value: 'true'
    }
    steps {
        sh './perf-tests.sh'
    }
}

// 3. Expression (custom logic)
stage('Deploy on Even Builds') {
    when {
        expression {
            return env.BUILD_NUMBER.toInteger() % 2 == 0
        }
    }
    steps {
        sh './deploy.sh'
```

```
    }
}

// 4. Multiple conditions (AND)
stage('Production Deployment') {
    when {
        allOf {
            branch 'main'
            environment name: 'DEPLOY_PROD', value: 'true'
        }
    }
    steps {
        input message: 'Deploy to production?'
        sh './deploy-prod.sh'
    }
}

// 5. Multiple conditions (OR)
stage('Deploy to Non-Prod') {
    when {
        anyOf {
            branch 'develop'
            branch 'staging'
        }
    }
    steps {
        sh './deploy-staging.sh'
    }
}

// Pseudo-code:
FOR EACH stage IN pipeline:
    evaluate_when_conditions()
    IF conditions_met:
        execute_stage()
    ELSE:
        skip_stage()
```

User: What about manual approvals?

Expert: Use `input` step:

```
// Manual Approval Pattern
stage('Approve Production') {
    steps {
        timeout(time: 24, unit: 'HOURS') {
            input message: 'Deploy to production?',
```

```
                    ok: 'Deploy',
                    submitter: 'ops-team,tech-leads'
            }
        }
    }

    // Advanced: Release agent during approval
    stage('Wait for Approval') {
        agent none  // Don't hold agent
        steps {
            input message: 'Proceed?'
        }
    }
    stage('Deploy') {
        agent { label 'production-deployer' }
        steps {
            sh './deploy.sh'
        }
    }

    // Pseudo-code:
    display_approval_dialog_in_ui()
    WAIT FOR user_clicks_approve OR timeout_reached
    IF approved:
        continue_pipeline()
    ELSE IF timeout:
        abort_pipeline()
```

User: Show me a real production pipeline.

Expert: Here's a complete example with all concepts:

```
    // Production-Grade Pipeline
    pipeline {
        agent none

        environment {
            APP_NAME = 'user-service'
            DOCKER_REGISTRY = 'registry.company.com'
            VERSION = "${env.GIT_COMMIT.take(7)}"
        }

        stages {
            // 1. Build and test in Maven container
            stage('Build & Test') {
                agent {
                    docker {
```

```
                image 'maven:3.8.1-jdk-11'
                args '-v maven-cache:/root/.m2'
            }
        }
        steps {
            sh 'mvn clean test'
            junit 'target/surefire-reports/*.xml'
            sh 'mvn package -DskipTests'
            stash name: 'jar', includes: 'target/*.jar'
        }
    }

    // 2. Build Docker image
    stage('Docker Build') {
        agent { label 'docker' }
        steps {
            unstash 'jar'
            sh """
                docker build -t ${DOCKER_REGISTRY}/${APP_NAME}:${VE
                docker push ${DOCKER_REGISTRY}/${APP_NAME}:${VERSIO
            """
        }
    }

    // 3. Deploy to staging (automatic)
    stage('Deploy Staging') {
        agent { label 'kubectl' }
        when { branch 'main' }
        steps {
            sh """
                kubectl set image deployment/${APP_NAME} \\
                    ${APP_NAME}=${DOCKER_REGISTRY}/${APP_NAME}:${VE
                    -n staging
                kubectl rollout status deployment/${APP_NAME} -n st
            """
        }
    }

    // 4. Manual approval for production
    stage('Approve Production') {
        agent none
        when { branch 'main' }
        steps {
            timeout(time: 24, unit: 'HOURS') {
                input message: 'Deploy to production?'
            }
        }
    }
```

```
        // 5. Deploy to production
        stage('Deploy Production') {
            agent { label 'kubectl' }
            when { branch 'main' }
            steps {
                sh """
                    kubectl set image deployment/${APP_NAME} \\
                        ${APP_NAME}=${DOCKER_REGISTRY}/${APP_NAME}:${VE
                        -n production
                    kubectl rollout status deployment/${APP_NAME} -n pr
                """
            }
        }
    }

    post {
        success {
            slackSend channel: '#deployments',
                      color: 'good',
                      message: "✅ ${APP_NAME} ${VERSION} deployed"
        }
        failure {
            slackSend channel: '#deployments',
                      color: 'danger',
                      message: "❌ ${APP_NAME} ${VERSION} failed"
        }
    }
}

// Flow visualization:
Code Commit
    ↓
Build & Test (Maven container)
    ↓
Docker Build (Docker agent)
    ↓
Deploy Staging (kubectl agent) [if main branch]
    ↓
Manual Approval [wait for human]
    ↓
Deploy Production (kubectl agent)
    ↓
Slack Notification
```

Chapter 4 Summary:

- Declarative pipelines: structured, recommended

- Agent can be: any, labeled, Docker, or per-stage
- `when` directive: conditional stage execution
- `input` step: manual approvals
- `stash` / `unstash` : transfer files between stages
- Production pattern: different agents for different tasks

Chapter 5: Groovy for Jenkins

User: When do I need Groovy in pipelines?

Expert: For logic that declarative syntax can't handle—loops, conditionals, complex decisions:

```
// Groovy in Pipelines
pipeline {
    agent any
    stages {
        stage('Groovy Logic') {
            steps {
                script {  // Enter Groovy land
                    // Variables
                    def version = '1.0.0'
                    def environments = ['dev', 'staging', 'prod']

                    // Conditionals
                    if (env.BRANCH_NAME == 'main') {
                        echo "Production branch detected"
                    }

                    // Loops
                    for (env in environments) {
                        echo "Environment: ${env}"
                    }

                    // File operations
                    def config = readJSON file: 'config.json'
                    echo "App version: ${config.version}"

                    // Error handling
                    try {
                        sh './risky-script.sh'
                    } catch (Exception e) {
                        echo "Failed: ${e.message}"
```

```
                        currentBuild.result = 'UNSTABLE'
                    }
                }
            }
        }
    }
}

// Groovy Basics:
═══════════════════════════════════════════════════════

// Variables (dynamic typing)
def name = 'Jenkins'
def count = 42
def isReady = true

// String interpolation (double quotes only!)
def msg1 = 'Hello ${name}'      // Literal: "Hello ${name}"
def msg2 = "Hello ${name}"      // Interpolated: "Hello Jenkins"

// Lists
def numbers = [1, 2, 3, 4, 5]
numbers << 6                    // Append
numbers.each { echo it }        // Iterate

