Unit 4

Integrating the system

Build systems

A build system is a key component in DevOps, and it plays an important role in the software development and delivery process. It automates the process of compiling and packaging source code into a deployable artifact, allowing for efficient and consistent builds.

Here are some of the key functions performed by a build system:

Compilation: The build system compiles the source code into a machine-executable format, such as a binary or an executable jar file.

Dependency Management: The build system ensures that all required dependencies are available and properly integrated into the build artifact. This can include external libraries, components, and other resources needed to run the application.

Testing: The build system runs automated tests to ensure that the code is functioning as intended, and to catch any issues early in the development process.

Packaging: The build system packages the compiled code and its dependencies into a single, deployable artifact, such as a Docker image or a tar archive.

Version Control: The build system integrates with version control systems, such as Git, to track changes to the code and manage releases.

Continuous Integration: The build system can be configured to run builds automatically whenever changes are made to the code, allowing for fast feedback and continuous integration of new code into the main branch.

Deployment: The build system can be integrated with deployment tools and processes to automate the deployment of the build artifact to production environments.

In DevOps, it's important to have a build system that is fast, reliable, and scalable, and that can integrate with other tools and processes in the software development and delivery pipeline. There are many build systems available, each with its own set of features and capabilities, and choosing the right one will depend on the specific needs of the project and team.

Jenkins build server

What is Jenkin?

Jenkins is an open source automation tool written in Java programming language that allows continuous integration.

Jenkins **builds** and **tests** our software projects which continuously making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build.

It also allows us to continuously **deliver** our software by integrating with a large number of testing and deployment technologies.

Jenkins offers a straightforward way to set up a continuous integration or continuous delivery environment for almost any combination of languages and source code repositories using pipelines, as well as automating other routine development tasks.

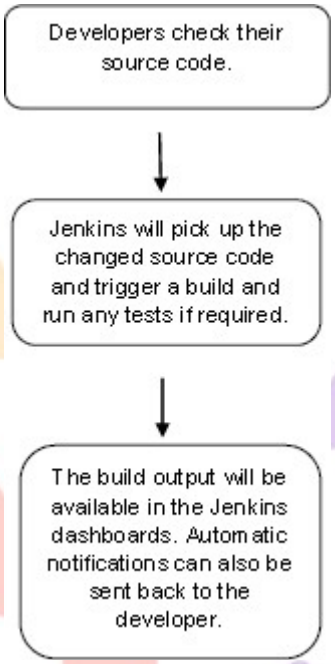
With the help of Jenkins, organizations can speed up the software development process through automation. Jenkins adds development life-cycle processes of all kinds, including build, document, test, package, stage, deploy static analysis and much more.

Jenkins achieves CI (Continuous Integration) with the help of plugins. Plugins is used to allow the integration of various DevOps stages. If you want to integrate a particular tool, you have to install the plugins for that tool. For example: Maven 2 Project, Git, HTML Publisher, Amazon EC2, etc.

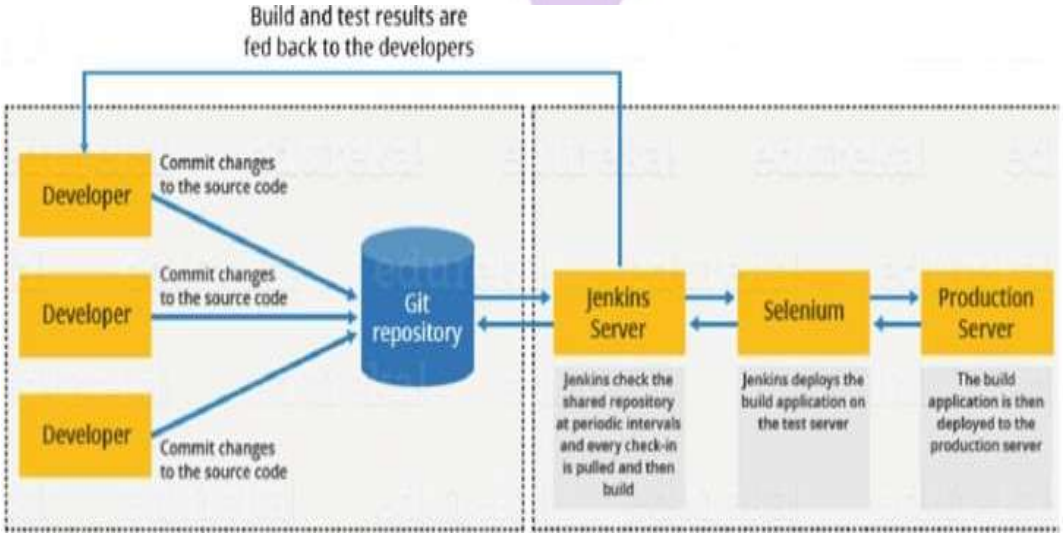
For example: If any organization is developing a project, then **Jenkins** will continuously test your project builds and show you the errors in early stages of your development.

Possible steps executed by Jenkins are for example:

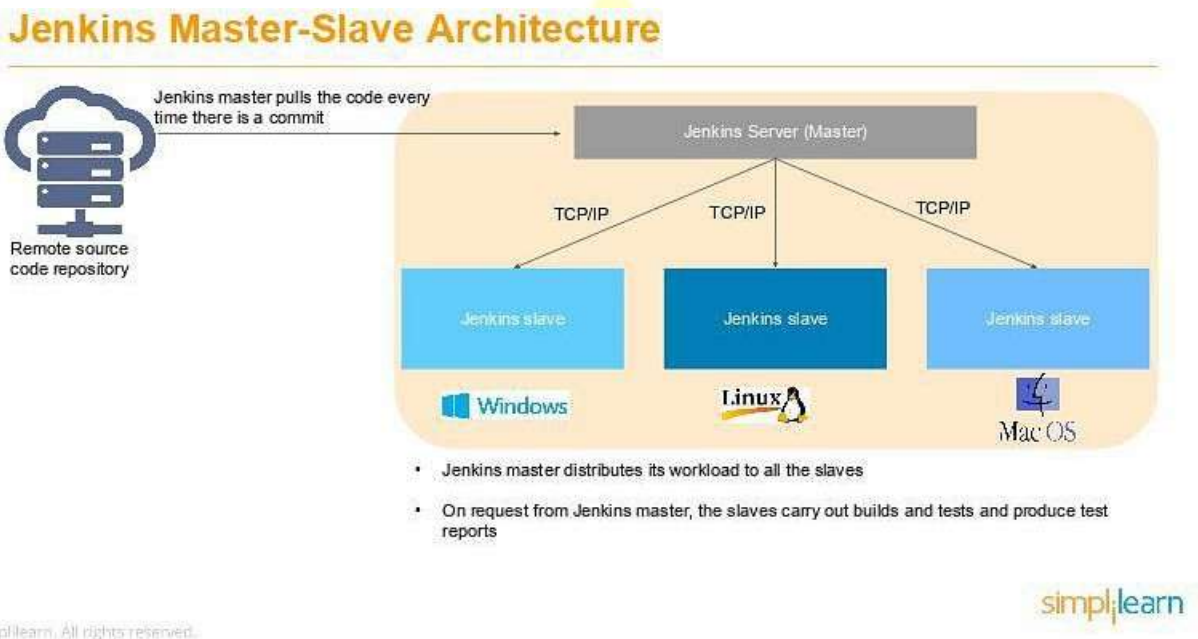
- Perform a software build using a build system like Gradle or Maven Apache
- Execute a shell script
- Archive a build result
- Running software tests



