Infrastructure Automation Tools

TEKsystems Global Services

WORKFORCE DEVELOPMENT

CI/CD and Infra Automation

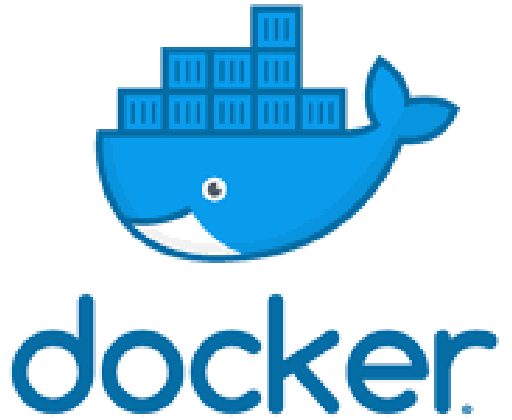
Learning Objectives

Day 2 Objectives

- Build and push Docker images using Jenkins pipelines
- Manually deploy a containerized application on a virtual machine
- Automate deployment using Ansible with an inventory file and playbook

Jenkins

Automation tools



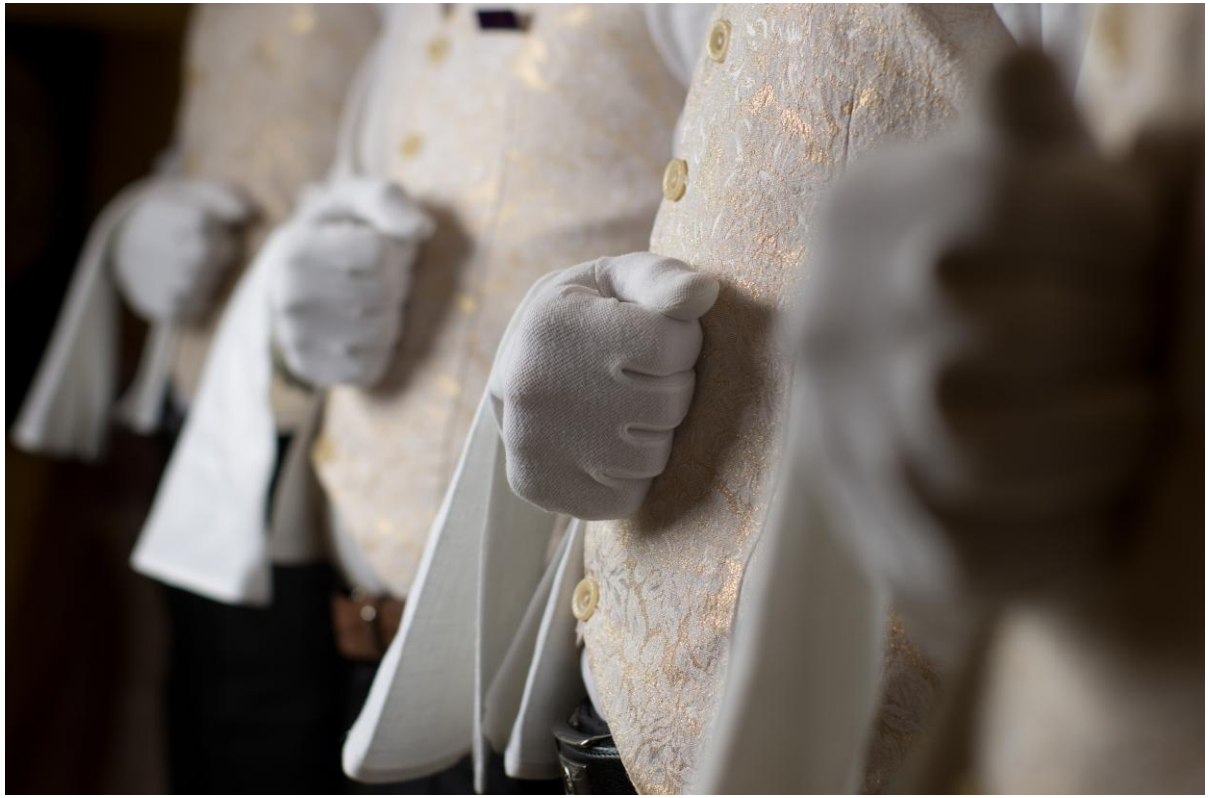
Jenkins

Jenkins is an open-source automation server used to build, test, and deploy software. It runs pipelines that allow teams to automate repetitive tasks and enforce consistent workflows.

Key points

- One of the most widely used CI/CD tools
- Uses jobs and pipelines to automate builds and deployments
- Highly extensible through plugins
- Can run on your own servers for full control

Jenkins Controller and Agents



Jenkins uses a controller–agent architecture to run pipelines efficiently.

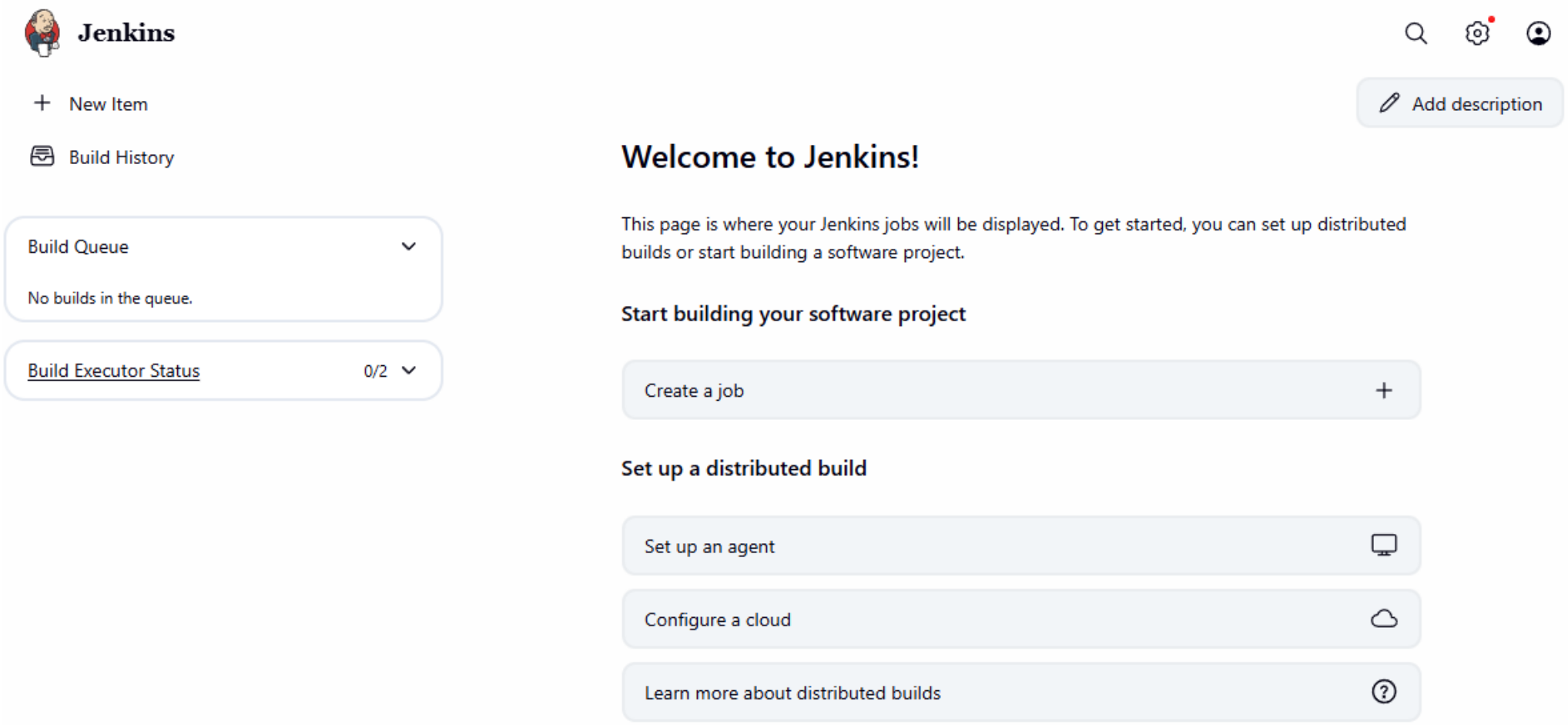
- **Controller** manages Jenkins: schedules jobs, provides the UI, stores configurations
- **Agents** are worker machines that execute the actual build and deployment steps
- Agents can be Linux, Windows, or containers depending on your needs
- This model allows Jenkins to scale and run multiple jobs in parallel

