**JENKINS DOCUMENT**

**Jenkins Pipeline and Jenkins Stages**

**Introduction:** Jenkins is an open-source automation server widely used for continuous integration and continuous delivery (CI/CD). One of its powerful features is the **Jenkins Pipeline**, which allows developers to define an entire build process as code, providing better control, visibility, and automation of the software delivery lifecycle.

**What is a Jenkins Pipeline?**

A **Jenkins Pipeline** is a suite of plugins that support implementing and integrating continuous delivery pipelines into Jenkins. Pipelines are written as code, typically using a domain-specific language called **Pipeline DSL**, based on Groovy.

* **Declarative Pipeline:** A newer, more structured syntax that simplifies writing pipelines with predefined blocks like stages, steps, etc.
* **Scripted Pipeline:** An older, more flexible syntax based entirely on Groovy scripting.

**What are Jenkins Stages?**

**Stages** are sections within a pipeline that segments the process into discrete phases, such as building, testing, deploying, etc. They help organize and visualize the different parts of your pipeline, making the process more manageable and understandable.

**Typical Components of a Jenkins Pipeline**

* **Stages:** Logical divisions (e.g., Build, Test, Deploy)
* **Steps:** Actions performed within each stage (e.g., compile code, run tests)
* **Post conditions:** Actions to do after stages (e.g., cleanup, notifications)

**Example of a Declarative Jenkins Pipeline:**

**pipeline {**

**agent any**

**stages {**

**stage('Build') {**

**steps {**

**echo 'Building the project...'**

**sh 'mvn clean compile'**

**}**

**}**

**stage('Test') {**

**steps {**

**echo 'Running tests...'**

**sh 'mvn test'**

**}**

**}**

**stage('Deploy') {**

**steps {**

**echo 'Deploying application...'**

**sh 'deploy\_script.sh'**

**}**

**}**

**}**

**}**

* **pipeline:**The root block that defines the pipeline**.**
* **agent:**Specifies where to run the pipeline.
* **stages:**Container for all the stages.
* Each stage contains steps, which are individual commands or actions.

**Key Benefits of Using Pipelines and Stages:**

* **Visualization:** Jenkins UI shows the progress of each stage.
* **Parallel execution:** Stages can run in parallel for faster workflows.
* **Code versioning:** Pipelines are stored as code, enabling version control.
* **Pipeline as code:** Easier to maintain, review, and modify.

**Declarative Pipeline vs. Scripted Pipeline**

| **Feature** | **Declarative Pipeline** | **Scripted Pipeline** |
| --- | --- | --- |
| **Syntax** | Simpler, more structured, YAML-like syntax | Groovy-based, flexible scripting language |
| **Ease of writing** | Easier, especially for beginners | More complex, suitable for advanced users |
| **Readability** | Highly readable and maintainable | Less readable, especially for complex scripts |
| **Flexibility** | Less flexible; predefined blocks (stages, steps) | Highly flexible; can write complex logic |
| **Error handling** | Limited, designed to be straightforward | More advanced, customizable error handling |
| **Pipeline syntax** | Enforces a specific structure | Free-form Groovy code |
| **Use Cases** | CI/CD pipelines with standard steps | Complex workflows requiring custom scripting |

**Example of Declarative Pipeline**

**pipeline {**

**agent any**

**stages {**

**stage('Build') {**

**steps {**

**echo 'Building...'**

**sh 'make build'**

**}**

**}**

**stage('Test') {**

**steps {**

**echo 'Testing...'**

**sh 'make test'**

**}**

**}**

**stage('Deploy') {**

**steps {**

**echo 'Deploying...'**

**sh 'make deploy'**

**}**

**}**

**}**

**}**

**Example of Scripted Pipeline**

**node {**

**stage('Build') {**

**echo 'Building...'**

**sh 'make build'**

**}**

**stage('Test') {**

**echo 'Testing...'**

**sh 'make test'**

**}**

**stage('Deploy') {**

**echo 'Deploying...'**

**sh 'make deploy'**

**}**

**}**

**In brief:**

* Use Declarative Pipeline for simplicity, readability, and standard workflows.
* Use Scripted Pipeline when you need advanced scripting capabilities and dynamic control.\