Jenkin workflow



Jenkins Master-Slave Architecture



As you can see in the diagram provided above, on the left is the Remote source code repository. The Jenkins server accesses the master environment on the left side and the master environment can push down to multiple other Jenkins Slave environments to distribute the workload.

That lets you run multiple builds, tests, and product environment across the entire architecture. Jenkins Slaves can be running different build versions of the code for different operating systems and the server Master controls how each of the builds operates.

Supported on a master-slave architecture, Jenkins comprises many slaves working for a master. This architecture - the Jenkins Distributed Build - can run identical test cases in different environments. Results are collected and combined on the master node for monitoring.

Jenkins Applications

Jenkins helps to automate and accelerate the software development process. Here are some of the most common applications of Jenkins:

1. Increased Code Coverage

Code coverage is determined by the number of lines of code a component has and how many of them get executed. Jenkins increases code coverage which ultimately promotes a transparent development process among the team members.

2. No Broken Code

Jenkins ensures that the code is good and tested well through continuous integration. The final code is merged only when all the tests are successful. This makes sure that no broken code is shipped into production.

What are the Jenkins Features?

Jenkins offers many attractive features for developers:

- **Easy Installation**

  Jenkins is a platform-agnostic, self-contained Java-based program, ready to run with packages for Windows, Mac OS, and Unix-like operating systems.

- **Easy Configuration**

  Jenkins is easily set up and configured using its web interface, featuring error checks and a built-in help function.

- **Available Plugins**

  There are hundreds of plugins available in the Update Center, integrating with every tool in the CI and CD toolchain.

- **Extensible**

  Jenkins can be extended by means of its plugin architecture, providing nearly endless possibilities for what it can do.

- **Easy Distribution**

  Jenkins can easily distribute work across multiple machines for faster builds, tests, and deployments across multiple platforms.

- **Free Open Source**

  Jenkins is an open-source resource backed by heavy community support.

As a part of our learning about what is Jenkins, let us next learn about the Jenkins architecture.

Jenkins build server

Jenkins is a popular open-source automation server that helps developers automate parts of the software development process. A Jenkins build server is responsible for building, testing, and deploying software projects.

A Jenkins build server is typically set up on a dedicated machine or a virtual machine, and is used to manage the continuous integration and continuous delivery (CI/CD) pipeline for a software project. The build server is configured with all the necessary tools, dependencies, and plugins to build, test, and deploy the project.

The build process in Jenkins typically starts with code being committed to a version control system (such as Git), which triggers a build on the Jenkins server. The Jenkins server then checks out the code, builds it, runs tests on it, and if everything is successful, deploys the code to a staging or production environment.

Jenkins has a large community of developers who have created hundreds of plugins that extend its functionality, so it's easy to find plugins to support specific tools, technologies, and workflows. For example, there are plugins for integrating with cloud infrastructure, running security scans, deploying to various platforms, and more.

Overall, a Jenkins build server can greatly improve the efficiency and reliability of the software development process by automating repetitive tasks, reducing the risk of manual errors, and enabling developers to focus on writing code.

Managing build dependencies

Managing build dependencies is an important aspect of continuous integration and continuous delivery (CI/CD) pipelines. In software development, dependencies refer to external libraries, tools, or resources that a project relies on to build, test, and deploy. Proper management of dependencies can ensure that builds are repeatable and that the build environment is consistent and up-to-date.

Here are some common practices for managing build dependencies in Jenkins:

Dependency Management Tools: Utilize tools such as Maven, Gradle, or npm to manage dependencies and automate the process of downloading and installing required dependencies for a build.

Version Pinning: Specify exact versions of dependencies to ensure builds are consistent and repeatable.

Caching: Cache dependencies locally on the build server to improve build performance and reduce the time it takes to download dependencies.

Continuous Monitoring: Regularly check for updates and security vulnerabilities in dependencies to ensure the build environment is secure and up-to-date.

Automated Testing: Automated testing can catch issues related to dependencies early in the development process.

By following these practices, you can effectively manage build dependencies and maintain the reliability and consistency of your CI/CD pipeline.

Jenkins plugins

Jenkins plugins are packages of software that extend the functionality of the Jenkins automation server. Plugins allow you to integrate Jenkins with various tools, technologies, and workflows, and can be easily installed and configured through the Jenkins web interface.

Some popular Jenkins plugins include:

Git Plugin: This plugin integrates Jenkins with Git version control system, allowing you to pull code changes, build and test them, and deploy the code to production.

Maven Plugin: This plugin integrates Jenkins with Apache Maven, a build automation tool commonly used in Java projects.

Amazon Web Services (AWS) Plugin: This plugin allows you to integrate Jenkins with Amazon Web Services (AWS), making it easier to run builds, tests, and deployments on AWS infrastructure.

Slack Plugin: This plugin integrates Jenkins with Slack, allowing you to receive notifications about build status, failures, and other important events in your Slack channels.

Blue Ocean Plugin: This plugin provides a new and modern user interface for Jenkins, making it easier to use and navigate.

Pipeline Plugin: This plugin provides a simple way to define and manage complex CI/CD pipelines in Jenkins.

Jenkins plugins are easy to install and can be managed through the Jenkins web interface. There are hundreds of plugins available, covering a wide range of tools, technologies, and use cases, so you can easily find the plugins that best meet your needs.