Jenkins Controller and Agents

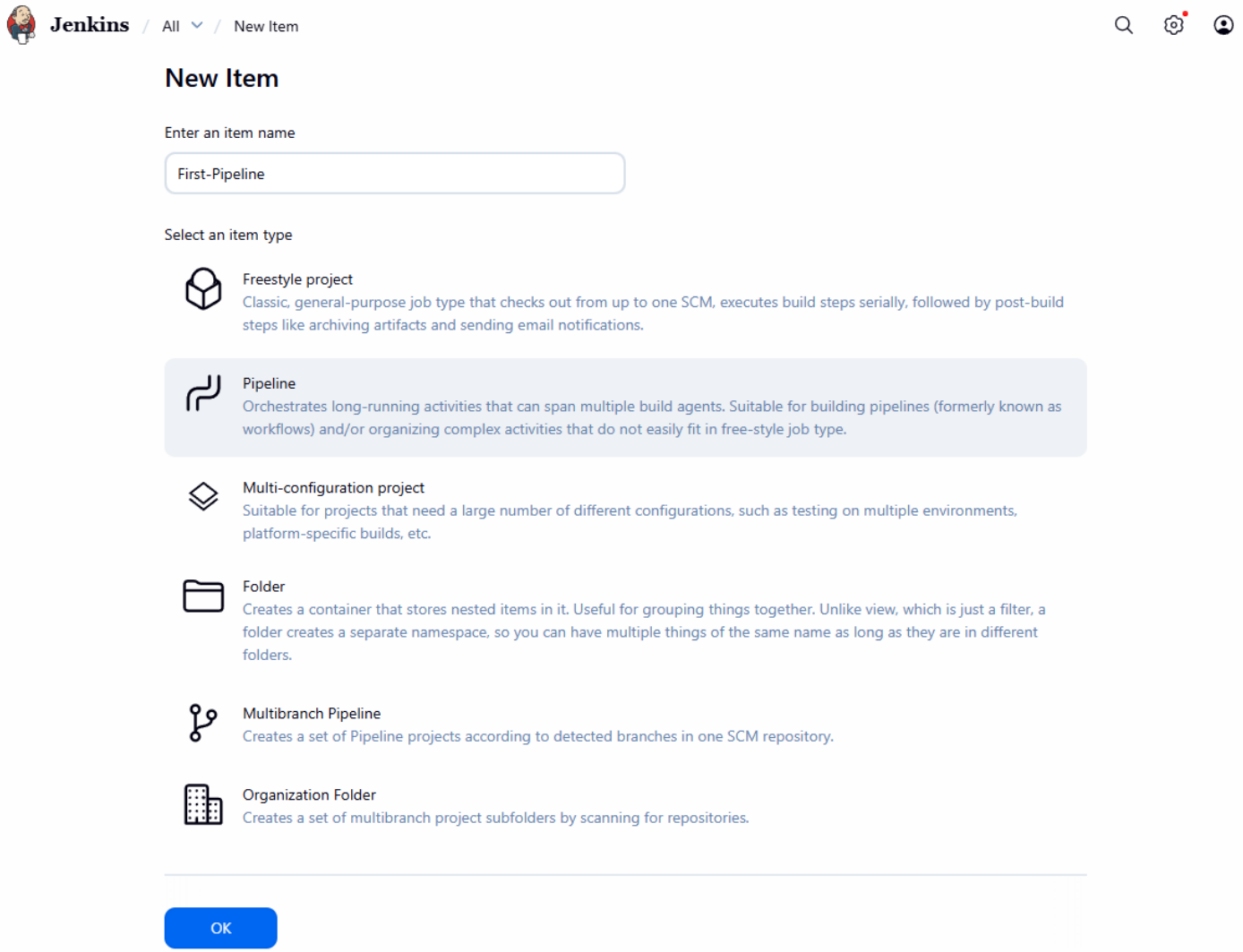
The Jenkins controller is a persistent server that stores your pipelines, settings, and job history. It provides the user interface and decides when jobs should run, giving you a stable place to manage and track all automation.

When a pipeline runs, the controller spins up an agent to execute the steps. Agents are ephemeral and disappear once the job is finished, but all logs, results, and pipeline definitions remain on the controller. This keeps builds clean while preserving everything you need.

Jenkins Welcome Page



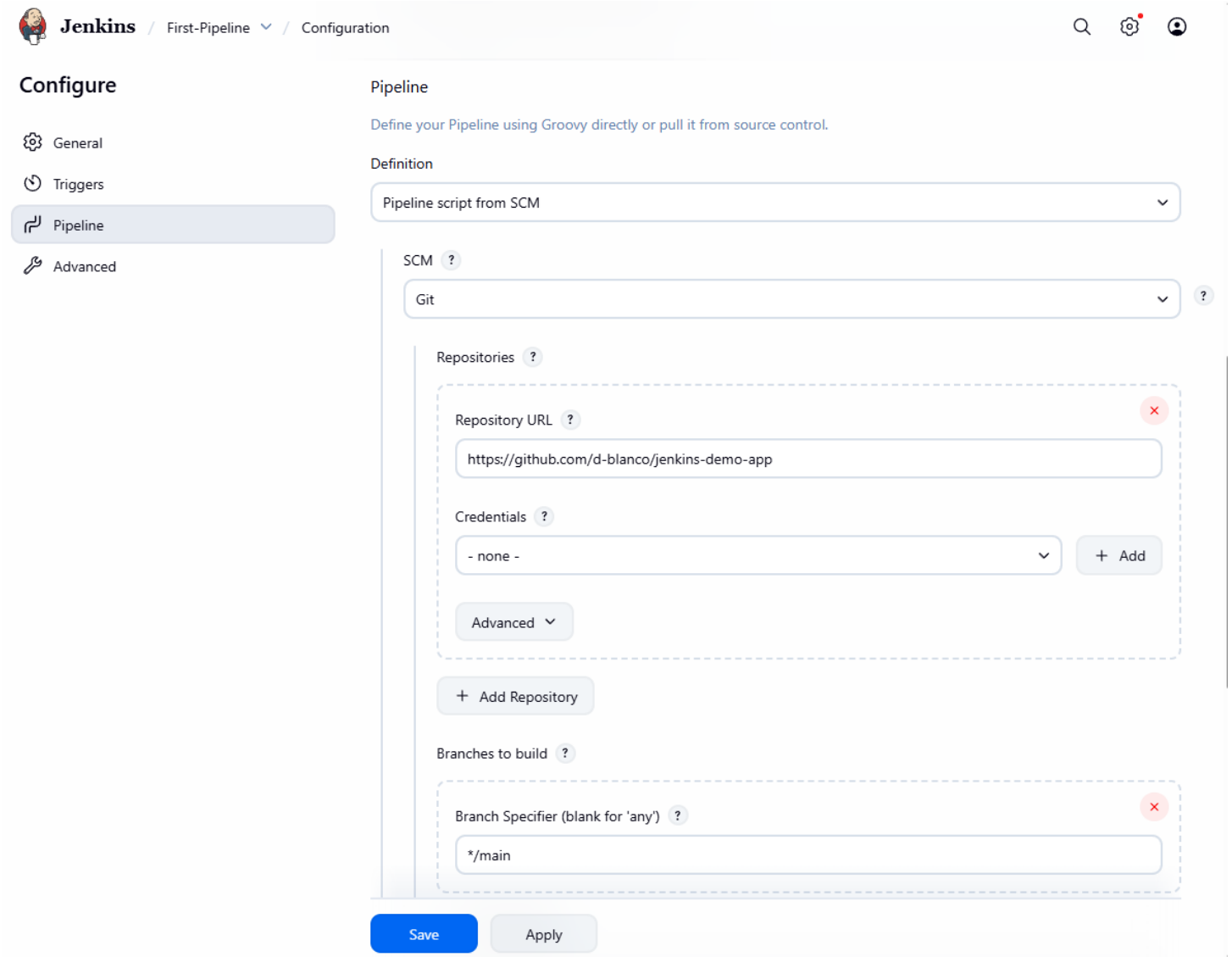
Jenkins Create Pipeline



Jenkins supports many item types, such as Freestyle jobs, Multi-branch pipelines, and traditional projects.
For CI/CD, the most common and recommended option is **Pipeline**, which lets you define your automation using code.