**Why Do We Need to Configure Master and Slave (Agent) Nodes in Jenkins?**

In Jenkins, a **Master** (also called Controller) manages the entire Jenkins environment, handling tasks such as scheduling jobs, managing build queues, and providing the user interface.  
**Slaves** (or Agents) are additional machines configured to offload build and testing workloads from the master, enabling distributed and parallel execution of jobs.

**Reasons for Configuring Master and Slave Nodes**

1. **Scalability and Load Distribution**

* **Purpose:** Distribute build and testing loads across multiple machines.
* **Benefit:** Reduces pressure on the master server, ensuring Jenkins remains responsive and capable of handling many jobs simultaneously.

1. **Parallel Execution of Jobs**

* **Purpose:** Run multiple builds or tests at the same time on different agents.
* **Benefit:** Faster overall build times and efficient utilization of resources.

1. **Environment Segregation**

* **Purpose:** Different projects or workflows may require distinct environments (e.g., specific OS, tools, or configurations).
* **Benefit:** Assign jobs to agents configured with the necessary environment, avoiding conflicts on the master or other slaves.

1. **Resource Optimization**

* **Purpose:** Use agents with specialized hardware (e.g., GPU nodes, high-memory servers).
* **Benefit:** Optimize performance for resource-intensive tasks.

1. **Fault Tolerance and Availability**

* **Purpose:** Maintain build capabilities even if the master or any single agent fails.
* **Benefit:** Increase overall system reliability and availability.

1. **Security and Isolation**

* **Purpose:** Run untrusted or external code on isolated nodes.
* **Benefit:** Protect the master and other nodes from potential security risks.

1. **Cost Management**

* **Purpose:** Utilize cheaper hardware or cloud resources as slaves.
* **Benefit:** Control costs by leveraging scalable infrastructure.

**In Summary**

* **Master** node controls Jenkins and manages job scheduling.
* **Slave/Agent** nodes execute jobs, providing flexibility, scalability, and resource management.

Configuring master and slave nodes is essential for building a robust, scalable, and efficient Jenkins environment that can effectively handle complex, large-scale CI/CD workflows.

**Automating Build Tools and CI/CD Pipelines**

Automating build tools and CI/CD pipelines involves designing processes that automatically compile, test, analyze, and deploy software with minimal manual intervention. Here's an overview of how this can be achieved effectively:

**1. Selecting the Right Tools**

* **Continuous Integration Servers:** Jenkins, GitLab CI, GitHub Actions, CircleCI, Travis CI
* **Build Tools:** Maven, Gradle, Ant, npm, MSBuild, etc.
* **Version Control:** Git, SVN, Mercurial
* **Artifact Repositories:** Nexus, Artifactory
* **Deployment Tools:** Ansible, Kubernetes, Docker, Terraform

**2. Defining Pipeline as Code**

* Create scripts (Jenkinsfile, .gitlab-ci.yml, workflows) that specify all steps for building, testing, and deploying.
* These files are stored in version control, ensuring the pipeline configuration is versioned along with source code.

**Example (Jenkinsfile):**

**pipeline {**

**agent any**

**stages {**

**stage('Checkout') {**

**steps {**

**git 'https://github.com/example/repo.git'**

**}**

**}**

**stage('Build') {**

**steps {**

**sh './gradlew build'**

**}**

**}**

**stage('Test') {**

**steps {**

**sh './gradlew test'**

**}**

**}**

**stage('Deploy') {**

**steps {**

**sh './deploy.sh'**

**}**

**}**

**}**

**}**

**3. Automating Build and Test**

* Integrate build tools into the pipeline script.
* Once code is committed, the CI server automatically triggers the pipeline.
* Automated tests run in isolated environments, ensuring each build's integrity.