// Maps
def person = [
    name: 'Alice',
    role: 'DevOps'
]
echo person.name                // "Alice"

// Functions
def greet(name) {
    return "Hello, ${name}!"
}
echo greet('World')             // "Hello, World!"
```

User: Show me practical examples.

Expert: Here are real interview scenarios:

```
// Pattern 1: Dynamic Deployment to Multiple Regions
script {
    def regions = ['us-east-1', 'us-west-2', 'eu-central-1']

    for (region in regions) {
```

```
        stage("Deploy to ${region}") {
            echo "Deploying to ${region}"
            sh "kubectl apply -f deploy.yaml --context=${region}"
        }
    }
}

// Pattern 2: Conditional Execution Based on File
script {
    if (fileExists('Dockerfile')) {
        echo 'Building Docker image'
        sh 'docker build -t myapp .'
    } else if (fileExists('pom.xml')) {
        echo 'Maven project detected'
        sh 'mvn clean package'
    } else {
        error 'Unknown project type'
    }
}

// Pattern 3: Reading Config and Dynamic Pipeline
script {
    // config.yaml contains list of services to deploy
    def config = readYaml file: 'config.yaml'

    for (service in config.services) {
        stage("Deploy ${service.name}") {
            sh """
                kubectl set image deployment/${service.name} \\
                    app=${service.image}:${VERSION} \\
                    --namespace=${service.namespace}
            """
        }
    }
}

// Pattern 4: Error Handling with Retry
script {
    def maxRetries = 3
    def attempt = 0
    def success = false

    while (attempt < maxRetries && !success) {
        try {
            attempt++
            echo "Attempt ${attempt} of ${maxRetries}"
            sh './flaky-api-call.sh'
            success = true
```

```
            } catch (Exception e) {
                if (attempt < maxRetries) {
                    sleep time: 5, unit: 'SECONDS'
                } else {
                    throw e
                }
            }
        }
    }

    // Pattern 5: Parallel Execution
    script {
        def deployments = [:]
        def regions = ['us-east', 'us-west', 'eu-central']

        for (region in regions) {
            def r = region  // Important: capture variable
            deployments[r] = {
                echo "Deploying to ${r}"
                sh "kubectl apply -f deploy.yaml --context=${r}"
            }
        }

        parallel deployments  // All run simultaneously!
    }

    // Pseudo-code for parallel:
    CREATE empty map deployments
    FOR EACH region:
        CREATE closure for region
        ADD closure to deployments map
    EXECUTE all closures in parallel
    WAIT FOR all to complete
```

Chapter 5 Summary:

- Use `script` blocks for Groovy logic in declarative pipelines
- File operations: readJSON, readYaml, fileExists
- Error handling: try-catch, retry patterns
- Parallel execution: create map of closures, use `parallel` step
- Variable capture in loops: use local copy (`def r = region`)

Chapter 6: CI Automation Workflows

User: How do I structure a complete CI/CD workflow?

Expert: Build → Test → Package → Deploy, but with nuance for different branches:

```
Branch-Based Workflow:
═══════════════════════════════════════════════════════

Feature Branch (feature/*)
    ├─ Build & Unit Test (always)
    ├─ Deploy to Dev environment (auto)
    └─ Create preview environment

Develop Branch
    ├─ Build & Test (always)
    ├─ Integration Tests
    ├─ Code Quality Analysis
    └─ Deploy to Staging (auto)

Main Branch (production)
    ├─ Build & Test (always)
    ├─ Full Test Suite
    ├─ Security Scans
    ├─ Deploy to Staging (auto)
    ├─ Manual Approval ⏸
    └─ Deploy to Production (manual)

Pull Request
    ├─ Build & Test (always)
    ├─ Code Coverage Check
    ├─ Security Scan
    ├─ Post results to PR
    └─ Create preview environment

Pseudo-code:
ON git_event:
    determine_branch_type()

    // Always run
    checkout_code()
    build_application()
    run_unit_tests()
    package_artifact()

    IF branch == "main":
        run_full_test_suite()
        deploy_to_staging()
        WAIT_FOR manual_approval
        deploy_to_production()
```

```
    ELSE IF branch == "develop":
        run_integration_tests()
        deploy_to_staging()

    ELSE IF is_pull_request:
        calculate_code_coverage()
        post_results_to_pr()
        create_preview_environment()

    ELSE:  // Feature branch
        deploy_to_dev_environment()
```

User: What about artifact management?

Expert: Version properly and store in repositories:

```
Artifact Versioning Strategy:
═══════════════════════════════════════════════════

// Version from Git
VERSION = git_commit_hash (e.g., "a3f5c2b")
         OR
         git_tag (e.g., "v1.2.3")
         OR
         timestamp + hash (e.g., "20240115-a3f5c2b")

// Docker Tagging Strategy
docker build -t registry.company.com/myapp:VERSION .
docker tag registry.company.com/myapp:VERSION \\
           registry.company.com/myapp:BRANCH-latest

// Multiple tags serve different purposes:
registry.company.com/myapp:a3f5c2b        // Specific version
registry.company.com/myapp:main-latest    // Latest from main
registry.company.com/myapp:v1.2.3         // Semantic version

Production Deployment:
└─ ALWAYS use specific version tag
   kubectl set image deployment/app app=myapp:a3f5c2b ✓

Development:
└─ Can use branch-latest
   docker pull myapp:develop-latest ✓

NEVER in Production:
└─ Never use "latest" tag
   kubectl set image deployment/app app=myapp:latest ✗
   (Which version is this? Can't rollback!)
```

```
Artifact Storage:
├─ JARs: Artifactory, Nexus
├─ Docker Images: Docker Registry, ECR, GCR
└─ General: S3, Azure Blob Storage
```

User: What about rollback strategies?

Expert: Four main patterns:

```
Rollback Strategies:
═══════════════════════════════════════════════════════

1. BLUE-GREEN DEPLOYMENT
─────────────────────────
Two identical environments. Switch traffic instantly.

Pseudo-code:
current_env = get_active_environment()  // "blue"
target_env = toggle(current_env)        // "green"

deploy_to(target_env, new_version)
run_smoke_tests(target_env)

IF tests_pass:
    switch_traffic_to(target_env)
    // Old environment (blue) ready for instant rollback
ELSE:
    rollback()  // Keep traffic on blue

Pros: Instant rollback, zero downtime
Cons: 2x infrastructure cost

2. CANARY DEPLOYMENT
────────────────────
Deploy to small subset, monitor, gradually increase.

Pseudo-code:
deploy_to_percentage(5%)  // 5% of servers
monitor_metrics(duration=5min)