Another popular choice is **Multibranch Pipeline**, which automatically creates pipelines for each branch in a repository.

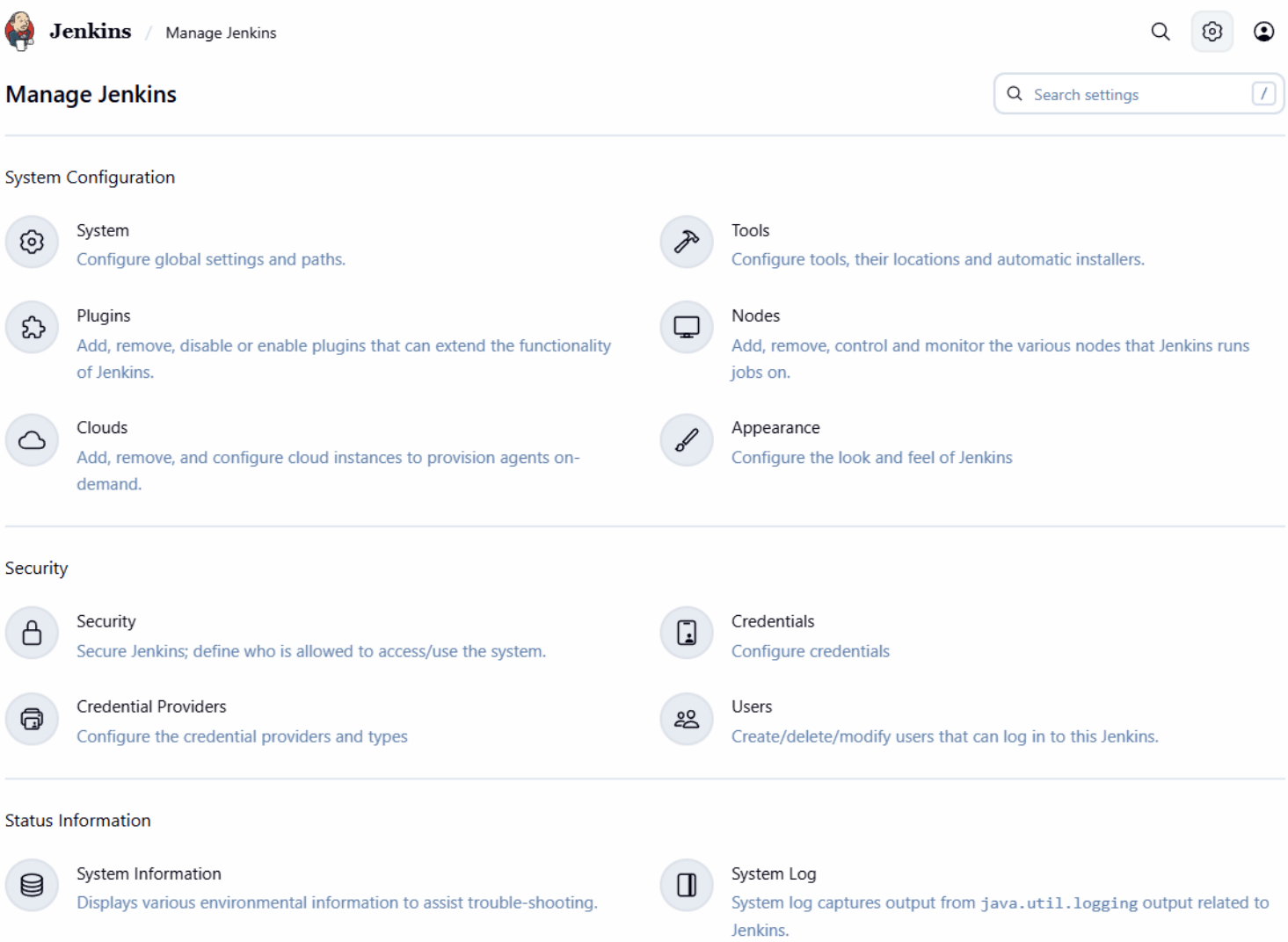
Jenkins SCM for Pipeline



A pipeline has many configuration options, including scheduling, webhook triggers, and build behavior.

The most important section here is **SCM (Source Control Management)**, which lets Jenkins pull your pipeline code directly from GitHub. Storing pipeline code in version control keeps it backed up, reviewable, and versioned instead of relying on Jenkins' internal storage.

Jenkins SCM for Pipeline

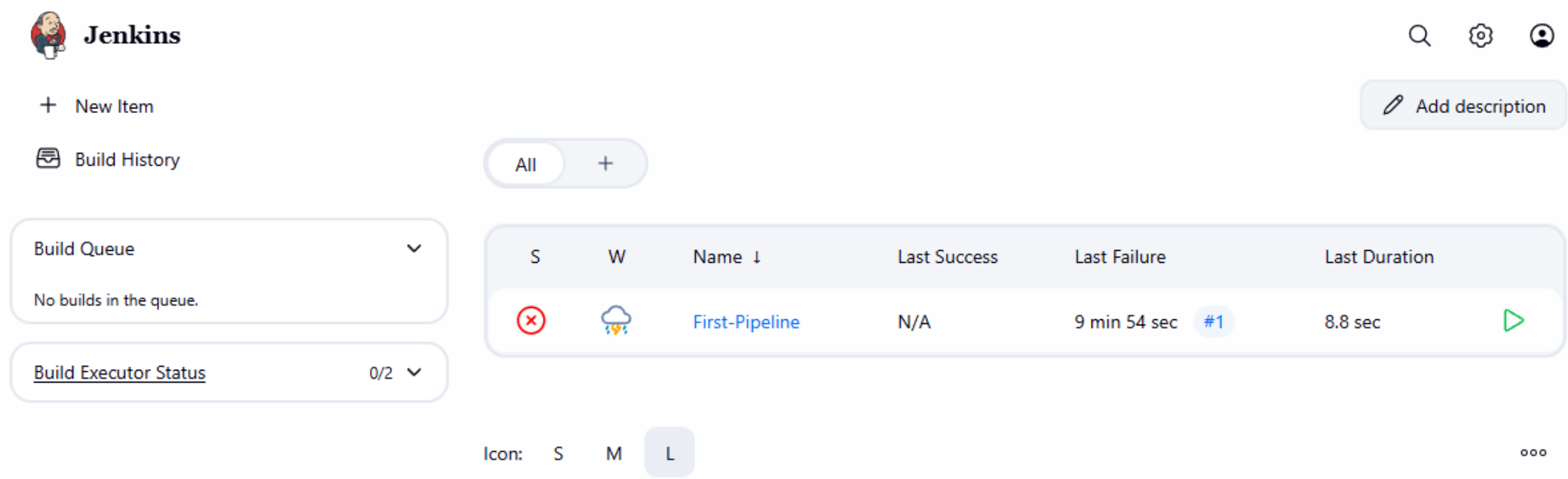


Jenkins can be extended through **plugins**, which add new features, integrations, and tools to the platform. The general settings page allows you to install and manage these plugins. Common plugins include Git integration, Docker pipeline steps, and credentials management.