**4. Continuous Integration and Feedback**

* Every commit triggers build and test runs.
* Results (pass/fail) are communicated immediately via dashboards, emails, or chat notifications.
* Developers get rapid feedback, enabling quick fixes.

**5. Automating Deployment (CD)**

* Once code passes tests, the pipeline proceeds to deploy automatically to staging, testing, or production environments.
* Tools like Docker, Kubernetes, Ansible automate deployment steps.

**Example (Deploy step):**

docker build -t myapp:latest .

docker push myregistry/myapp:latest

kubectl apply -f deployment.yaml

**6. Infrastructure as Code (IaC)**

* Automate provisioning of infrastructure using scripts (Terraform, CloudFormation).
* Ensures environments are consistent and reproducible.

**7. Monitoring and Feedback**

* Implement monitoring tools (Prometheus, ELK Stack) to track the health of deployments and infrastructure.
* Use dashboards and alerts for proactive management.

**Summary of the Process**

1. **Version control** holds source code and pipeline scripts.
2. **Trigger pipeline** automatically on code commits or PRs.
3. **Build** compiles code and generates artifacts.
4. **Test** runs automated tests for validation.
5. **Analyze** code quality and security scans.
6. **Deploy** to target environments automatically.
7. **Notify** stakeholders of build status, successes, or failures.
8. **Repeat** continuously for rapid, reliable delivery.

**In short:**

By integrating version control, scripting pipelines, and configuring automation tools, we streamline and speed up software delivery, reduce manual errors, and improve deployment confidence—forming the core of modern CI/CD practices.

**Steps to Take When a Pipeline Gets Aborted or a Build Fails**

When your Jenkins pipeline is aborted or encounters a failure, it's important to follow a structured approach to diagnose, fix, and prevent future issues.

**1. Review Build Logs and Console Output**

* **Action:** Check the detailed console logs of the failed or aborted build.
* **Purpose:** Identify the exact step where failure occurred and understand the error messages or exceptions.

**2. Analyze the Error Details**

* **Common issues:**
  + Compilation errors
  + Failed tests
  + Environment or dependency issues
  + Configuration or syntax errors
  + External service failures (e.g., database, APIs)
* **Action:** Pinpoint the root cause based on log messages.

**3. Verify the Environment and Dependencies**

* **Action:** Ensure that:
  + The build environment (agents/master) has the correct tools and versions installed.
  + Dependencies or external services are available and accessible.
  + No network issues or resource limitations affected the build.

**4. Reproduce the Issue Locally**

* **Action:** Try running the same build commands locally on your machine or in a similar environment.
* **Purpose:** Confirm whether the problem is environment-specific or code-related.

**5. Fix the Identified Issue**

* **Possible fixes:**
  + Correct code errors or syntax issues
  + Update dependencies or tools
  + Fix configuration or environment setup
  + Modify pipeline script if there are logical errors
* **Action:** Make the necessary code changes or environment adjustments.

**6. Rerun the Pipeline**

* **Action:** Trigger the build again from Jenkins.
* **Best Practice:** Use "rebuild" options or manually trigger a new build with identical parameters.

**7. Implement Fail-Safe and Notifications**

* **Actions:**
  + Set up notifications (email, Slack) to inform stakeholders of failures.
  + Use "post" actions in Jenkinsfile to send alerts or perform cleanup.

**8. Prevent Future Failures**

* **Best practices:**
  + Add validation or static code analysis steps
  + Implement resource checks before critical steps
  + Use retries for flaky steps
  + Maintain good test coverage to catch issues early
  + Automate dependency management

**9. Document the Issue and Resolution**

* **Action:** Record what caused the failure and how it was fixed.
* **Purpose:** Improve team knowledge and enable quicker troubleshooting next time.