IF error_rate < threshold:
    deploy_to_percentage(10%)
    monitor_metrics(duration=5min)

    IF still_healthy:
        deploy_to_percentage(50%)
        deploy_to_percentage(100%)
ELSE:
    rollback_canary()
```

```
Pros: Limits blast radius, real user testing
Cons: Requires good monitoring, longer deployment

3. ROLLING UPDATE
─────────────────
Update servers one by one.

Pseudo-code:
FOR EACH server IN servers:
    deploy_new_version(server)
    wait_for_health_check()

    IF health_check_fails:
        rollback_all()
        STOP

    remove_old_version(server)

Pros: No extra infrastructure, gradual
Cons: Multiple versions running simultaneously

4. FEATURE FLAGS
────────────────
Deploy code with features disabled. Enable gradually.

Pseudo-code:
deploy_new_code_to_production()
feature_flag_set("new_feature", enabled=False)

// Later, gradually enable
enable_feature_for_percentage(1%)
monitor_metrics()

IF metrics_good:
    enable_feature_for_percentage(10%)
    enable_feature_for_percentage(100%)
ELSE:
    disable_feature()  // No redeployment needed!

Pros: Instant disable, no redeployment, A/B testing
Cons: Code complexity, technical debt

Interview Decision Matrix:
──────────────────────────
Q: "Which rollback strategy should we use?"
A: "Depends on constraints:
   • Blue-green: If you have spare capacity and need instant rollback
   • Canary: If you want gradual rollout with monitoring
   • Rolling: If you have limited resources
   • Feature flags: If you want to decouple deployment from release"
```

Chapter 6 Summary:

- Branch-based workflows: different strategies per branch
- Artifact versioning: use Git hash, never "latest" in production
- Rollback strategies: blue-green, canary, rolling, feature flags
- Choose based on constraints: infrastructure, risk tolerance, monitoring

---

Chapter 7: Maven Fundamentals

User: What is Maven and why do I need it?

Expert: Maven manages dependencies, builds, and packaging for Java projects. Alternative is managing manually—impossible at scale.

```
Maven Solves:
═══════════════════════════════════════════════════════════

Problem: 50+ dependencies, each with their own dependencies
Solution: Declare in pom.xml, Maven downloads automatically

pom.xml (Project Object Model):
──────────────────────────────────

<?xml version="1.0"?>
<project>
    <!-- Project Identity -->
    <groupId>com.company</groupId>
    <artifactId>user-service</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <!-- Dependencies -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <version>2.7.0</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
```

```
            <scope>test</scope>  <!-- Only for testing -->
        </dependency>
    </dependencies>
</project>

Maven Coordinates (Identity):
─────────────────────────────
groupId:artifactId:version

Example: org.springframework.boot:spring-boot-starter-web:2.7.0

Interview Tip:
"Maven coordinates uniquely identify artifacts in repositories,
similar to how GPS coordinates identify locations."
```

User: What's the Maven lifecycle?

Expert: Predefined phases executed in order:

```
Maven Lifecycle:
═══════════════════════════════════════════════════════════

clean      → Delete target/ directory
validate   → Validate project structure
compile    → Compile source code (src/main/java → target/classes)
test       → Run unit tests (src/test/java)
package    → Create JAR/WAR (target/myapp-1.0.0.jar)
verify     → Run integration tests
install    → Install to local repo (~/.m2/repository)
deploy     → Upload to remote repository

Key Insight: Running a phase executes all previous phases!

mvn package
    ├─ validate
    ├─ compile
    ├─ test
    └─ package

mvn clean package
    ├─ clean
    ├─ validate
    ├─ compile
    ├─ test
    └─ package

Common Commands:
────────────────
mvn clean test              // Fresh build, run tests
```

```
mvn clean package          // Build JAR
mvn clean install          // Build and install to local repo
mvn package -DskipTests    // Build without testing

Interview Question:
Q: "What's the difference between package and install?"
A: "Package creates the JAR in target/. Install additionally
   copies it to ~/.m2/repository, making it available to other
   local projects that depend on it."
```

User: How do dependencies work?

Expert: Maven downloads from repositories and handles transitive dependencies:

```
Dependency Resolution:
═══════════════════════════════════════════════════════

You declare:
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.7.0</version>
</dependency>

Maven downloads spring-boot-starter-web PLUS all its dependencies:
    spring-boot-starter-web 2.7.0
    ├─ spring-boot-starter 2.7.0
    │  ├─ spring-boot 2.7.0
    │  ├─ spring-core 5.3.20
    │  └─ ...
    ├─ spring-web 5.3.20
    ├─ spring-webmvc 5.3.20
    └─ tomcat-embed-core 9.0.63
       └─ ...

Total: ~30 JARs from 1 dependency!

Repository Search Order:
1. Local repository (~/.m2/repository)
2. Remote repositories (Maven Central, company Artifactory)

Pseudo-code:
FOR EACH dependency IN pom.xml:
    IF exists_in_local_repo:
        USE local_version
    ELSE:
        download_from_remote_repo()
        save_to_local_repo()
```

```
    FOR EACH transitive_dependency:
        REPEAT process

Commands:
mvn dependency:tree      // Show all dependencies
mvn dependency:analyze   // Find unused/missing dependencies
```

User: What about version conflicts?

Expert: Maven uses "nearest definition" wins:

```
Dependency Conflict Resolution:
═══════════════════════════════════════════════

Scenario:
Your Project
├─ Library A → commons-lang:3.0 (depth: 2)
└─ Library B → commons-lang:2.6 (depth: 2)

Both at same depth → Maven picks first declared in pom.xml

Override Solution:
<dependencies>
    <!-- Explicit declaration wins (depth: 1) -->
    <dependency>
        <groupId>commons-lang</groupId>
        <artifactId>commons-lang</artifactId>
        <version>3.12.0</version>
    </dependency>

    <dependency>
        <groupId>library-a</groupId>
        <artifactId>library-a</artifactId>
        <version>1.0</version>
    </dependency>
</dependencies>

Exclusion Solution:
<dependency>
    <groupId>library-a</groupId>
    <artifactId>library-a</artifactId>
    <version>1.0</version>
    <exclusions>
        <exclusion>
            <groupId>commons-lang</groupId>
            <artifactId>commons-lang</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

```
Interview Tip:
"For multi-module projects, use <dependencyManagement> in
parent POM to centralize version control."
```

User: What are dependency scopes?