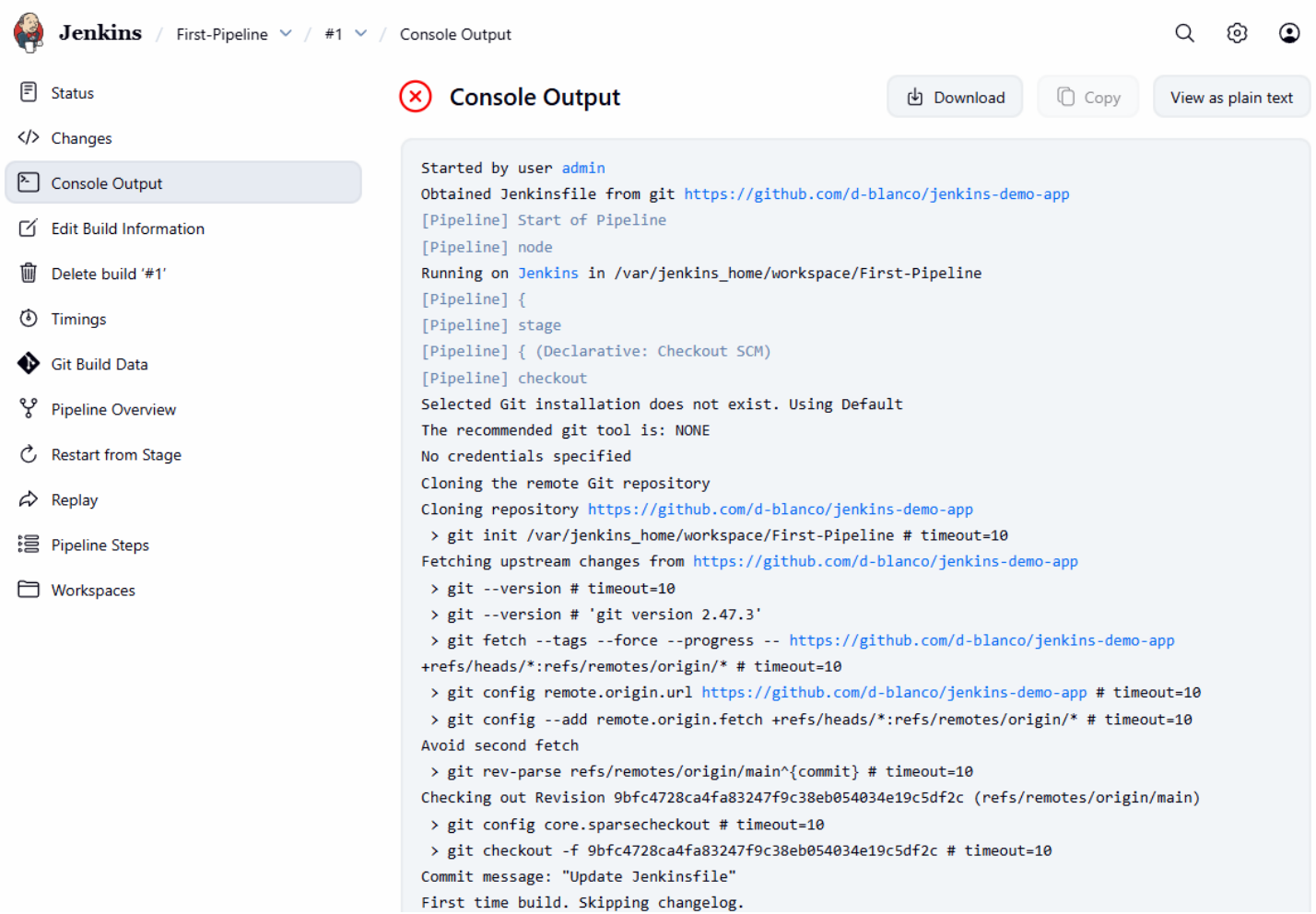
This page also provides access to **secrets and secure credentials**, which Jenkins stores safely and allows pipelines to reference without exposing them in logs or scripts.

Jenkins SCM for Pipeline

The home page shows all pipelines and their current or last build status. From here, you can quickly view results, trigger new runs, or open a pipeline's details.



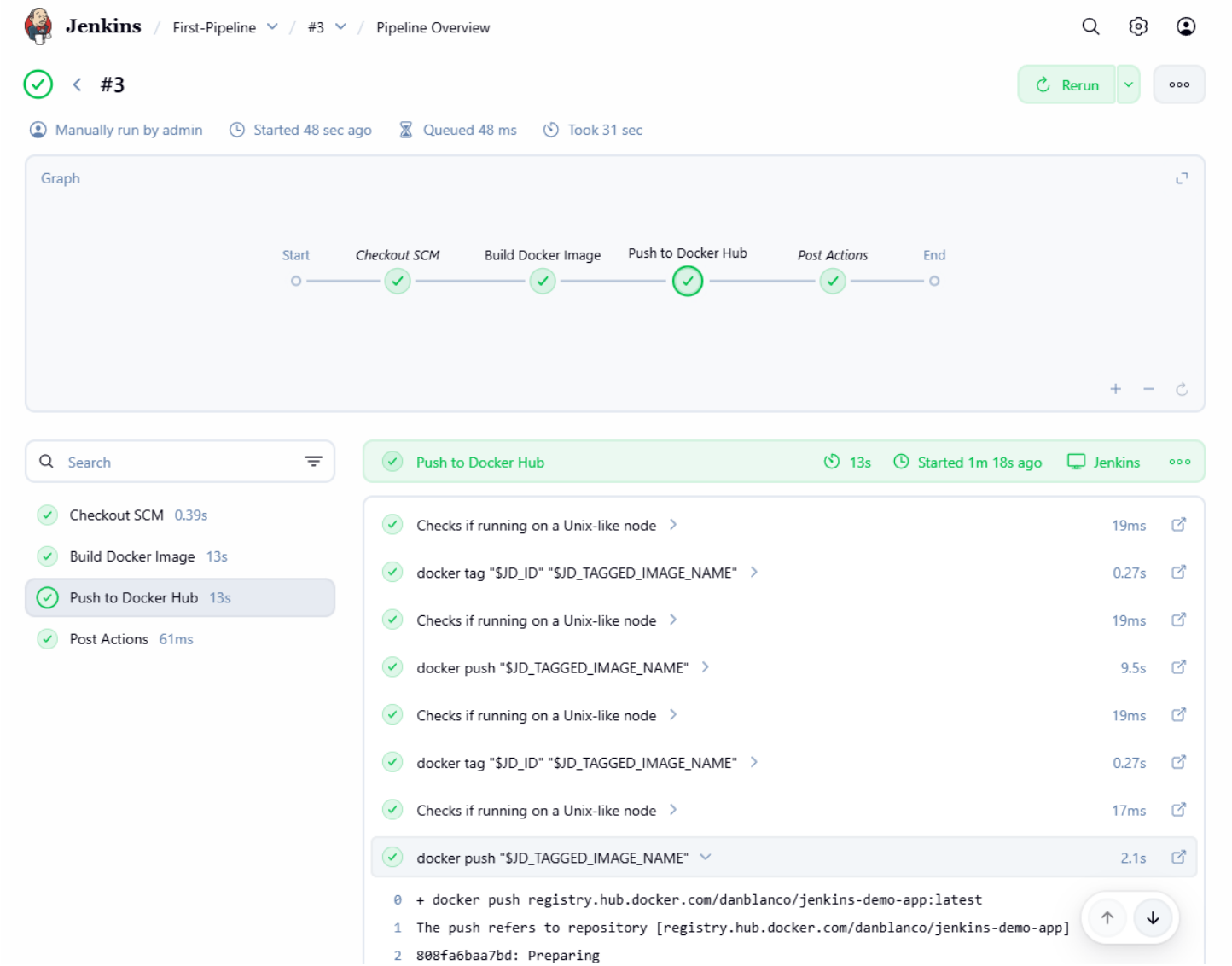
Jenkins SCM for Pipeline



The console output page shows the logs for a **specific run** of a pipeline, letting you see exactly what happened during that execution. It is the main place troubleshoot any errors.