By using plugins, you can greatly improve the efficiency and automation of your software development process, and make it easier to integrate Jenkins with the tools and workflows you use.

Git Plugin

The Git Plugin is a popular plugin for Jenkins that integrates the Jenkins automation server with the Git version control system. This plugin allows you to pull code changes from a Git repository, build and test the code, and deploy it to production.

With the Git Plugin, you can configure Jenkins to automatically build and test your code whenever changes are pushed to the Git repository. You can also configure it to build and test code on a schedule, such as once a day or once a week.

The Git Plugin provides a number of features for managing code changes, including:

Branch and Tag builds: You can configure Jenkins to build specific branches or tags from your Git repository.

Pull Requests: You can configure Jenkins to build and test pull requests from your Git repository, allowing you to validate code changes before merging them into the main branch.

Build Triggers: You can configure Jenkins to build and test code changes whenever changes are pushed to the Git repository or on a schedule.

Code Quality Metrics: The Git Plugin integrates with tools such as SonarQube to provide code quality metrics, allowing you to track and improve the quality of your code over time.

Notification and Reporting: The Git Plugin provides notifications and reports on build status, failures, and other important events. You can configure Jenkins to send notifications via email, Slack, or other communication channels.

By using the Git Plugin, you can streamline your software development process and make it easier to manage code changes and collaborate with other developers on your team.

file system layout

In DevOps, the file system layout refers to the organization and structure of files and directories on the systems and servers used for software development and deployment. A well-designed file system layout is critical for efficient and reliable operations in a DevOps environment.

Here are some common elements of a file system layout in DevOps:

Code Repository: A central code repository, such as Git, is used to store and manage source code, configuration files, and other artifacts.

Build Artifacts: Build artifacts, such as compiled code, are stored in a designated directory for easy access and management.

Dependencies: Directories for storing dependencies, such as libraries and tools, are designated for easy management and version control.

Configuration Files: Configuration files, such as YAML or JSON files, are stored in a designated directory for easy access and management.

Log Files: Log files generated by applications, builds, and deployments are stored in a designated directory for easy access and management.

Backup and Recovery: Directories for storing backups and recovery data are designated for easy management and to ensure business continuity.

Environment-specific Directories: Directories are designated for each environment, such as development, test, and production, to ensure that the correct configuration files and artifacts are used for each environment.

By following a well-designed file system layout in a DevOps environment, you can improve the efficiency, reliability, and security of your software development and deployment processes.

The host server

In Jenkins, a host server refers to the physical or virtual machine that runs the Jenkins automation server. The host server is responsible for running the Jenkins process and providing resources, such as memory, storage, and CPU, for executing builds and other tasks.

The host server can be either a standalone machine or part of a network or cloud-based infrastructure. When running Jenkins on a standalone machine, the host server is responsible for all aspects of the Jenkins installation, including setup, configuration, and maintenance.

When running Jenkins on a network or cloud-based infrastructure, the host server is responsible for providing resources for the Jenkins process, but the setup, configuration, and maintenance may be managed by other components of the infrastructure.

By providing the necessary resources and ensuring the stability and reliability of the host server, you can ensure the efficient operation of Jenkins and the success of your software development and deployment processes.

To host a server in Jenkins, you'll need to follow these steps:

Install Jenkins: You can install Jenkins on a server by downloading the Jenkins WAR file, deploying it to a servlet container such as Apache Tomcat, and starting the server.

Configure Jenkins: Once Jenkins is up and running, you can access its web interface to configure and manage the build environment. You can install plugins, set up security, and configure build jobs.

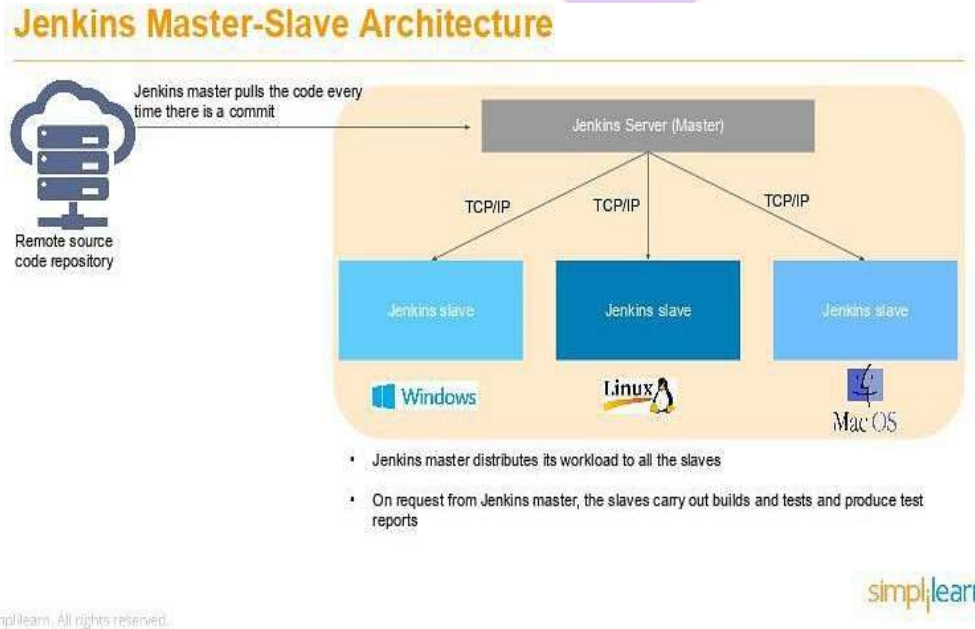
Create a Build Job: To build your project, you'll need to create a build job in Jenkins. This will define the steps involved in building your project, such as checking out the code from version control, compiling the code, running tests, and packaging the application.

Schedule Builds: You can configure your build job to run automatically at a specific time or when certain conditions are met. You can also trigger builds manually from the web interface.

Monitor Builds: Jenkins provides a variety of tools for monitoring builds, such as build history, build console output, and build artifacts. You can use these tools to keep track of the status of your builds and to diagnose problems when they occur.

Build slaves

Jenkins Master-Slave Architecture



As you can see in the diagram provided above, on the left is the Remote source code repository. The Jenkins server accesses the master environment on the left side and the master environment can push down to multiple other Jenkins Slave environments to distribute the workload.