**Summary:**

* Investigate logs to identify root cause
* Verify environment and dependencies
* Reproduce the issue locally if possible
* Implement fixes and rerun the build
* Improve pipeline resilience and notifications
* Document the incident for future reference

**What to Do If Your Jenkins Master Server Is Down**

When the Jenkins master server becomes unavailable or is down, it affects the entire CI/CD pipeline since it manages jobs, configurations, and builds. Here's a structured approach to handle the situation:

**1. Identify the Cause of the Downtime**

* Check server hardware, network connectivity, or cloud provider status.
* Review logs (if accessible via server interface or remote logging).
* Determine whether the server crashed, experienced resource exhaustion, or encountered a configuration issue.

**2. Restore the Jenkins Master Server**

**If the server is down due to hardware failure or crash:**

* **Restore from Backup:**
  + Retrieve the latest Jenkins home directory backup, including job configurations, system settings, plugins, and credentials.
  + Deploy the backup to a new or existing server.
* **Reinstall Jenkins:**
  + Install the same Jenkins version used previously.
  + Restore from backup to recover all configurations.

**If the server is accessible in some capacity:**

* Restart the Jenkins service:

sudo systemctl restart jenkins

* Check logs for issues preventing startup:

sudo journalctl -u jenkins

**3. Promote an Agent or Slave as a Temporary Master (if applicable)**

* If you have **agent nodes configured with master capabilities** and a setup for **redundant infrastructure**, consider promoting an agent to be the new master temporarily.
* But this depends on your setup and infrastructure policies.

**4. Validate the System After Recovery**

* Ensure Jenkins is fully operational.
* Check that all jobs and configurations are intact.
* Verify that plugins are active and up-to-date.
* Test critical pipelines to confirm normal operation.

**5. Implement or Improve High Availability (HA) Strategies**

* To prevent future downtime, consider setting up **Jenkins in a high-availability cluster** using:
  + Multi-master setup (with caveats)
  + Shared or replicated storage for Jenkins home
  + Backup and restore procedures

**6. Notify Stakeholders**

* Inform development and QA teams about the downtime.
* Provide an estimate for recovery if ongoing issues exist.

**Summary**

* **Diagnose** the root cause.
* **Restore** from backups or redeploy Jenkins.
* **Validate** the system after recovery.
* **Implement HA strategies** to minimize future risks.

**What to Do When Jenkins Jobs Take Too Much Time to Run**

When Jenkins builds are slow, it can impact development velocity and productivity. Here are steps to diagnose, optimize, and improve build times:

**1. Identify Bottlenecks**

* **Check Build Logs:** Review console output to identify slow steps.
* **Monitor Resource Usage:** Use tools like top, htop, or Jenkins monitoring plugins to check CPU, memory, disk I/O, and network utilization.
* **Analyze Duration of Each Stage:** Use Jenkins' built-in timing reports or plugins (e.g., Timestats) to see which stages or steps are slow.

**2. Optimize the Build Process**

* **Parallelize Tasks:**
  + Use Jenkins pipeline's parallel directive to run independent stages simultaneously.
  + Example:

stage('Parallel Tasks') {

parallel {

stage('Test') { steps { sh 'make test' } }

stage('Build') { steps { sh 'make build' } }

}

}

* **Cache Dependencies:**
  + Use caching mechanisms (e.g., Maven local repository, npm cache, Docker layers) to avoid redundant downloads or rebuilds.
* **Incremental Builds:**
  + Configure build tools to perform incremental or only changed component builds.

**3. Improve Infrastructure**

* **Dedicated Agents/Nodes:**
  + Assign builds to faster, dedicated nodes with sufficient resources.
* **Use SSDs and Adequate Hardware:**
  + Ensure build agents have fast disks, enough CPU cores, and RAM.
* **Scaling:**
  + Add more agents or scale cloud resources depending on demand.

**4. Optimize Code and Tests**

* **Reduce Test Time:**
  + Use optimized or selective tests.
  + Parallelize tests if possible.
* **Profile Build Steps:**
  + Identify slow scripts or commands and optimize or replace them.

**5. Simplify or Modularize Pipelines**

* Break large pipelines into smaller, modular jobs that can run independently or in parallel.
* Avoid unnecessary steps or redundant processes.