These logs are generated on the **agent** that runs the job but are saved in Jenkins so they remain available even after the agent disappears. From this view, you can also access additional run details and settings related to that build.

Jenkins SCM for Pipeline



The build overview page shows the results of a specific pipeline run but organizes the output by **stages** instead of a single continuous log. This makes it easier to see which stage succeeded, failed, or took the longest.

Each stage expands to show its own logs and details, giving you a clearer picture of where problems occurred without scrolling through the full console output.

Jenkins SCM for Pipeline

```
pipeline {
    agent any

    stages {
        stage('Hello') {
            steps {
                echo 'Hello World!'
            }
        }

        stage('Hello with Shell') {
            steps {
                sh 'echo "Hello from shell!"'
            }
        }
    }
}
```

In Jenkins, a **stage** represents a major section of the pipeline, similar to a step in GitHub Actions. It defines the high-level flow of the build, such as “Build,” “Test,” or “Deploy.”

Inside each stage, Jenkins uses **steps**, which are the individual actions that run. Steps can execute shell commands or use the Jenkins Pipeline **DSL (Domain Specific Language)**—a structured, readable syntax for defining pipeline behavior.

This example shows a simple pipeline with two stages and one step in each, demonstrating how Jenkins organizes and executes pipeline logic.

Jenkins SCM for Pipeline

```
pipeline {
    agent any

    stages {
        stage('Hello') {
            steps {
                echo 'Hello World!'
            }
        }

        stage('Hello with Shell') {
            steps {
                sh 'echo "Hello from shell!"'
            }
        }
    }
}
```

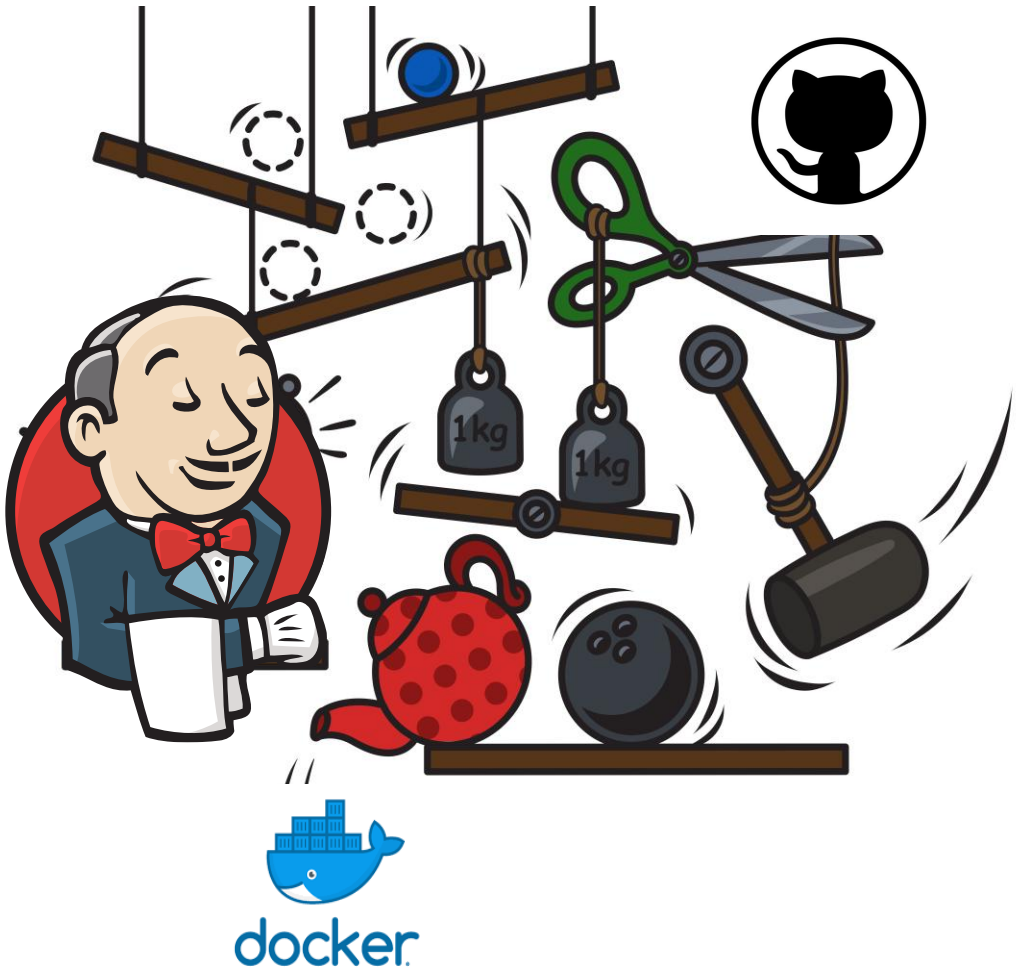
This example is a **Declarative Pipeline**, written in **Groovy**, which provides a structured and opinionated way to define stages, steps, and the overall pipeline flow. Declarative pipelines are the recommended modern approach because they are easier to read, validate, and maintain.

Jenkins also supports **Scripted Pipelines**, which are fully Groovy-based and give complete programmatic control. Scripted pipelines are more flexible but harder to write and understand, so they are used mostly for advanced automation.

Although Jenkins does not use YAML for pipelines like GitHub Actions, it does allow pipelines to be defined in multiple ways:

Jenkins

Jenkins



Tools like GitHub Actions, Jenkins, Docker, and others are often combined to move software from development to production. These tools can overlap or even replace each other, but teams often use several of them together to handle different responsibilities.

Common reasons pipelines become complex

- One tool handles builds, another handles deployments
- Security or approval workflows add extra layers
- Infrastructure teams and application teams use different systems
- Organizations evolve over time instead of starting from scratch

Real pipelines often look like a chain of interconnected tools, each performing a specific part of the release process.

POP QUIZ:

What is the primary role of the Jenkins controller?

A. To execute all pipeline steps directly
B. To store pipeline configurations and manage job scheduling
C. To automatically deploy applications to production
D. To run containers as part of CI/CD

POP QUIZ:

What is the primary role of the Jenkins controller?

A. To execute all pipeline steps directly
B. **To store pipeline configurations and manage job scheduling**
C. To automatically deploy applications to production
D. To run containers as part of CI/CD

POP QUIZ:

What is a Jenkins agent?

A. A temporary worker that runs pipeline steps and disappears afterward
B. A permanent machine that stores build logs
C. A Docker container that runs the Jenkins UI
D. A backup server for the controller

POP QUIZ:

What is a Jenkins agent?

A. **A temporary worker that runs pipeline steps and disappears afterward**
B. A permanent machine that stores build logs
C. A Docker container that runs the Jenkins UI
D. A backup server for the controller

POP QUIZ:

In Jenkins, what is a *stage* used for?

A. A single low-level command executed by the agent
B. Storing environment variables for the controller
C. Installing plugins during runtime
D. Grouping major sections of the pipeline such as Build, Test, or Deploy

POP QUIZ:

In Jenkins, what is a *stage* used for?

A. A single low-level command executed by the agent
B. Storing environment variables for the controller
C. Installing plugins during runtime
D. **Grouping major sections of the pipeline such as Build, Test, or Deploy**

POP QUIZ:

What does the Pipeline DSL (Domain Specific Language) do?

A. Automatically translates Jenkinsfiles into YAML
B. Provides structured syntax for defining stages and steps in a pipeline
C. Allows Jenkins to replace Dockerfiles
D. Encrypts Jenkins credentials

POP QUIZ:

What does the Pipeline DSL (Domain Specific Language) do?

A. Automatically translates Jenkinsfiles into YAML
B. **Provides structured syntax for defining stages and steps in a pipeline**
C. Allows Jenkins to replace Dockerfiles
D. Encrypts Jenkins credentials

POP QUIZ:

Where are Jenkins pipeline logs stored after a run completes?

A. On the ephemeral agent permanently
B. They are deleted immediately after the agent shuts down
C. Stored on the Jenkins controller and accessible in the Console Output
D. Saved only in Docker Hub

POP QUIZ:

Where are Jenkins pipeline logs stored after a run completes?