That lets you run multiple builds, tests, and product environment across the entire architecture. Jenkins Slaves can be running different build versions of the code for different operating systems and the server Master controls how each of the builds operates.

Supported on a master-slave architecture, Jenkins comprises many slaves working for a master. This architecture - the Jenkins Distributed Build - can run identical test cases in different environments. Results are collected and combined on the master node for monitoring.

The standard Jenkins installation includes Jenkins master, and in this setup, the master will be managing all our build system's tasks. If we're working on a number of projects, we can run numerous jobs on each one. Some projects require the use of specific nodes, which necessitates the use of slave nodes.

The Jenkins master is in charge of scheduling jobs, assigning slave nodes, and sending builds to slave nodes for execution. It will also keep track of the slave node state (offline or online), retrieve build results from slave nodes, and display them on the terminal output. In most installations, multiple slave nodes will be assigned to the task of building jobs.

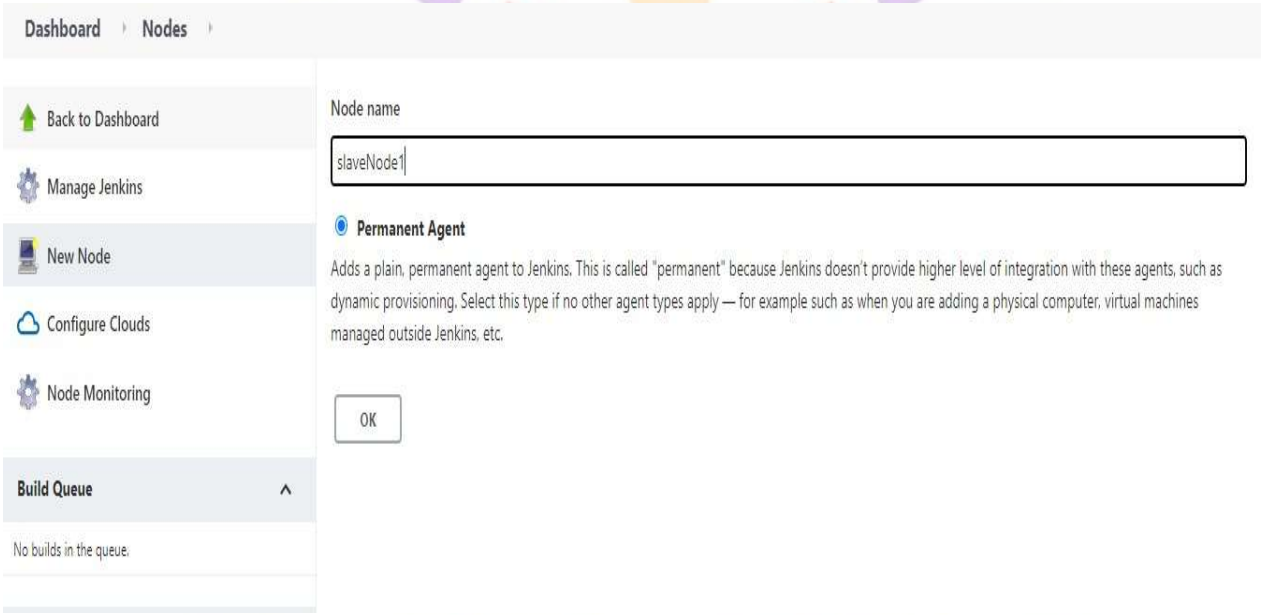
Before we get started, **let's double-check that we have all of the prerequisites in place for adding a slave node**:

- **Jenkins Server** is up and running and ready to use
- Another server for a slave node configuration
- The Jenkins server and the slave server are both connected to the same network

To configure the Master server, we'll log in to the Jenkins server and follow the steps below.

First, we'll go to **"Manage Jenkins -> Manage Nodes -> New Node" to create a new node**:

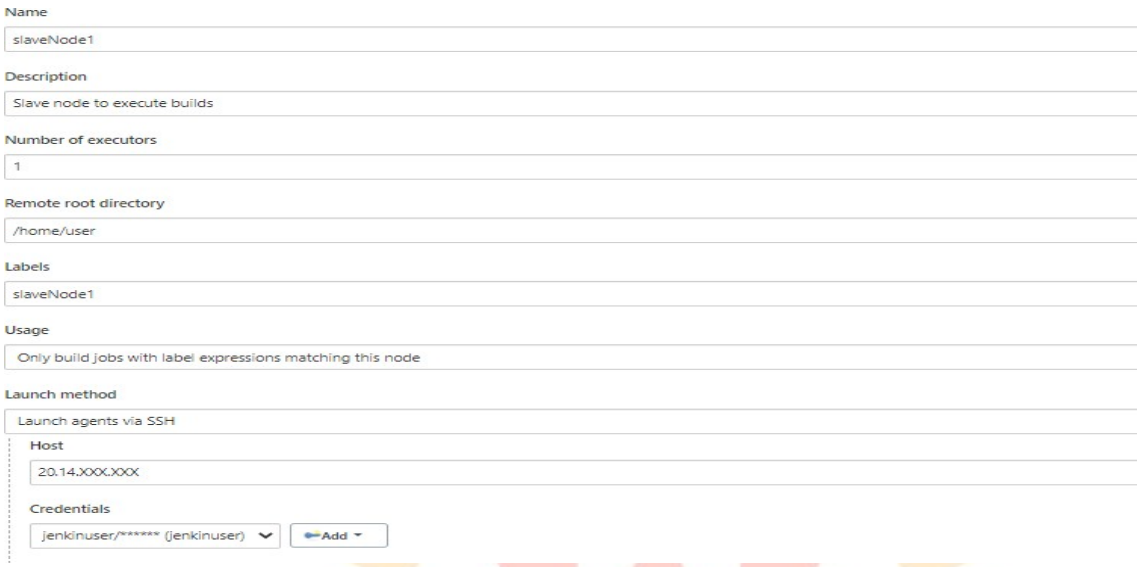
On the next screen, we **enter the "Node Name" (slaveNode1), select "Permanent Agent", then click "OK"**:



After clicking "OK", **we'll be taken to a screen with a new form where we need to fill out the slave node's information**. We're considering the slave node to be running on Linux operating systems, hence the launch method is set to "Launch agents via ssh".

In the same way, we'll add relevant details, such as the name, description, and a number of executors.