**6. Use Build Caching and Artifacts Effectively**

* Cache build artifacts between runs.
* Reuse previous build outputs when appropriate.

**7. Adjust Build Frequency and Triggers**

* Only trigger builds when necessary (e.g., on code changes, scheduled intervals).
* Use polling or webhook triggers judiciously.

**8. Review and Optimize Plugins**

* Remove unnecessary plugins.
* Keep plugins up-to-date for performance improvements.

**Summary**

* **Diagnose** slow steps using logs and resource monitoring.
* **Parallelize** tasks and use caching.
* **Upgrade infrastructure** if needed.
* **Optimize code, tests, and pipeline structure**.
* **Review triggers** to avoid unnecessary builds.

**1. CI — Continuous Integration**

* **Definition:** The practice of automatically integrating code changes from multiple contributors into a shared repository multiple times a day.
* **Goal:** Detect integration issues early, ensure code is always in a deployable state.
* **Key Activities:**
  + Developers commit code frequently.
  + Automated builds and tests run on each commit.
  + Feedback is provided quickly if any issues occur.

**Example:** When a developer pushes new code, Jenkins automatically builds the project and runs tests to verify code correctness.

**2. CD — Continuous Delivery**

* **Definition:** Extending CI to automatically prepare code changes for release to production, ensuring that the codebase can be deployed at any time.
* **Goal:** Keep the software in a deployable state, with automated deployment processes that can be triggered manually.
* **Difference from Continuous Deployment:** The deployment to production happens manually but the preparation (building, testing, staging) is automated.

**Example:** After passing all tests, the code is automatically deployed to a staging environment. Deployment to production is performed manually but is easy and reliable.

**3. CD — Continuous Deployment**

* **Definition:** Further extends CD, where every change that passes all automated tests is automatically deployed to production without manual intervention.
* **Goal:** Achieve fully automated, rapid software release cycles.

**Example:** A new feature passes test automation and is automatically released to users immediately.

**Summary of Differences**

| **Aspect** | **Continuous Integration (CI)** | **Continuous Delivery (CD)** | **Continuous Deployment (CD)** |
| --- | --- | --- | --- |
| **Definition** | Automated merging & testing of code changes | Automated deployment readiness to staging or production | Automatic release of code to production after tests pass |
| **Automation scope** | Build and test | Build, test, and prepare deployment | Build, test, and deploy automatically |
| **Manual step?** | No (every commit triggers CI) | Yes (manual trigger for deployment) | No (deployments happen automatically) |
| **Goal** | Detect issues early | Keep code deployable at all times | Rapid, reliable releases without manual intervention |

**In a nutshell:**

* **CI** helps catch integration issues early.
* **CD (Delivery)** automates release preparation, making deployments easier.
* **CD (Deployment)** automates the actual release process, ensuring continuous, rapid delivery.

**1. Automates Build and Deployment Processes**

* Jenkins automates compiling code, running tests, packaging applications, and deploying to environments, reducing manual effort and errors.

**2. Facilitates Continuous Integration and Continuous Delivery**

* Enables developers to integrate code frequently, with automated testing ensuring that new changes do not break the existing system.
* Supports automated deployment pipelines, making releases faster and more reliable.

**3. Supports a Wide Range of Tools and Languages**

* Jenkins has a vast ecosystem of plugins, supporting many programming languages (Java, Python, Node.js, etc.) and tools (Git, Maven, Docker, Kubernetes, etc.).

**4. Extensible and Modular Architecture**

* The plugin-based architecture allows customization. You can add new functionalities like notifications, code quality analysis, static code checks, etc.

**5. Easy Configuration and Usage**

* Offers a user-friendly web interface for setting up, configuring, and monitoring build jobs.
* Supports "Pipeline as Code" with Jenkinsfiles, enabling version-controlled, reproducible build processes.

**6. Improving Software Quality and Delivery Speed**

* Automated testing and deployment reduce manual errors.
* Fast feedback loops help developers identify issues early, improving quality.
* Faster releases give competitive advantage and customer satisfaction.