A. On the ephemeral agent permanently
B. They are deleted immediately after the agent shuts down
C. **Stored on the Jenkins controller and accessible in the Console Output**
D. Saved only in Docker Hub

POP QUIZ:

What is the main advantage of storing a Jenkinsfile in SCM (like GitHub)?

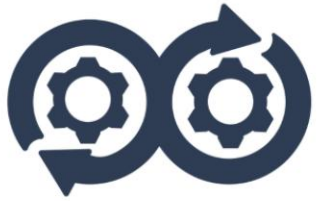
A. It allows Jenkins to run without a controller
B. Pipelines become versioned, backed up, and updated through code changes
C. Jenkins can automatically create EC2 instances
D. It removes the need for agents

POP QUIZ:

What is the main advantage of storing a Jenkinsfile in SCM (like GitHub)?

A. It allows Jenkins to run without a controller
B. **Pipelines become versioned, backed up, and updated through code changes**
C. Jenkins can automatically create EC2 instances
D. It removes the need for agents

Lab: Jenkins Docker



Manual Deployment

Manual Deployment

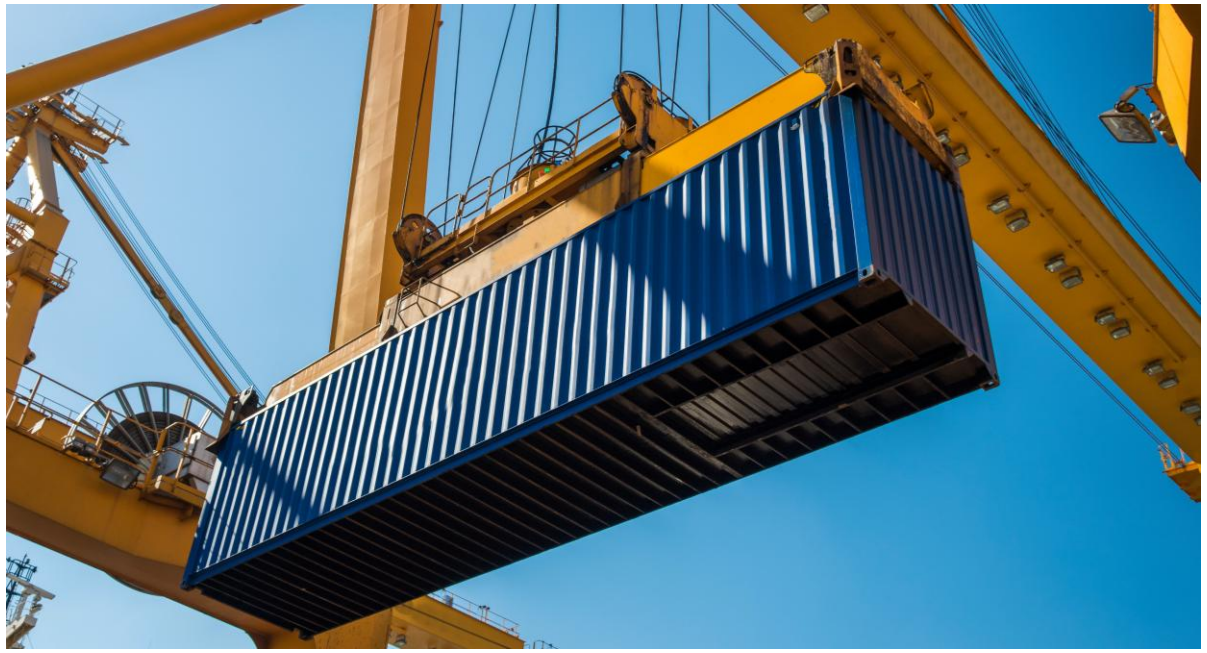
Manual Deployment

Manual to Automated

Before you automate a deployment, you must understand how to do it manually. Automation simply repeats the steps you would perform yourself, so if the manual process is unclear or incorrect, the automated version will fail as well.

Knowing the manual steps gives you insight into what must be installed, configured, started, and verified on the server. This ensures your automation tools can reproduce the deployment reliably and consistently.

Deployment Via Containers



Since Docker Hub is our delivery source for packaged images, we need a way to run those images on a server. Images run inside **containers**, and there are many platforms that can manage them, such as Kubernetes or AWS ECS.

For this simple example, we will use the **Docker Runtime** directly on a virtual machine. This allows us to pull the image, run it as a container, and clearly understand the manual steps before we automate the process.

Deployment Via Docker

Deployment Requirements

- A virtual machine we can SSH into
- A public IP address to access the application
- Firewall or security group rules open for the app's ports
- Docker installed and the Docker service running

Manual Deployment Steps

- Pull the image from Docker Hub
- Run the container with correct port bindings
- Provide any required environment variables or config values
- Optionally run the container in detached (daemon) mode

Automating Deployments

Before automating deployment, you should be able to manage the container manually. This includes viewing logs, stopping and starting the container, and testing the application directly on the VM.

Understanding how the container behaves in a manual workflow helps ensure that your automation will be accurate, reliable, and easier to troubleshoot.

Containers

- A container stops when its **main process** exits. Docker does not keep containers alive on its own.

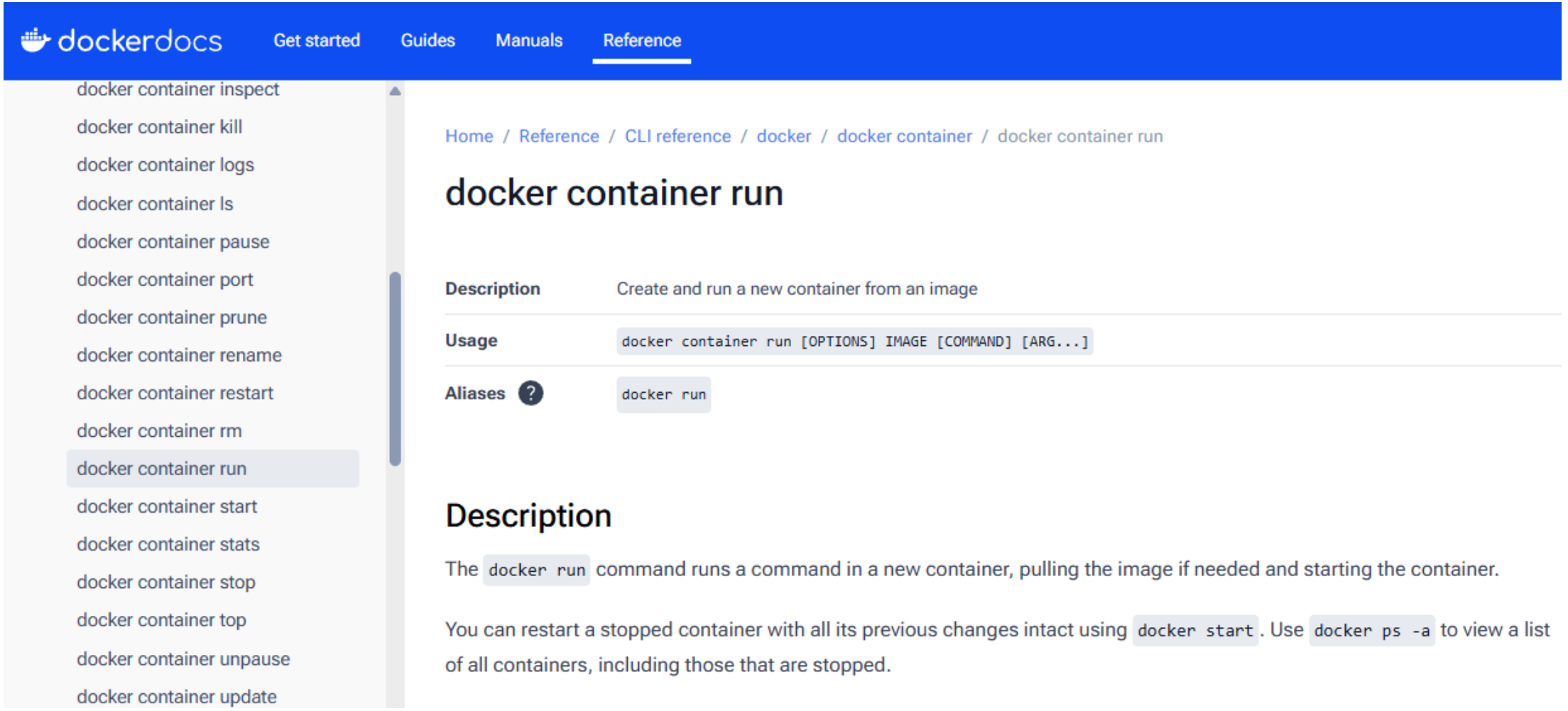
- Containers read **environment variables** at runtime, allowing the same image to run with different configurations in dev, test, or production.