We'll save our work by pressing the "Save" button. The "Labels" with the name "slaveNode1" will help us to set up jobs on this slave node:



4. Building the Project on Slave Nodes

Now that our master and slave nodes are ready, we'll discuss the steps for building the project on the slave node.

For this, we start by clicking “New Item” in the top left corner of the dashboard.

Next, we need to enter the name of our project in the “Enter an item name” field and select the “Pipeline project”, and then click the “OK” button.

On the next screen, we'll enter a “Description” (optional) and navigate to the “Pipeline” section. Make sure the “Definition” field has the Pipeline script option selected.

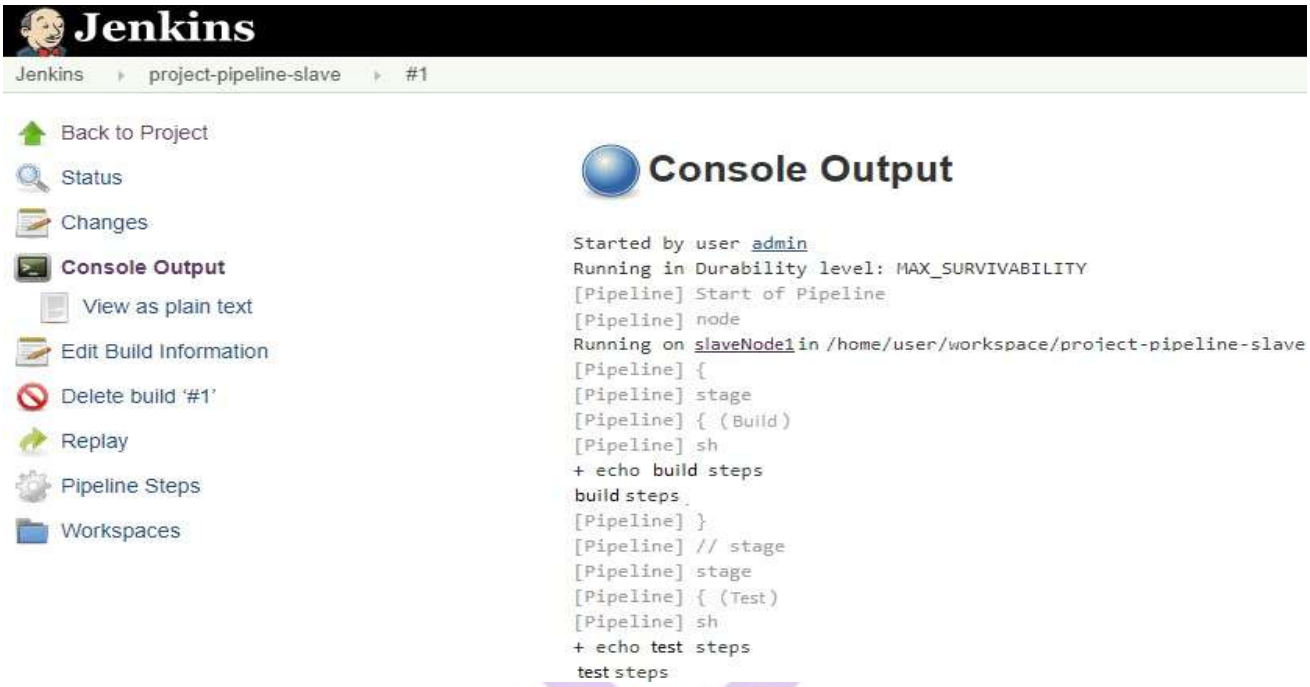
After this, we copy and paste the following declarative Pipeline script into a “script” field:

```
node('slaveNode1'){
   stage('Build') {
      sh '"echo build steps"'
   }
   stage('Test') {
      sh '"echo test steps"'
   }
}
```

Copy

Next, we click on the “Save” button. This will redirect to the Pipeline view page.

On the left pane, we click the "Build Now" button to execute our Pipeline. After Pipeline execution is completed, we'll see the Pipeline view:



We can verify the history of the executed build under the Build History by clicking the build number. As shown above, when we click on the build number and select "Console Output", we can see that the pipeline ran on our *slaveNode1* machine.

Software on the host

To run software on the host in Jenkins, you need to have the necessary dependencies and tools installed on the host machine. The exact software you'll need will depend on the specific requirements of your project and build process. Some common tools and software used in Jenkins include:

Java: Jenkins is written in Java and requires Java to be installed on the host machine.

Git: If your project uses Git as the version control system, you'll need to have Git installed on the host machine.

Build Tools: Depending on the programming language and build process of your project, you may need to install build tools such as Maven, Gradle, or Ant.

Testing Tools: To run tests as part of your build process, you'll need to install any necessary testing tools, such as JUnit, TestNG, or Selenium.

Database Systems: If your project requires access to a database, you'll need to have the necessary database software installed on the host machine, such as MySQL, PostgreSQL, or Oracle.

Continuous Integration Plugins: To extend the functionality of Jenkins, you may need to install plugins that provide additional tools and features for continuous integration, such as the Jenkins GitHub plugin, Jenkins Pipeline plugin, or Jenkins Slack plugin.

To install these tools and software on the host machine, you can use a package manager such as apt or yum, or you can download and install the necessary software manually. You can also use a containerization tool such as Docker to run Jenkins and the necessary software in isolated containers, which can simplify the installation process and make it easier to manage the dependencies and tools needed for your build process.

Trigger

These are the most common Jenkins build triggers:

- Trigger builds remotely
- Build after other projects are built
- Build periodically
- GitHub hook trigger for GITScm polling
- Poll SCM

1. Trigger builds remotely :

If you want to trigger your project built from anywhere anytime then you should select **Trigger builds remotely** option from the build triggers.

You'll need to provide an authorization token in the form of a string so that only those who know it would be able to remotely trigger this project's builds. This provides the predefined URL to invoke this trigger remotely.

```
predefined URL to trigger build remotely:

JENKINS_URL/job/JobName/build?token=TOKEN_NAME

JENKINS_URL: the IP and PORT which the Jenkins server is running

TOKEN_NAME: You have provided while selecting this build trigger.

//Example:

http://e330c73d.ngrok.io/job/test/build?token=12345
```

Whenever you will hit this URL from anywhere you project build will start.

2. Build after other projects are built

If your project depends on another project build then you should select **Build after other projects are built** option from the build triggers.