Expert: Control when dependencies are available:

```
Dependency Scopes:
═══════════════════════════════════════════════════

COMPILE (default)
├─ Available: Compile, Runtime, Test
├─ Included in JAR: Yes
└─ Example: spring-core, gson

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <scope>compile</scope>  <!-- or omit -->
</dependency>

TEST
├─ Available: Test only
├─ Included in JAR: No
└─ Example: junit, mockito

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <scope>test</scope>
</dependency>

PROVIDED
├─ Available: Compile, Test
├─ Included in JAR: No (server provides it)
└─ Example: servlet-api, lombok

<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
</dependency>

RUNTIME
├─ Available: Runtime, Test
├─ Included in JAR: Yes
└─ Example: JDBC drivers
```

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>

Interview Decision Matrix:
Q: "Which scope for a JDBC driver?"
A: "Runtime - not needed at compile time, but required at
   runtime and must be included in the JAR."
```

Chapter 7 Summary:

- Maven: dependency management, build automation
- pom.xml: project configuration, dependencies
- Lifecycle: compile → test → package → install → deploy
- Dependencies: automatic download, transitive resolution
- Scopes: compile, test, provided, runtime
- Conflicts: nearest definition wins, can override explicitly

---

Chapter 8: Maven in CI/CD

User: How do I optimize Maven in Jenkins?

Expert: Caching is critical:

```
// Maven Optimization in Jenkins
pipeline {
    agent {
        docker {
            image 'maven:3.8.1-jdk-11'
            // CRITICAL: Cache dependencies
            args '-v maven-cache:/root/.m2'
        }
    }

    stages {
        stage('Build') {
            steps {
                sh '''
                    mvn clean package \\
```

```
                        -T 4 \\                  // 4 parallel threads
                        -DskipTests \\           // Skip tests (run sep
                        --batch-mode \\          // Non-interactive
                        --offline                // Use cache only
                '''
            }
        }
    }
}

Performance Impact:
___________________
Without caching:  15 minutes (downloads 200 MB each time)
With caching:     3 minutes  (80% faster!)
With -T 4:        2 minutes  (parallelization)
With --offline:   1.5 minutes (no network checks)

Interview Insight:
"Caching Maven dependencies saves ~500 minutes daily for a
team with 100 builds. That's 8+ hours of compute time."
```

User: What about testing strategy?

Expert: Run tests explicitly, skip when packaging:

```groovy
// Testing Strategy
stages {
    stage('Unit Tests') {
        steps {
            sh 'mvn test'
            junit 'target/surefire-reports/*.xml'
        }
    }

    stage('Integration Tests') {
        steps {
            sh 'mvn verify -DskipUnitTests'
            junit 'target/failsafe-reports/*.xml'
        }
    }

    stage('Package') {
        steps {
            // Tests already ran, skip them
            sh 'mvn package -DskipTests'
        }
    }
```

```
}

Why -DskipTests?
─────────────────
mvn package  // Runs tests again (waste!)
mvn package -DskipTests  // Just packages (efficient!)

Interview Tip:
"-DskipTests skips test execution but compiles test code.
-Dmaven.test.skip=true skips both compilation and execution.
Use -DskipTests when tests already ran in previous stage."
```

User: How about multi-module projects?

Expert: Maven handles build order automatically:

```
Multi-Module Project:
═══════════════════════════════════════════════════

project-root/
├── pom.xml (parent)
├── module-common/
│   └── pom.xml
├── module-api/
│   └── pom.xml (depends on common)
└── module-web/
    └── pom.xml (depends on api)

Parent pom.xml:
<modules>
    <module>module-common</module>
    <module>module-api</module>
    <module>module-web</module>
</modules>

Build Order (automatic):
1. module-common (no dependencies)
2. module-api (depends on common)
3. module-web (depends on api)

Commands:
mvn clean install                    // Build all modules
mvn install -pl module-web -am       // Build web + dependencies
  -pl = project list
  -am = also make (build dependencies)

mvn install -pl module-web           // Build only web (assumes deps bu

Pseudo-code:
```

```
analyze_dependencies()
create_build_graph()
FOR EACH module IN topological_sort(graph):
    build_module(module)
```

User: Common Maven pitfalls?

Expert: Here are the top issues:

```
Maven CI/CD Pitfalls:
═══════════════════════════════════════════════════════

1. SNAPSHOT in Production ✗
   <version>1.0.0-SNAPSHOT</version>
   Problem: Mutable, non-reproducible builds
   Solution: Use release versions in production

2. No Caching ✗
   Problem: Downloads 200 MB every build
   Solution: Mount Maven cache volume

3. Running Tests Twice ✗
   stage('Test') { sh 'mvn test' }
   stage('Package') { sh 'mvn package' }  // Runs tests again!
   Solution: sh 'mvn package -DskipTests'

4. Single-Threaded Builds ✗
   mvn clean package
   Solution: mvn clean package -T 4

5. No Build Timeout ✗
   Problem: Hangs forever
   Solution: timeout(time: 30, unit: 'MINUTES') { }

6. Ignoring Test Failures ✗
   sh 'mvn test || true'
   Solution: Let it fail naturally, capture results with junit

Interview Answer Template:
"Common Maven issues in CI/CD are:
 • SNAPSHOT dependencies (non-reproducible)
 • Missing dependency caching (slow builds)
 • Running tests multiple times (inefficient)
 • Single-threaded builds (underutilized resources)
I address these with proper versioning, Docker volume caching,
-DskipTests flag, and -T flag for parallelization."
```

Chapter 8 Summary:

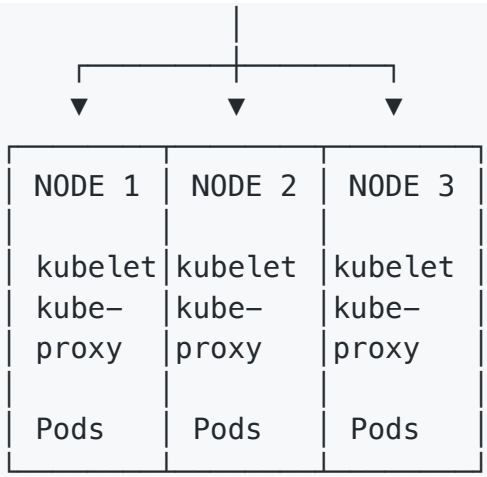
- Cache dependencies with Docker volumes (80% faster!)
- Parallel builds with `-T` flag
- Run tests once, use `-DskipTests` when packaging
- Multi-module: Maven handles build order automatically
- Avoid: SNAPSHOT in prod, no caching, running tests twice

Chapter 9: Kubernetes Basics

User: What is Kubernetes and why do we need it?

Expert: Kubernetes orchestrates containers at scale. Try managing 100 microservices on 50 servers manually—impossible!