- Images are **immutable**, but containers are not. You configure them through runtime settings, not by changing the image.

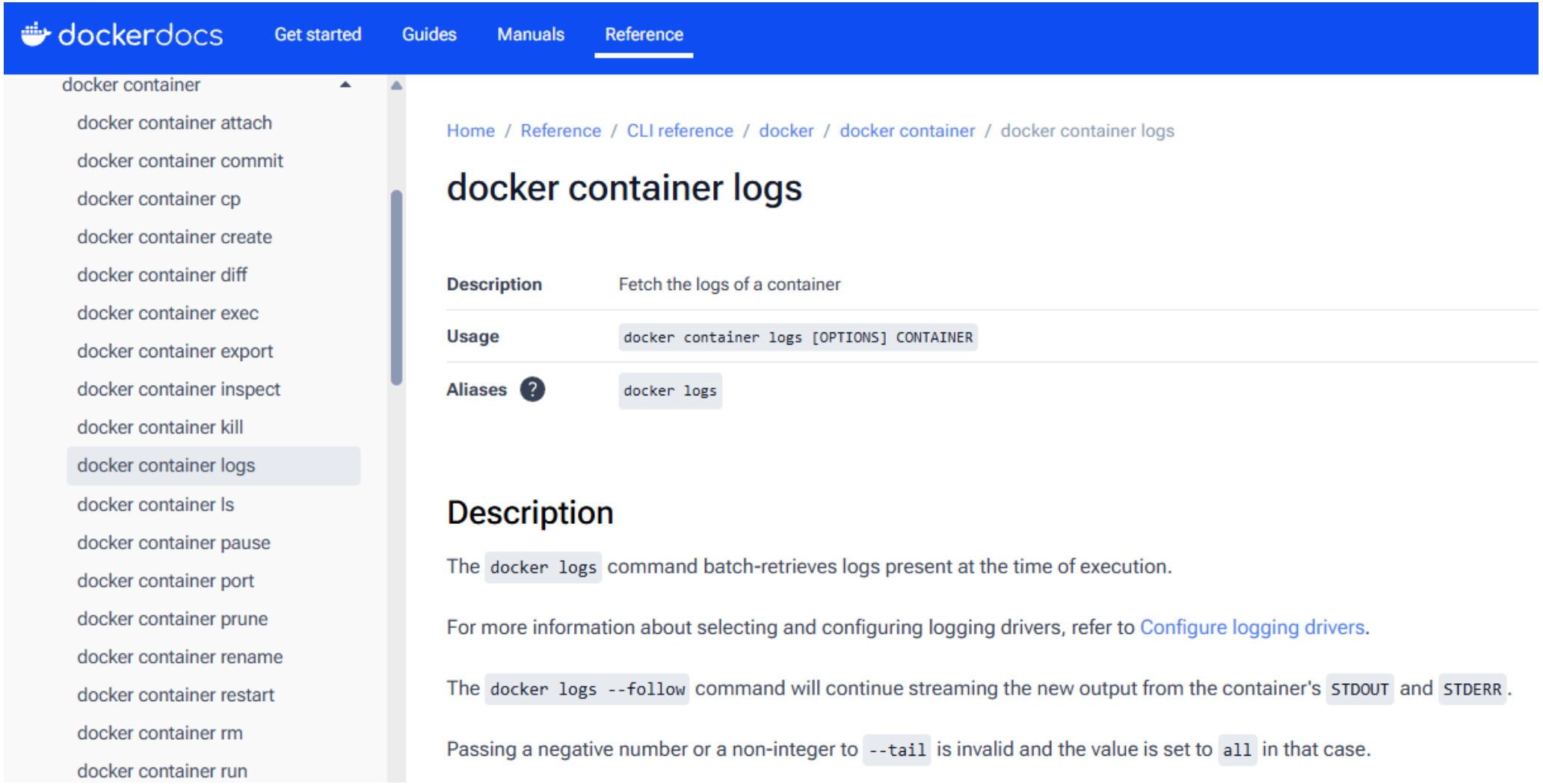
- Logs and output come directly from the main process, which makes them easy to capture in a CI/CD pipeline.

- Since images package dependencies, builds are consistent across environments, which is ideal for automated testing and delivery.

Docker Run



Docker Logs



POP QUIZ:

Why do we perform a manual deployment before automating it?

A. Because automation repeats the steps you already understand manually
B. Because automation tools cannot run without manual deployment
C. Because Docker requires it for licensing
D. Because VMs only allow manual deployment

POP QUIZ:

Why do we perform a manual deployment before automating it?

A. **Because automation repeats the steps you already understand manually**
B. Because automation tools cannot run without manual deployment
C. Because Docker requires it for licensing
D. Because VMs only allow manual deployment

POP QUIZ:

What is required before you can manually deploy a container to a VM?

A. A Kubernetes manifest
B. An SSH key and Docker installed on the VM
C. A load balancer already configured
D. A Jenkins agent running on the VM

POP QUIZ:

What is required before you can manually deploy a container to a VM?

A. A Kubernetes manifest
B. **An SSH key and Docker installed on the VM**
C. A load balancer already configured
D. A Jenkins agent running on the VM

POP QUIZ:

What does the `-p 80:8000` flag do when running a container?

A. It installs Docker
B. It scales containers across multiple VMs
C. It maps a VM port to a container port
D. It validates the Dockerfile before running

POP QUIZ:

What does the `-p 80:8000` flag do when running a container?

A. It installs Docker
B. It scales containers across multiple VMs
C. **It maps a VM port to a container port**
D. It validates the Dockerfile before running

POP QUIZ:

What should you do AFTER manually deploying the container?

A. Push the VM image to Docker Hub
B. Test the running application
C. Delete the Dockerfile
D. Disable all inbound traffic

POP QUIZ:

What should you do AFTER manually deploying the container?

A. Push the VM image to Docker Hub
B. **Test the running application**
C. Delete the Dockerfile
D. Disable all inbound traffic

POP QUIZ:

What happens if a container’s main process exits?