**7. Open Source and Cost-Effective**

* Jenkins is free and open-source, with a large community providing support and plugins.

**8. Community and Plugins Support**

* Extensive community support, plugins for integration with cloud platforms, version control, testing frameworks, notifications, etc.

**In summary:**

**Jenkins is used because it streamlines, automates, and accelerates the software development lifecycle, ensuring higher quality, faster releases, and easier management of complex workflows.**

**1. GitLab CI/CD**

* Integrated directly into GitLab repositories.
* Uses .gitlab-ci.yml files for pipeline definitions.
* Supports auto-scaling runners, Docker, Kubernetes, and more.
* Suitable for teams using GitLab as their source code management.

**2. GitHub Actions**

* Native CI/CD solution within GitHub repositories.
* Define workflows in .github/workflows YAML files.
* Supports running jobs on GitHub-hosted or self-hosted runners.
* Easily integrates with GitHub ecosystem features**.**

**3. CircleCI**

* Cloud-based CI/CD platform.
* Supports Docker, Kubernetes, and various cloud providers.
* YAML-based configuration.
* Known for fast builds and good Docker support.

**4. Travis CI**

* Cloud-hosted or self-hosted.
* Configuration via .travis.yml.
* Popular in open-source projects for seamless GitHub integration.

**5. Azure DevOps Pipelines**

* Microsoft's CI/CD solution integrated into Azure DevOps.
* Supports YAML pipelines and classic editor.
* Deep integration with other Azure services.

**6. Bitbucket Pipelines**

* Built into Bitbucket repositories.
* YAML configuration file.
* Supports Docker, Kubernetes, and cloud integrations.

**7. Bamboo**

* Atlassian's CI/CD server product.
* Integrates tightly with Jira and Bitbucket.
* Supports complex workflows with dedicated deployment projects.

**8. Spinnaker**

* Open-source multi-cloud continuous delivery platform.
* Focuses more on deployment automation across cloud providers.
* Works well with Jenkins or other CI tools for build automation.

**9. Argo CD**

* Kubernetes-native continuous delivery tool.
* Automates deploying applications in Kubernetes.
* Integrates with GitOps workflows.

**Summary:**

**While Jenkins is very popular and powerful, many other tools exist for CI/CD, each suited to different needs or environments. The choice depends on factors like existing infrastructure, cloud platform, team familiarity, and project requirements.**

**Jenkins has an extensive ecosystem of plugins that enhance its functionality. The plugins you use depend on your project requirements, environment, and workflow. Here are some of the most popular and widely used Jenkins plugins:**

**1. Git Plugin**

* Purpose: Integrates Git repositories with Jenkins.
* Use case: Cloning repositories, polling for changes, triggering builds on commits.

**2. Maven Integration Plugin**

* Purpose: Supports building Maven projects.
* Use case: Run Maven goals, integrate Maven build steps**.**

**3. Pipeline Plugin**

* Purpose: Enables defining workflows as code using Jenkinsfiles.
* Use case: Create complex, scriptable build pipelines.

**4. Docker Plugin**

* Purpose: Builds, pushes, and manages Docker containers.
* Use case: Containerized builds and deployments.

**5. Credentials Plugin**

* Purpose: Securely manage credentials like passwords, SSH keys, tokens.
* Use case: Use credentials safely in pipelines and jobs.

1. **Slack Notification Plugin**

* Purpose: Send build status notifications to Slack channels.
* Use case: Real-time team notifications.

**7. Blue Ocean Plugin**

* Purpose: Modern, user-friendly UI for Jenkins pipelines.
* Use case: Better visualization and flow management of pipelines.

**8. JUnit Plugin**

* Purpose: Records and displays test results.
* Use case: Visualize unit test reports in Jenkins**.**

**9. Email Extension Plugin**

* Purpose: Send customizable email notifications.
* Use case: Notify developers on build failures or success**.**

**10. Artifactory Plugin**

* Purpose: Integrate with JFrog Artifactory for artifact management.
* Use case: Store, retrieve, and promote build artifacts.