```
Kubernetes Solves:
═══════════════════════════════════════════════════════════

Manual Docker Problems:
├─ Deployment: SSH to 50 servers, run docker commands
├─ Scaling: Manually start 20 instances, distribute manually
├─ Failures: Don't know until customers complain
├─ Load Balancing: Manually configure NGINX
└─ Updates: Manually update each container, downtime

Kubernetes Solutions:
├─ Deployment: kubectl apply -f deployment.yaml (seconds)
├─ Scaling: kubectl scale --replicas=20 (automatic distribution)
├─ Failures: Automatic detection and restart
├─ Load Balancing: Built-in, automatic
└─ Updates: Rolling updates, zero downtime

Architecture:
═══════════════════════════════════════════════════════════


┌──────────────────────────────────────────┐
│      CONTROL PLANE (Master)              │
│                                          │
│  API Server    Scheduler                 │
│  • REST API    • Pod placement           │
│  • kubectl     • Resource-based          │
│                                          │
│  Controller    etcd                      │
│  • Desired     • Cluster state           │
│    state       • Key-value store         │
└──────────────┬───────────────────────────┘
```

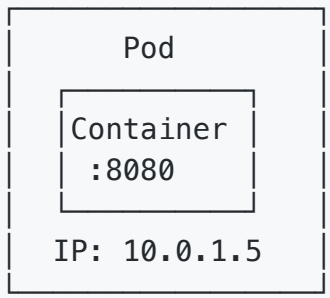



```
Interview Explanation:
"Control plane is the brain—makes decisions (scheduling,
scaling, healing). Worker nodes are the muscle—run the
actual containers in pods."
```

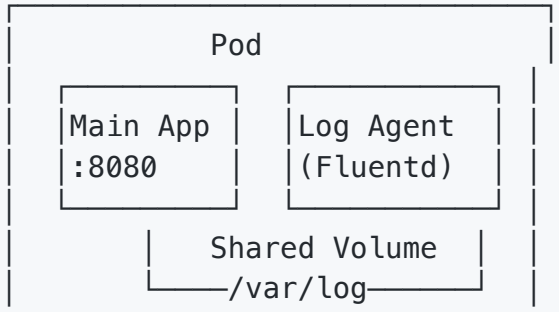
User: What's a pod?

Expert: Smallest deployable unit in Kubernetes:

```
Pod Concept:
═══════════════════════════════════════════════════════

Pod = Wrapper around container(s)
     + Shared network namespace
     + Shared storage
     + Same host

Single-Container Pod (90% of cases):
```




```
Multi-Container Pod (Sidecar Pattern):
```



```
│  IP: 10.0.1.5 (both share)  │
└─────────────────────────────┘

Pod YAML:
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
  - name: app
    image: registry.company.com/myapp:1.0.0
    ports:
    - containerPort: 8080
    resources:
      requests:
        memory: "256Mi"
        cpu: "250m"
      limits:
        memory: "512Mi"
        cpu: "500m"

kubectl commands:
kubectl get pods                    // List pods
kubectl describe pod myapp-pod      // Details
kubectl logs myapp-pod              // View logs
kubectl exec -it myapp-pod -- bash  // Shell into pod
```

User: What's the difference between Pod and Deployment?

Expert: Pods are instances, Deployments manage replicas:

```
Pod vs Deployment:
═══════════════════════════════════════════════════

POD (Ephemeral)
├─ Single instance
├─ No self-healing
├─ If deleted → gone forever
└─ Use: Almost never directly

DEPLOYMENT (Managed)
├─ Manages multiple pod replicas
├─ Self-healing (recreates failed pods)
├─ Scaling capability
├─ Rolling updates
├─ Rollback capability
└─ Use: Always in production
```

```
Visualization:
┌─────────────────────────────────┐
│   Deployment (desired: 3)       │
│   ┌──────┐  ┌──────┐  ┌──────┐  │
│   │Pod 1 │  │Pod 2 │  │Pod 3 │  │
│   └──────┘  └──────┘  └──────┘  │
└─────────────────────────────────┘
        ▲
        │ If Pod 2 crashes
        │
        ▼
┌─────────────────────────────────┐
│   Deployment (desired: 3)       │
│   ┌──────┐  ┌──────┐  ┌──────┐  │
│   │Pod 1 │  │Pod 4 │  │Pod 3 │  │
│   └──────┘  └──────┘  └──────┘  │
│              (auto-created)     │
└─────────────────────────────────┘

Deployment YAML:
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: app
        image: myapp:1.0.0
        ports:
        - containerPort: 8080

Commands:
kubectl create deployment myapp --image=myapp:1.0.0 --replicas=3
kubectl scale deployment myapp --replicas=10
kubectl set image deployment/myapp myapp=myapp:2.0.0
kubectl rollout status deployment/myapp
kubectl rollout undo deployment/myapp
```

User: What are namespaces?

Expert: Virtual clusters for isolation:

```
Namespaces:
═══════════════════════════════════════════════════════

Purpose: Environment/team isolation in same cluster

Default Namespaces:
├─ default: Resources without namespace
├─ kube-system: Kubernetes system components
├─ kube-public: Public resources
└─ kube-node-lease: Node heartbeats

Custom Namespaces (typical):
├─ dev: Development environment
├─ staging: Pre-production
└─ production: Live traffic

Commands:
kubectl create namespace staging
kubectl get pods -n staging
kubectl apply -f deployment.yaml -n production
kubectl get pods --all-namespaces

DNS in Namespaces:
<service>.<namespace>.svc.cluster.local

Examples:
myapp.production.svc.cluster.local
myapp.staging.svc.cluster.local

Short form (within same namespace):
myapp

Interview Insight:
"Namespaces enable running dev, staging, and production
in the same cluster with resource quotas and access
control per namespace."
```

Chapter 9 Summary:

- Kubernetes: container orchestration at scale
- Control plane: manages cluster (API, scheduler, controller, etcd)
- Worker nodes: run containers (kubelet, kube-proxy)
- Pods: smallest unit (usually 1 container)
- Deployments: manage pod replicas with self-healing
- Namespaces: virtual clusters for isolation

Chapter 10: Kubernetes Pods

User: Let's go deeper on pods. What's the lifecycle?

Expert: Pods go through distinct phases:

```
Pod Lifecycle:
═══════════════════════════════════════════════════════════

PENDING → RUNNING → SUCCEEDED/FAILED

PENDING
├─ Waiting for node assignment
├─ Pulling images
└─ Creating containers

RUNNING
├─ At least one container running
└─ Or starting/restarting

SUCCEEDED (for Jobs)
└─ All containers exited with 0

FAILED
└─ At least one container failed

Restart Policies:
─────────────────

ALWAYS (default)
└─ For: Long-running services (web servers, APIs)
    Restart regardless of exit status

ONFAILURE
└─ For: Jobs (data processing, batch)
    Restart only if failed (non-zero exit)

NEVER
└─ For: One-time tasks (migrations, init)
    Never restart

Pod Status Examples:
kubectl get pods
NAME                    STATUS              RESTARTS
myapp-abc123            Running             0
myapp-def456            CrashLoopBackOff    5
myapp-ghi789            ImagePullBackOff    0
```

```
Debug:
kubectl describe pod myapp-abc123  // Check Events section
kubectl logs myapp-abc123
kubectl logs myapp-abc123 --previous  // Before restart
```

User: How do I configure pods?

Expert: Three methods: environment variables, ConfigMaps, Secrets:

```
Pod Configuration:
═══════════════════════════════════════════════════════

Method 1: Environment Variables (Simple)
─────────────────────────────────────────
spec:
  containers:
  - name: app
    env:
    - name: LOG_LEVEL
      value: "info"
    - name: DATABASE_URL
      value: "postgres://db:5432"

Method 2: ConfigMap (Reusable)
──────────────────────────────
# Create ConfigMap
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  log_level: "info"
  database_url: "postgres://db:5432"

# Use in Pod
spec:
  containers:
  - name: app
    envFrom:
    - configMapRef:
        name: app-config

Method 3: Secret (Sensitive)
────────────────────────────
# Create Secret
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
```

```
type: Opaque
data:
  username: YWRtaW4=      # base64 encoded
  password: cGFzczEyMzQ=  # base64 encoded

# Use in Pod
spec:
  containers:
  - name: app
    env:
    - name: DB_USER
      valueFrom:
        secretKeyRef:
          name: db-secret
          key: username

ConfigMap vs Secret:
────────────────────
ConfigMap: Non-sensitive (URLs, config)
Secret: Sensitive (passwords, tokens)

⚠️ Important: Secrets are base64, NOT encrypted!
Use external secret managers in production:
- AWS Secrets Manager
- HashiCorp Vault
- Azure Key Vault
```

User: What about health checks?

Expert: Critical for production:

```
Health Checks (Probes):
═══════════════════════════════════════════════════════

LIVENESS PROBE
──────────────
Question: "Is container alive?"
Action if fails: Restart container

livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 30
  periodSeconds: 10
  failureThreshold: 3

Pseudo-code:
WAIT initialDelaySeconds
```

```
EVERY periodSeconds:
    check_health_endpoint()
    IF fails >= failureThreshold:
        restart_container()

READINESS PROBE
_______________
Question: "Ready to serve traffic?"
Action if fails: Remove from service

readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 5
  failureThreshold: 3

Pseudo-code:
WAIT initialDelaySeconds
EVERY periodSeconds:
    check_readiness_endpoint()
    IF fails >= failureThreshold:
        remove_from_load_balancer()
    ELSE IF passes_after_failure:
        add_back_to_load_balancer()

STARTUP PROBE (for slow apps)
_____________________________
startupProbe:
  httpGet:
    path: /startup
    port: 8080
  periodSeconds: 10
  failureThreshold: 30  // 30 * 10s = 5 min max startup

Interview Best Practices:
_________________________
1. ALWAYS use readiness probes
   → Prevents traffic to non-ready pods

2. Usually use liveness probes
   → Restarts stuck containers

3. Different endpoints:
   /health (liveness) - "Am I alive?"
   /ready (readiness) - "Can I serve traffic?"

4. Lightweight checks
   → Don't query database in /health
   → Just check if app is responsive
```

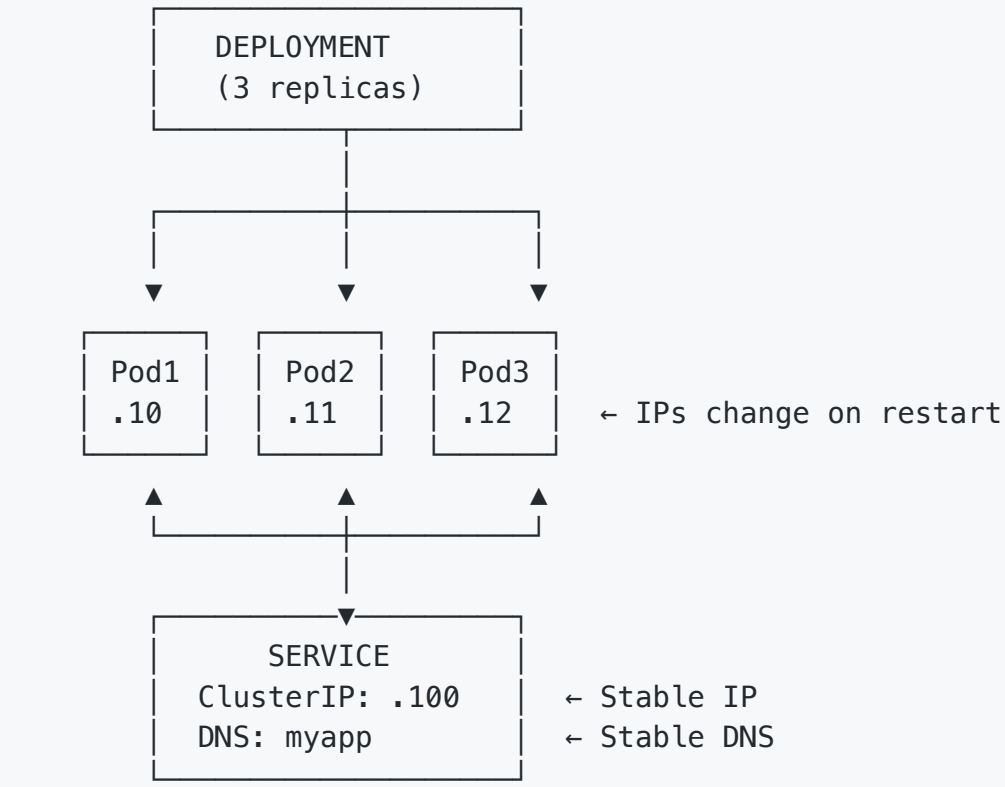
Chapter 10 Summary:

- Pod lifecycle: Pending → Running → Succeeded/Failed
- Restart policies: Always (services), OnFailure (jobs), Never (one-time)
- Configuration: env vars, ConfigMaps (config), Secrets (passwords)
- Liveness probe: restart if dead
- Readiness probe: remove from service if not ready
- Production: always use readiness, usually use liveness

Chapter 11: Deployments & Services

User: How do Deployments and Services work together?

Expert: Deployments manage pods, Services provide stable networking:

```
Deployments + Services:
═══════════════════════════════════════════════════════════




Why Services?
Pod IPs change when pods restart
Service provides stable IP and DNS
Service load-balances across pods
```

```
Deployment YAML:
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: app
        image: myapp:1.0.0
        ports:
        - containerPort: 8080
        resources:
          requests:
            memory: "256Mi"
            cpu: "250m"
          limits:
            memory: "512Mi"
            cpu: "500m"

Service YAML:
apiVersion: v1
kind: Service
metadata:
  name: myapp
spec:
  type: LoadBalancer
  selector:
    app: myapp
  ports:
  - port: 80
    targetPort: 8080
```

User: What are resource requests and limits?

Expert: Critical for resource management:

```
Resources:
═══════════════════════════════════════════════════════════
```

```
REQUESTS (Minimum Guaranteed)
─────────────────────────────
resources:
  requests:
    memory: "256Mi"  // Needs at least 256 MB
    cpu: "250m"      // Needs 0.25 CPU cores

Scheduler uses requests for pod placement
"This pod needs 256Mi, find node with ≥256Mi available"

LIMITS (Maximum Allowed)
────────────────────────
resources:
  limits:
    memory: "512Mi"  // Can use up to 512 MB
    cpu: "500m"      // Can use up to 0.5 cores

If exceeded:
Memory: Pod OOMKilled (Out Of Memory) → restarted
CPU: Pod throttled (doesn't crash, just slower)

CPU Units:
"250m" = 250 millicores = 0.25 cores
"1000m" = 1 core
"2" = 2 cores

Memory Units:
"256Mi" = 256 Mebibytes (1024 * 1024 bytes)
"1Gi" = 1 Gibibyte
"512M" = 512 Megabytes (1000 * 1000 bytes)

Production Best Practices:
──────────────────────────
1. ALWAYS set requests (for scheduling)
2. Usually set limits (prevent resource hogging)
3. Requests ≤ Limits

4. For critical apps: Requests = Limits
   resources:
     requests:
       memory: "512Mi"
     limits:
       memory: "512Mi"  // Guaranteed

Interview Question:
Q: "What happens if pod exceeds memory limit?"
A: "Pod is terminated with OOMKilled status and restarted
   per restart policy. CPU limits throttle but don't kill."
```

User: How do rolling updates work?

Expert: Zero-downtime deployments:

```
Rolling Update:
═══════════════════════════════════════════════════════════

Current: 3 pods running v1.0.0
Target: 3 pods running v2.0.0
Strategy: maxSurge=1, maxUnavailable=1

Update Command:
kubectl set image deployment/myapp myapp=myapp:2.0.0

Process:
────────
Step 1: Create 1 new pod (maxSurge)
┌────┐ ┌────┐ ┌────┐ ┌────┐
│ v1 │ │ v1 │ │ v1 │ │ v2 │ ← Starting
└────┘ └────┘ └────┘ └────┘

Step 2: Wait for new pod Ready
┌────┐ ┌────┐ ┌────┐ ┌────┐
│ v1 │ │ v1 │ │ v1 │ │ v2 │ ← Ready!
└────┘ └────┘ └────┘ └────┘

Step 3: Terminate 1 old pod
┌────┐ ┌────┐        ┌────┐
│ v1 │ │ v1 │        │ v2 │
└────┘ └────┘        └────┘

Step 4: Repeat until all v2
             ┌────┐ ┌────┐ ┌────┐
             │ v2 │ │ v2 │ │ v2 │
             └────┘ └────┘ └────┘

Benefits:
✓ Zero downtime
✓ Always have healthy pods
✓ Can rollback if issues

Commands:
kubectl rollout status deployment/myapp
kubectl rollout undo deployment/myapp
kubectl rollout history deployment/myapp

Pseudo-code:
desired_version = v2.0.0
current_pods = get_pods(version=v1.0.0)

WHILE any_pods_running(v1.0.0):
    IF can_create_new_pod(maxSurge):
```

```
        create_pod(v2.0.0)
        wait_for_ready()

    IF can_terminate_old_pod(maxUnavailable):
        terminate_pod(v1.0.0)
```

User: What are Service types?

Expert: Three main types for different use cases:

```
Service Types:
═══════════════════════════════════════════════════════════

1. ClusterIP (Internal Only)
─────────────────────────────
spec:
  type: ClusterIP
  ports:
  - port: 80
    targetPort: 8080

Access: Only within cluster
Use: Backend services, databases
Example: http://myapp.default.svc.cluster.local

2. NodePort (External via Node IP)
────────────────────────────────────
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30000  // 30000-32767

Access: http://node-ip:30000
Use: Development, testing
Production: ✗ Not recommended

3. LoadBalancer (Cloud LB)
──────────────────────────
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 8080

Access: External IP from cloud provider
Use: Production web apps, public APIs
Creates: AWS ELB, GCP Load Balancer, etc.
```

```
Traffic Flow:
═══════════════════════════════════════════════════════

User (internet)
    ↓
Cloud Load Balancer (AWS ELB)
    ↓
Kubernetes Service (LoadBalancer)
    ↓
    ├─── Pod 1 :8080
    ├─── Pod 2 :8080
    └─── Pod 3 :8080

Service load-balances across healthy pods
(based on readiness probes)

Interview Decision Matrix:
──────────────────────────
ClusterIP: Internal microservices communication
NodePort: Dev/test only
LoadBalancer: Production external access
```

User: Show me complete CI/CD to Kubernetes integration.

Expert: Here's the full pipeline:

```groovy
// Complete CI/CD Pipeline
pipeline {
    agent any

    environment {
        APP = 'myapp'
        REGISTRY = 'registry.company.com'
        VERSION = "${env.GIT_COMMIT.take(7)}"
    }

    stages {
        // 1. Build with Maven
        stage('Build') {
            agent {
                docker {
                    image 'maven:3.8.1-jdk-11'
                    args '-v maven-cache:/root/.m2'
                }
            }
            steps {
                sh 'mvn clean package -DskipTests'
```