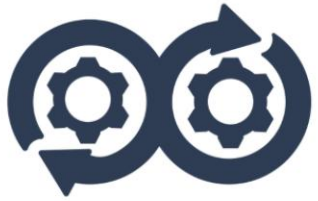
A. Docker automatically rebuilds the image
B. Docker restarts the VM
C. The container stops completely
D. The container switches to a backup process

POP QUIZ:

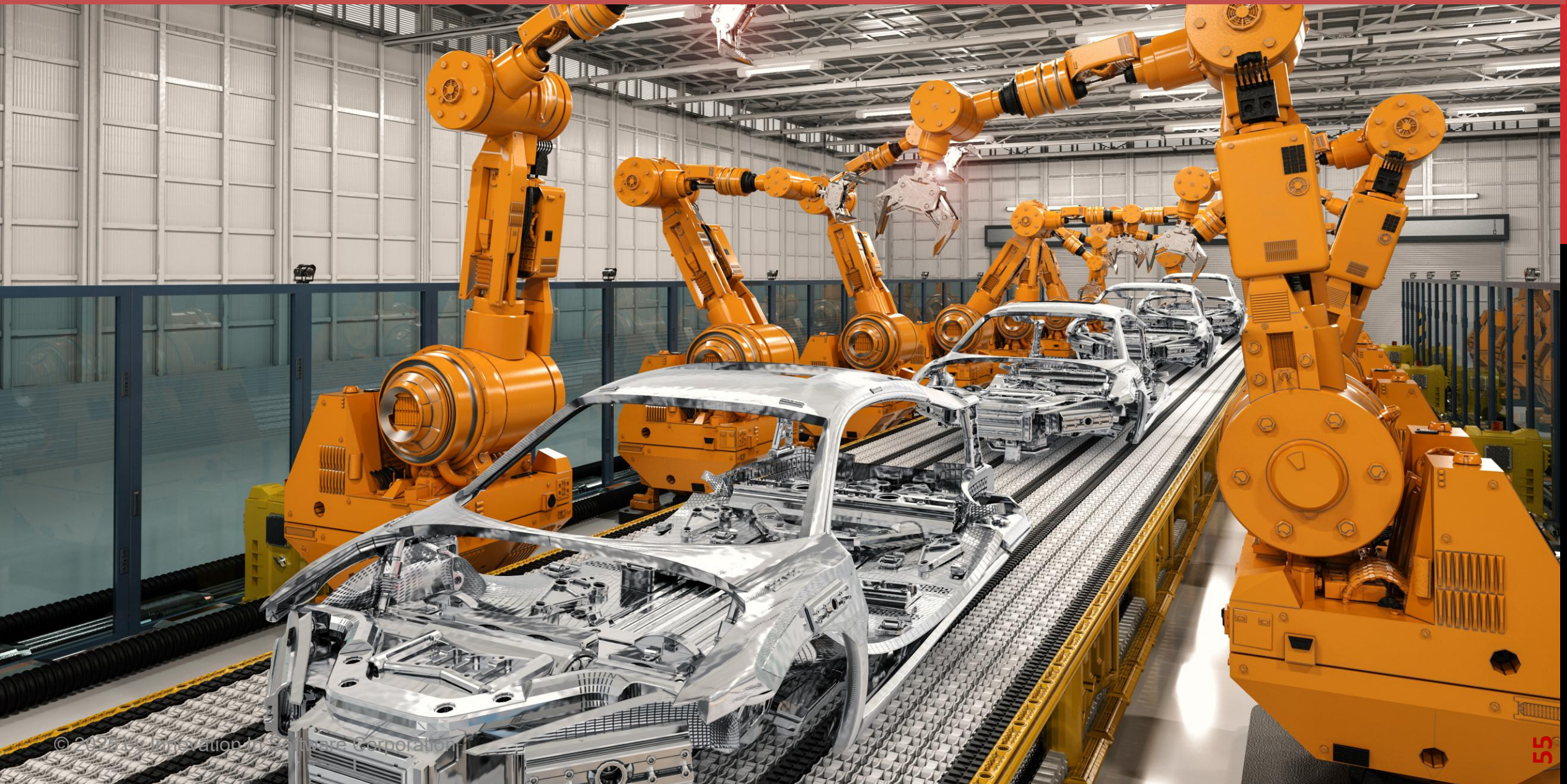
What happens if a container’s main process exits?

A. Docker automatically rebuilds the image
B. Docker restarts the VM
C. **The container stops completely**
D. The container switches to a backup process

Lab: Manual Deployment



Ansible Deployment



Ansible

Ansible is an automation tool that configures systems, installs software, and runs tasks across servers using simple YAML playbooks. It connects over SSH and does not require agents on the machines it manages.

It is commonly used for repeatable deployments, server configuration, and automating manual steps — making it a perfect fit for automating the container deployment you just performed manually.

Ansible Features

Playbook

The main file that defines *what* you want to do. A playbook contains one or more plays, and each play runs tasks on a group of hosts.

Tasks

The individual actions Ansible performs, such as installing a package, copying a file, or starting a service. Tasks are listed inside a playbook or can be organized into roles for reuse.

Inventory

A list of servers Ansible will manage. Inventories can be simple files with IP addresses or dynamic sources that fetch servers based on tags or cloud metadata.

Ansible Features

Roles

A way to organize playbooks into reusable components. Roles bundle tasks, variables, templates, and handlers into a structured format to keep projects clean and maintainable.

Variables

Values that can change between environments or hosts. Variables let you customize deployments without modifying the playbook itself.

Handlers

Special tasks that run only when notified, often used for actions like restarting a service after a configuration change.

Idempotency

A core Ansible principle ensuring that running the same playbook multiple times produces the same result. Ansible only makes changes when something is actually different.

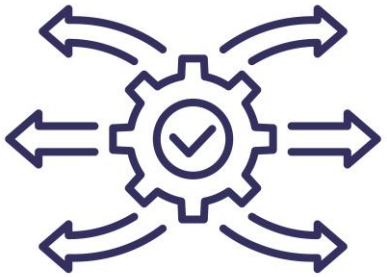
Ansible

Ansible lets us turn the manual deployment process into repeatable, reliable automation. Instead of logging into servers and typing commands, Ansible enforces configuration the same way every time.

Key points

- Eliminates human error
- Makes deployments repeatable across many servers
- Ensures systems converge to the same state
- Integrates naturally with CI/CD pipelines
- Deployment becomes “infrastructure as code”

Continuous Deployment



Ansible is typically used during the **CD (Continuous Deployment)** phase to configure servers and deploy application versions.

Common patterns

- CI (GitHub Actions / Jenkins) builds & tests the app
- CI pushes the image to Docker Hub
- CD uses Ansible to deploy the image to servers
- Repeat for blue/green or rolling updates

Build → Deliver → Deploy

Tools for CICD

Tool	Purpose	Example
Terraform	Create infrastructure	EC2, VPC, SGs
Ansible	Configure / deploy software	Install Docker, run container
Jenkins / GitHub Actions	Orchestrate automation	Trigger builds/deploys

These tools are complementary of each other

Ansible Playbook

A deployment playbook usually includes:

- Installing or configuring dependencies (Docker, packages, services)
- Pulling the correct application version
- Starting the application
- Verifying the service is running
- Ensuring idempotency for repeated runs

Idempotency

Idempotency means the playbook produces the same result every time you run it.

Why it matters for deployment

- Safe to re-run after failures
- Ensures consistent environments
- Detects real changes (new version, config updates)
- Allows hands-off automation

POP QUIZ:

What is the purpose of an Ansible playbook?

A. To define the desired state and steps needed to configure or deploy systems
B. To store SSH keys for remote hosts
C. To open firewall ports automatically
D. To build Docker images for CI workflows

POP QUIZ:

What is the purpose of an Ansible playbook?

A. **To define the desired state and steps needed to configure or deploy systems**
B. To store SSH keys for remote hosts
C. To open firewall ports automatically
D. To build Docker images for CI workflows

POP QUIZ:

Why is Ansible useful for deployment across multiple servers?

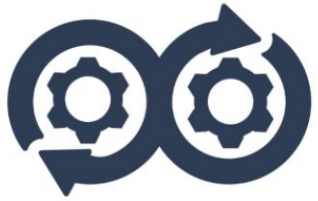
A. It ignores server differences and assumes all hosts are identical
B. It can run commands in parallel and enforce consistent configuration
C. It only runs on servers with Docker pre-installed
D. It requires no SSH permissions

POP QUIZ:

Why is Ansible useful for deployment across multiple servers?

A. It ignores server differences and assumes all hosts are identical
B. **It can run commands in parallel and enforce consistent configuration**
C. It only runs on servers with Docker pre-installed
D. It requires no SSH permissions

Lab: Ansible Deployment



Congratulations

You accomplished a full end-to-end deployment workflow today.

Today you learned

- How Jenkins runs pipelines using the controller/agent model
- How to build and push Docker images through Jenkins
- How to manually deploy and test containers on a VM
- How to automate deployments with Ansible using playbooks, tasks, and roles

You now have both the manual and automated foundations needed for real-world deployment workflows.