**11. Kubernetes Plugin**

* Purpose: Run builds in Kubernetes environments.
* Use case: Dynamic provisioning of build agents.

**12. SonarQube Scanner Plugin**

* Purpose: Integrate code quality analysis.
* Use case: Run static code analysis during the build.

**13. Allure Report Plugin**

* Purpose: Generate detailed test reports.
* Use case: Enhance test result visualization.

**Parameterized Build in Jenkins**

A **Parameterized Build** is a type of Jenkins job that allows you to pass **parameters** (inputs) to the build process at the time of triggering. This makes your builds more flexible and configurable, enabling you to run the same job with different settings without creating multiple jobs.

**Key Concepts**

* **Parameters:** Variables defined in the job configuration that can accept user input, environment variables, or dynamic values.
* **Usage:** Parameters influence build behavior—like selecting a branch, version number, environment configuration, or other options.

**Advantages of Parameterized Builds**

* **Flexibility:** Run builds with different inputs without creating multiple jobs.
* **Interactivity:** End users or scripts can specify options at build time.
* **Automation:** Dynamic values can be supplied via scripts or APIs.

**Common Types of Parameters**

* **String Parameter:** Text input (e.g., version number).
* **Boolean Parameter:** Yes/No options (e.g., whether to deploy).
* **Choice Parameter:** Dropdown list of options.
* **Password Parameter:** Secure input.
* **File Parameter:** Upload files for build input.

**How to Create a Parameterized Build**

1. **Configure Job:**
   * Go to your Jenkins job → Click "Configure."
2. **Enable Parameters:**
   * Check "This build is parameterized."
3. **Add Parameters:**
   * Select parameter types and define their default values and descriptions.
4. **Use Parameters in Build Steps:**
   * Reference parameter values in build scripts using syntax like $PARAMETER\_NAME or ${PARAMETER\_NAME}.

**Example (shell script in build step):**

echo "Deploying version: $VERSION"

**Triggering Parameterized Builds**

* Manually: Parameter inputs are prompted when clicking "Build with Parameters."
* Via scripts or APIs: Parameters can be passed programmatically.

**Troubleshooting a Failed Jenkins Job**

When a Jenkins job fails, systematic troubleshooting helps identify and resolve issues efficiently. Here’s a structured approach:

**1. Review the Console Output**

* **Access the build’s console log:** Click on the failed build→ "Console Output."
* **Look for error messages:** Identify the specific step or command that failed.
* **Check for stack traces or exceptions:** These often point directly to the root cause.

**2. Analyze the Error Details**

* **Common issues include:**
  + Compilation errors
  + Failed tests
  + Missing dependencies
  + Environment problems
  + Network issues
  + Permission/access rights
* **Identify the nature of the failure** (syntax error, timeout, environment mismatch).

**3. Verify Build Environment and Configuration**

* **Check environment variables:** Ensure they are correctly set.
* **Confirm dependencies:** Are required tools, libraries, or services available and correctly configured?
* **Validate node/agent status:** Ensure the agent executing the job is online and has sufficient resources (CPU, RAM, disk space).

**4. Reproduce the Issue Locally**

* **Run the same commands manually** on the agent machine or local environment.
* **Purpose:** Verify if the issue is environment-specific or code-related.

**5. Test and Validate**

* **Run tests independently** to confirm which tests are failing.
* **Update or fix the code/configuration** based on errors.

**6. Check External Systems and Services**

* For jobs that depend on external APIs, databases, or services:
  + Verify their availability.
  + Check network connectivity.
  + Confirm correct credentials and permissions.

**7. Review Recent Changes**

* **Code changes:** Might introduce a bug.
* **Configuration updates:** Could cause environment issues.
* **Dependency upgrades:** May lead to incompatibility.