In this, you must specify the project(Job) names in the **Projects to watch** field section and select one of the following options:

1. Trigger only if the build is Note: A **build** is stable if it was **built** successfully and no publisher reports it as **unstable**
2. Trigger even if the build is u Note: A **build** is **unstable** if it was **built** successfully and one or more publishers report it **unstable**
3. Trigger even if the build fails

After that, It starts watching the specified projects in the **Projects to watch** section.

Whenever the build of the specified project completes (either is stable, unstable or failed according to your selected option) then this project build invokes.

3)Build periodically:

If you want to schedule your project build periodically then you should select the **Build periodically** option from the build triggers.

You must specify the periodical duration of the project build in the scheduler field section

This field follows the syntax of cron (with minor differences). Specifically, each line consists of 5 fields separated by TAB or whitespace:

```
MINUTE HOUR DOM MONTH DOW
```

MINUTE	Minutes within the hour (0–59)

HOUR	The hour of the day (0–23)
DOM	The day of the month (1–31)
MONTH	The month (1–12)
DOW	The day of the week (0–7) where 0 and 7 are Sunday.

To specify multiple values for one field, the following operators are available. In the order of precedence,

- * specifies all valid values
- M-N specifies a range of values
- M-N/X or */X steps by intervals of X through the specified range or whole valid range
- A,B,...,Z enumerates multiple values

Examples:

```
# every fifteen minutes (perhaps at :07, :22, :37, :52)
 H/15 * * * *
# every ten minutes in the first half of every hour (three times, perhaps at :04, :14, :24)
 H(0-29)/10 * * * *
# once every two hours at 45 minutes past the hour starting at 9:45 AM and finishing at 3:45 PM every weekday.
45 9-16/2 * * 1-5
# once in every two hours slot between 9 AM and 5 PM every weekday (perhaps at 10:38 AM, 12:38 PM, 2:38 PM, 4:38 PM)
H H(9-16)/2 * * 1-5
# once a day on the 1st and 15th of every month except December
H H 1,15 1-11 *
```

After successfully scheduled the project build then the scheduler will invoke the build periodically according to your specified duration.

4)GitHub webhook trigger for GITScm polling:

A webhook is an HTTP callback, an HTTP POST that occurs when something happens through a simple event-notification via HTTP POST.

GitHub webhooks in Jenkins are used to trigger the build whenever a developer commits something to the branch.

Let's see how to add build a webhook in GitHub and then add this webhook in Jenkins.

1. Go to your project repository.
2. Go to “settings” in the right corner.
3. Click on “webhooks.”
4. Click “Add webhooks.”
5. Write the Payload URL as

```
http://e330c73d.ngrok.io/github-webhook

//This URL is a public URL where the Jenkins server is running
```

Here https://e330c73d.ngrok.io/ is the IP and port where my Jenkins is running.

If you are running Jenkins on localhost then writing https://localhost:8080/github-webhook/ will not work because Webhooks can only work with the public IP.

So if you want to make your localhost:8080 expose public then we can use some tools.

In this example, we used ngrok tool to expose my local address to the public.

To know more on how to add webhook in Jenkins pipeline, visit: https://blog.knoldus.com/opsinit-adding-a-github-webhook-in-jenkins-pipeline/

5) Poll SCM:

Poll SCM periodically polls the SCM to check whether changes were made (i.e. new commits) and builds the project if new commits were pushed since the last build.

You must schedule the polling duration in the scheduler field. Like we explained above in the Build periodically section. You can see the Build periodically section to know how to schedule.

After successfully scheduled, the scheduler polls the SCM according to your specified duration in scheduler field and builds the project if new commits were pushed since the last build.LET'S INITIATE A PARTNERSHIP

Job chaining

Job chaining in Jenkins refers to the process of linking multiple build jobs together in a sequence. When one job completes, the next job in the sequence is automatically triggered. This allows you to create a pipeline of builds that are dependent on each other, so you can automate the entire build process.

There are several ways to chain jobs in Jenkins:

Build Trigger: You can use the build trigger in Jenkins to start one job after another. This is done by configuring the upstream job to trigger the downstream job when it completes.

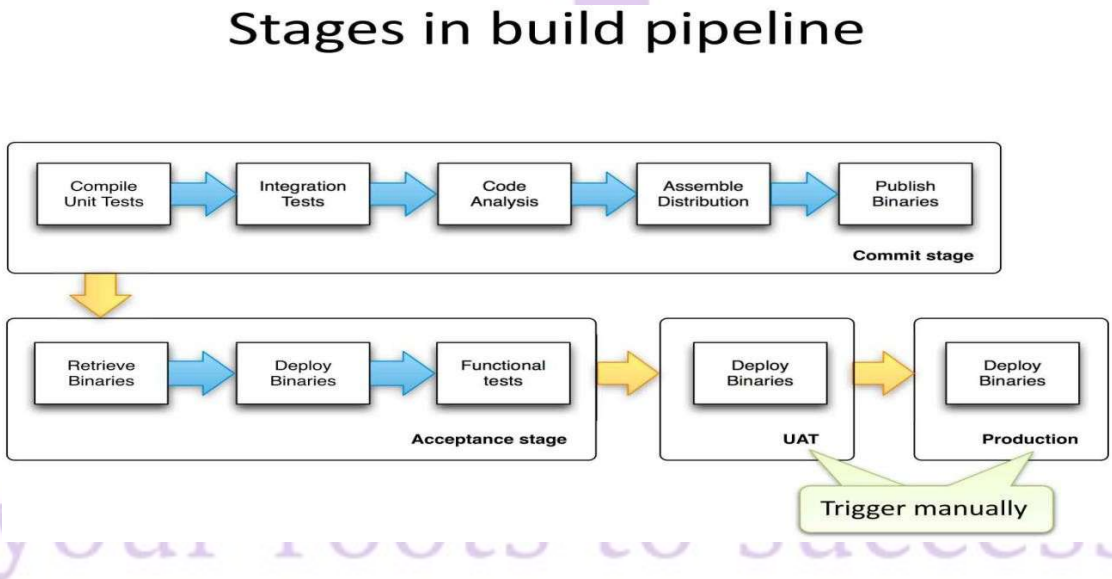
Jenkinsfile: If you are using Jenkins Pipeline, you can write a Jenkinsfile to define the steps in your build pipeline. The Jenkinsfile can contain multiple stages, each of which represents a separate build job in the pipeline.