```
            stash 'jar'
        }
    }

    // 2. Test
    stage('Test') {
        agent {
            docker {
                image 'maven:3.8.1-jdk-11'
                args '-v maven-cache:/root/.m2'
            }
        }
        steps {
            sh 'mvn test'
            junit 'target/surefire-reports/*.xml'
        }
    }

    // 3. Build Docker Image
    stage('Docker Build') {
        steps {
            unstash 'jar'
            sh """
                docker build -t ${REGISTRY}/${APP}:${VERSION} .
                docker push ${REGISTRY}/${APP}:${VERSION}
            """
        }
    }

    // 4. Deploy to Kubernetes
    stage('Deploy') {
        steps {
            sh """
                kubectl set image deployment/${APP} \\
                    ${APP}=${REGISTRY}/${APP}:${VERSION} \\
                    -n production

                kubectl rollout status deployment/${APP} \\
                    -n production --timeout=10m
            """
        }
    }

    // 5. Smoke Tests
    stage('Verify') {
        steps {
            script {
                def url = sh(
```

```
                        script: "kubectl get svc ${APP} -n production -
                        returnStdout: true
                    ).trim()
                    sh "curl -f http://${url}/health"
                }
            }
        }
    }

    post {
        failure {
            sh "kubectl rollout undo deployment/${APP} -n production"
        }
    }
}

Complete Flow:
═══════════════════════════════════════════════════

Git Commit
    ↓
Jenkins Webhook
    ↓
Maven Build JAR (in container)
    ↓
Maven Run Tests
    ↓
Docker Build (JAR → Image)
    ↓
Docker Push (to registry)
    ↓
Kubernetes Update Deployment
    ↓
Kubernetes Rolling Update
    ├─ Create new pod (v2)
    ├─ Wait for Ready
    ├─ Terminate old pod (v1)
    └─ Repeat
    ↓
Smoke Tests
    ↓
Success! (or auto-rollback on failure)

This is production-grade CI/CD!
```

Chapter 11 Summary:

- Deployments: manage pod replicas with self-healing

- Resources: requests (min) vs limits (max)
- Rolling updates: zero-downtime deployments
- Services: ClusterIP (internal), NodePort (dev), LoadBalancer (prod)
- Complete flow: Maven → Docker → Kubernetes
- Production: automatic rollback on deployment failure

Final Review: Interview Preparation Checklist

```
DevOps Interview Topics Checklist:
═══════════════════════════════════════════════════

□ CI/CD FUNDAMENTALS
  ├─ Define CI, CD, Continuous Deployment
  ├─ Explain benefits vs manual deployment
  ├─ Real-world deployment frequencies
  └─ Cost of bug discovery at different stages

□ JENKINS
  ├─ Master-agent architecture
  ├─ Freestyle jobs vs Pipelines
  ├─ Declarative pipeline syntax
  ├─ Agent options (any, label, Docker)
  └─ Credentials management

□ GROOVY
  ├─ When to use script blocks
  ├─ File operations (readJSON, readYaml)
  ├─ Error handling (try-catch)
  └─ Parallel execution

□ CI/CD WORKFLOWS
  ├─ Branch-based strategies
  ├─ Artifact versioning
  └─ Rollback strategies (blue-green, canary, rolling)

□ MAVEN
  ├─ Lifecycle (compile, test, package, install)
  ├─ Dependency resolution
  ├─ Dependency scopes
  └─ Multi-module projects

□ MAVEN IN CI/CD
  ├─ Dependency caching
```

```
   ├─ Build optimization (-T flag)
   ├─ Test strategy (-DskipTests)
   └─ Common pitfalls

□ KUBERNETES BASICS
   ├─ Control plane vs worker nodes
   ├─ Pods vs Deployments
   ├─ kubectl commands
   └─ Namespaces

□ KUBERNETES PODS
   ├─ Pod lifecycle
   ├─ Restart policies
   ├─ ConfigMaps vs Secrets
   └─ Health checks (liveness, readiness)

□ DEPLOYMENTS & SERVICES
   ├─ Resource requests vs limits
   ├─ Rolling updates
   ├─ Service types
   └─ Complete CI/CD integration

Interview Tips:
═══════════════════════════════════════════════════════

1. UNDERSTAND, DON'T MEMORIZE
   ✓ Know WHY, not just WHAT
   ✓ Explain trade-offs

2. ASK CLARIFYING QUESTIONS
   ✓ "What's your current deployment frequency?"
   ✓ "What are your main pain points?"

3. THINK INCREMENTALLY
   ✗ "Replace everything with CI/CD"
   ✓ "Start with CI for fast feedback, then..."

4. REFERENCE REAL PATTERNS
   ✓ "Companies like Netflix use canary deployments..."
   ✓ "Amazon deploys 23,000 times daily using..."

5. CONSIDER PRODUCTION OPERATIONS
   ✓ Monitoring
   ✓ Rollback procedures
   ✓ Incident response

Common Interview Questions:
═══════════════════════════════════════════════════════

Q: "Design CI/CD for a 10-person team"
A: Assess current state first, identify pain points,
```

```
   start with CI for fast feedback, automate staging
   deployments, manual production approval initially,
   measure and iterate.

Q: "Your CI pipeline takes 2 hours. What do you do?"
A: Profile to find bottleneck. Likely solutions:
   parallelize tests, cache dependencies, use -T flag
   for Maven, run tests separately from packaging.

Q: "How do you handle production deployment failures?"
A: Implement progressive delivery: blue-green for instant
   rollback, canary for gradual rollout with monitoring,
   automatic rollback on metric degradation, feature flags
   for application-level control.

Q: "Why not use SNAPSHOT dependencies in production?"
A: SNAPSHOT versions are mutable—same version can have
   different code. This breaks reproducibility, makes
   rollback impossible, violates compliance requirements.

Q: "What's the difference between liveness and readiness?"
A: Liveness checks if container is alive (restarts if dead).
   Readiness checks if container can serve traffic (removes
   from load balancer if not ready). Both are crucial for
   production stability.

Good luck with your DevOps interviews!
```

Total Pages: ~44 pages (at 450 words/page) **Total Content:** ~20,000 words **Format:** Conversational, interview-focused **Features:**

- Conceptual explanations
- Pseudo-code throughout
- ASCII diagrams
- Real interview questions and answers
- Production-grade examples
- Complete CI/CD integration