**8. Use Debugging and Diagnostic Tools**

* Insert debugging statements (echo, printenv, set -x, etc.) into build steps.
* Enable additional logging if supported by plugins.

**9. Check for Infrastructure or Resource Issues**

* Ensure the build agent or server isn't resource-starved.
* Look for disk space, memory, or network problems.

**10. Search for Known Issues**

* Check plugin or tool logs.
* Search online for error messages or logs.
* Consult community forums or Jira if using enterprise tools.

**11. Fix and Rerun**

* Apply necessary fixes.
* Trigger a rerun (manual or automatically triggered).
* Monitor logs closely to confirm resolution.

**Proactive Tips**

* **Set up notifications:** To alert team on failures.
* **Implement post-build actions:** Send reports or snapshots for analysis.
* **Automate cleanup:** Remove stale build artifacts and logs.

**In summary:**

* Start with the **console log** review.
* Isolate the **error message**.
* Confirm environment and dependencies.
* Reproduce issues locally.
* Fix issues and **rerun**.
* Document recurring problems to improve stability.

**Upstream and Downstream in Jenkins (and CI/CD Workflows)**

**Definition**

* **Upstream** and **Downstream** refer to the relationship between Jenkins jobs or stages based on their execution dependencies:
  + **Upstream job/stage:** The job/stage that initiates or triggers another.
  + **Downstream job/stage:** The job/stage that is triggered or depends on the upstream.

**In Jenkins context:**

* **Upstream Job:**
  + A job that triggers another job after successful completion.
  + Example: **Build job** is upstream of **Test job**.
* **Downstream Job:**
  + A job that runs after an upstream job completes.
  + Example: **Deploy job** is downstream of **Test job**.

**Visual Representation**

[Job A (Upstream)] ---> [Job B (Downstream)]

* **Job A** runs first; upon success, it triggers **Job B**.

**Purpose**

* To define build pipelines with sequential steps.
* Manage dependencies between jobs.
* Enable complex workflows in CI/CD pipelines.

**Implementation in Jenkins**

* Use **"Build other projects"** post-build action.
* Use **Pipeline syntax:** build job step, or stage dependencies.
* Plugins like **Parameterized Trigger Plugin** facilitate downstream triggers with parameters.

**Example**

// In a Jenkins pipeline

stage('Build') {

// Trigger test job after build

build(job: 'TestJob')

}

**Summary**

* **Upstream:** Job/stage that initiates triggers.
* **Downstream:** Job/stage that receives trigger or depends on upstream.

Understanding these relationships helps design efficient, automated build pipelines with clear dependencies**Upstream and Downstream in Jenkins (and CI/CD workflows)**

**Definition:**

* **Upstream:** Refers to a job or process that **triggers or feeds** into another job. It is the **source** or **preceding** job.
* **Downstream:** Refers to a job or process **triggered by** or **dependent on** the upstream job. It is the **dependent** or **subsequent** job.

**In Context of Jenkins:**

* When a **build job** (upstream) completes successfully, it can **trigger** one or more other jobs (downstream).
* This chain of jobs helps create **pipelines of dependent tasks** — e.g., build → test → deploy.

**Example Scenario:**

* **Upstream Job:** Build the project.
* **Downstream Job:** Run integration tests after the build completes successfully.
* **Further Downstream:** Deploy to staging after tests pass.

**Visualization:**

Build (Upstream)

|

v

Test (Downstream)

|

v

Deploy (Further downstream)

**How This Works in Jenkins:**

* You can configure \*\*.

Would you like an example of configuring upstream/downstream jobs or how to set them up in a pipeline?"Build other projects"\*\* in a job’s post-build actions.

* You can use **Pipeline syntax** to define **downstream triggers**.
* **Parameter passing**: Upstream jobs can pass parameters to downstream jobs.

**Summary:**

* **Upstream:** The source job that initiates or feeds into another job.
* **Downstream:** The job that is triggered or depends on the upstream job.

Understanding this relationship helps in designing effective CI/CD pipelines, controlling build flow, and managing dependencies.