JobDSL plugin: The JobDSL plugin allows you to programmatically create and manage Jenkins jobs. You can use this plugin to create a series of jobs that are linked together and run in sequence.

Multi-Job plugin: The Multi-Job plugin allows you to create a single job that runs multiple build steps, each of which can be a separate build job. This plugin is useful if you have a build pipeline that requires multiple build jobs to be run in parallel.

By chaining jobs in Jenkins, you can automate the entire build process and ensure that each step is completed before the next step is started. This can help to improve the efficiency and reliability of your build process, and allow you to quickly and easily make changes to your build pipeline.

Build pipelines



A build pipeline in DevOps is a set of automated processes that compile, build, and test software, and prepare it for deployment. A build pipeline represents the end-to-end flow of code changes from development to production.

The steps involved in a typical build pipeline include:

Code Commit: Developers commit code changes to a version control system such as Git.

Build and Compile: The code is built and compiled, and any necessary dependencies are resolved.

Unit Testing: Automated unit tests are run to validate the code changes.

Integration Testing: Automated integration tests are run to validate that the code integrates correctly with other parts of the system.

Staging: The code is deployed to a staging environment for further testing and validation.

Release: If the code passes all tests, it is deployed to the production environment.

Monitoring: The deployed code is monitored for performance and stability.

A build pipeline can be managed using a continuous integration tool such as Jenkins, TravisCI, or CircleCI. These tools automate the build process, allowing you to quickly and easily make changes to the pipeline, and ensuring that the pipeline is consistent and reliable.

In DevOps, the build pipeline is a critical component of the continuous delivery process, and is used to ensure that code changes are tested, validated, and deployed to production as quickly and efficiently as possible. By automating the build pipeline, you can reduce the time and effort required to deploy code changes, and improve the speed and quality of your software delivery process.

Build servers

When you're developing and deploying software, one of the first things to figure out is how to take your code and deploy your working application to a production environment where people can interact with your software.

Most development teams understand the importance of version control to coordinate code commits, and build servers to compile and package their software, but Continuous Integration (CI) is a big topic.

Why build servers are important

Build servers have 3 main purposes:

- Compiling committed code from your repository many times a day
- Running automatic tests to validate code
- Creating deployable packages and handing off to a deployment tool, like Octopus Deploy

Without a build server you're slowed down by complicated, manual processes and the needless time constraints they introduce. For example, without a build server:

- Your team will likely need to commit code before a daily deadline or during change windows
- After that deadline passes, no one can commit again until someone manually creates and tests a build
- If there are problems with the code, the deadlines and manual processes further delay the fixes

Without a build server, the team battles unnecessary hurdles that automation removes. A build server will repeat these tasks for you throughout the day, and without those human-caused delays.

But CI doesn't just mean less time spent on manual tasks or the death of arbitrary deadlines, either. By automatically taking these steps many times a day, you fix problems sooner and your results become more predictable. Build servers ultimately help you deploy through your pipeline with more confidence.

Building servers in DevOps involves several steps:

Requirements gathering: Determine the requirements for the server, such as hardware specifications, operating system, and software components needed.

Server provisioning: Choose a method for provisioning the server, such as physical installation, virtualization, or cloud computing.

Operating System installation: Install the chosen operating system on the server.

Software configuration: Install and configure the necessary software components, such as web servers, databases, and middleware.

Network configuration: Set up network connectivity, such as IP addresses, hostnames, and firewall rules.

Security configuration: Configure security measures, such as user authentication, access control, and encryption.

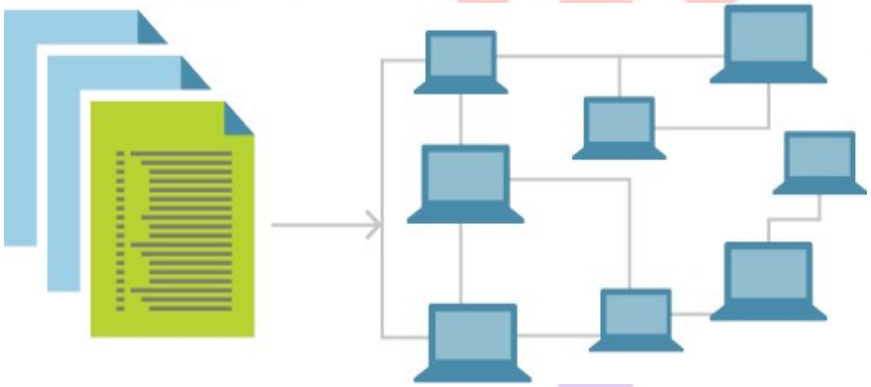
Monitoring and maintenance: Implement monitoring and maintenance processes, such as logging, backup, and disaster recovery.

Deployment: Deploy the application to the server and test it to ensure it is functioning as expected.

Throughout the process, it is important to automate as much as possible using tools such as Ansible, Chef, or Puppet to ensure consistency and efficiency in building servers.

Infrastructure as code

Infrastructure as code (IaC) uses DevOps methodology and versioning with a descriptive model to define and deploy infrastructure, such as networks, virtual machines, load balancers, and connection topologies. Just as the same source code always generates the same binary, an IaC model generates the same environment every time it deploys.



IaC is a key DevOps practice and a component of continuous delivery. With IaC, DevOps teams can work together with a unified set of practices and tools to deliver applications and their supporting infrastructure rapidly and reliably at scale.

IaC evolved to solve the problem of *environment drift* in release pipelines. Without IaC, teams must maintain deployment environment settings individually. Over time, each environment becomes a "snowflake," a unique configuration that can't be reproduced automatically. Inconsistency among environments can cause deployment issues. Infrastructure administration and maintenance involve manual processes that are error prone and hard to track.

IaC avoids manual configuration and enforces consistency by representing desired environment states via well-documented code in formats such as JSON. Infrastructure deployments with IaC are repeatable and prevent runtime issues caused by configuration drift or missing dependencies.

Release pipelines execute the environment descriptions and version configuration models to configure target environments. To make changes, the team edits the source, not the target.

Idempotence, the ability of a given operation to always produce the same result, is an important IaC principle. A deployment command always sets the target environment into the same configuration, regardless of the environment's starting state. Idempotency is achieved by either automatically configuring the existing target, or by discarding the existing target and recreating a fresh environment.

IAC can be achieved by using tools such as Terraform, CloudFormation, or Ansible to define infrastructure components in a file that can be versioned, tested, and deployed in a consistent and automated manner.

Benefits of IAC include:

Speed: IAC enables quick and efficient provisioning and deployment of infrastructure.

Consistency: By using code to define and manage infrastructure, it is easier to ensure consistency across multiple environments.

Repeatability: IAC allows for easy replication of infrastructure components in different environments, such as development, testing, and production.

Scalability: IAC makes it easier to scale infrastructure as needed by simply modifying the code.

Version control: Infrastructure components can be versioned, allowing for rollback to previous versions if necessary.

Overall, IAC is a key component of modern DevOps practices, enabling organizations to manage their infrastructure in a more efficient, reliable, and scalable way.

Building by dependency order

Building by dependency order in DevOps is the process of ensuring that the components of a system are built and deployed in the correct sequence, based on their dependencies. This is necessary to ensure that the system functions as intended, and that components are deployed in the right order so that they can interact correctly with each other.

The steps involved in building by dependency order in DevOps include:

Define dependencies: Identify all the components of the system and the dependencies between them. This can be represented in a diagram or as a list.

Determine the build order: Based on the dependencies, determine the correct order in which components should be built and deployed.

Automate the build process: Use tools such as Jenkins, TravisCI, or CircleCI to automate the build and deployment process. This allows for consistency and repeatability in the build process.

Monitor progress: Monitor the progress of the build and deployment process to ensure that components are deployed in the correct order and that the system is functioning as expected.

Test and validate: Test the system after deployment to ensure that all components are functioning as intended and that dependencies are resolved correctly.

Rollback: If necessary, have a rollback plan in place to revert to a previous version of the system if the build or deployment process fails.

In conclusion, building by dependency order in DevOps is a critical step in ensuring the success of a system deployment, as it ensures that components are deployed in the correct order and that dependencies are resolved correctly. This results in a more stable, reliable, and consistent system.

Build phases

In DevOps, there are several phases in the build process, including:

Planning: Define the project requirements, identify the dependencies, and create a build plan.

Code development: Write the code and implement features, fixing bugs along the way.

Continuous Integration (CI): Automatically build and test the code as it is committed to a version control system.

Continuous Delivery (CD): Automatically deploy code changes to a testing environment, where they can be tested and validated.

Deployment: Deploy the code changes to a production environment, after they have passed testing in a pre-production environment.

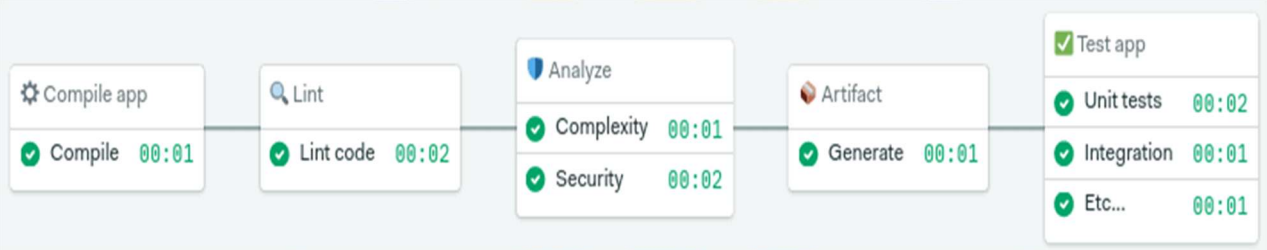
Monitoring: Continuously monitor the system to ensure that it is functioning as expected, and to detect and resolve any issues that may arise.

Maintenance: Continuously maintain and update the system, fixing bugs, adding new features, and ensuring its stability.

These phases help to ensure that the build process is efficient, reliable, and consistent, and that code changes are validated and deployed in a controlled manner. Automation is a key aspect of DevOps, and it helps to make these phases more efficient and less prone to human error.

In continuous integration (CI), this is where we build the application for the first time. The build stage is the first stretch of a CI/CD pipeline, and it automates steps like downloading dependencies, installing tools, and compiling.

Besides building code, build automation includes using tools to check that the code is safe and follows best practices. The build stage usually ends in the artifact generation step, where we create a production-ready package. Once this is done, the testing stage can begin.



The build stage starts from code commit and runs from the beginning up to the test stage

We'll be covering testing in-depth in future articles (subscribe to the newsletter so you don't miss them). Today, we'll focus on build automation.

Build automation verifies that the application, at a given code commit, can qualify for further testing. We can divide it into three parts:

1. **Compilation**: the first step builds the application.
2. **Linting**: checks the code for programmatic and stylistic errors.
3. **Code analysis**: using automated source-checking tools, we control the code's quality.
4. **Artifact generation**: the last step packages the application for release or deployment.

Alternative build servers

There are several alternative build servers in DevOps, including:

Jenkins - an open-source, Java-based automation server that supports various plugins and integrations.

Travis CI - a cloud-based, open-source CI/CD platform that integrates with Github.

CircleCI - a cloud-based, continuous integration and delivery platform that supports multiple languages and integrates with several platforms.

GitLab CI/CD - an integrated CI/CD solution within GitLab that allows for complete project and pipeline management.

Bitbucket Pipelines - a CI/CD solution within Bitbucket that allows for pipeline creation and management within the code repository.

AWS CodeBuild - a fully managed build service that compiles source code, runs tests, and produces software packages that are ready to deploy.

Azure Pipelines - a CI/CD solution within Microsoft Azure that supports multiple platforms and programming languages.

Collating quality measures

In DevOps, collating quality measures is an important part of the continuous improvement process. The following are some common quality measures used in DevOps to evaluate the quality of software systems:

Continuous Integration (CI) metrics - metrics that track the success rate of automated builds and tests, such as build duration and test pass rate.

Continuous Deployment (CD) metrics - metrics that track the success rate of deployments, such as deployment frequency and time to deployment.

Code review metrics - metrics that track the effectiveness of code reviews, such as review completion time and code review feedback.

Performance metrics - measures of system performance in production, such as response time and resource utilization.

User experience metrics - measures of how users interact with the system, such as click-through rate and error rate.

Security metrics - measures of the security of the system, such as the number of security vulnerabilities and the frequency of security updates.

Incident response metrics - metrics that track the effectiveness of incident response, such as mean time to resolution (MTTR) and incident frequency.

By regularly collating these quality measures, DevOps teams can identify areas for improvement, track progress over time, and make informed decisions about the quality of their systems.