Introduction to Jenkins

Module 2: Jenkins Overview

Module Topics

- Jenkins in modern DevOps ecosystems
- Jenkins LTS architecture and components
- Plugins, security model, and governance
- Controller/agent overview
- SCM-driven automation (GitHub, GitLab)
- Installing Jenkins LTS
- Running Jenkins in Docker
- Configuration-as-Code (JCasC)
- Initial setup wizard and global tools

Origins

- In the early 2000s, most software teams
  - Integrated code infrequently
  - Ran builds manually
  - Discovered integration problems late in the development cycle
- In 2004 by Kohsuke Kawaguchi at Sun Microsystems created the Hudson project
  - Open-source written in Java
  - Designed to automate builds and tests
  - Integrated well with Java build tools like Ant and Maven
  - Easy to set up and extensible
  - Focused on automation rather than process enforcement

Hudson to Jenkins (2011)

- In 2010–2011, Oracle acquired Sun Microsystems
  - This led to disagreements between the Hudson community and Oracle regarding
    - *Project governance*
    - *Trademark ownership*
    - *Contribution control*
- As a result
  - The community forked Hudson
  - The new project was named Jenkins
  - Most contributors and users moved to Jenkins
- From this point on
  - Jenkins became a community-driven project and the dominant tool for pipeline automation
  - Hudson gradually declined in relevance

Job Centric CI (2011–2014)

- Early versions of Jenkins focused on
  - Freestyle jobs
  - UI-driven configuration
  - Scripted build steps
- Characteristics of early Jenkins
  - Configuration stored on the server
  - Heavy reliance on plugins
  - Tight coupling between built configurations and tool chains
  - Builds were often non-reproducible, especially after configuration changes
- This reflected the state of CI at the time
  - Automation existed, but IaC was not common, nor was the idea of a pipeline

Pipeline as Code (2014–2016)

- As DevOps practices matured, teams demanded
  - Configuration of the pipeline logic needed to be versioned as the logic evolved
  - Reproducibility became a primary concern: consistent builds
  - Better support for complex workflows
- This led to the introduction of the Jenkins Pipeline and Jenkinsfile
- Innovations
  - Pipelines defined in code
  - Stored in Git repositories
  - Reviewed like application code
  - Support for stages, parallelism, and conditions
- This marked a major evolution in Jenkins’ design philosophy

DevOps Era (2016–Present)

- Jenkins evolved to support
  - Microservices
  - Containers and Docker
  - Kubernetes-based build agents
  - Cloud-native workflows
- Notable Trends
  - Declarative pipelines simplified syntax
  - Dynamic agents replaced static build servers
  - Integration with modern SCM platforms (GitHub, GitLab)
- Jenkins shifted from being “a CI tool” to a general-purpose automation engine

Jenkins Today

- Why Jenkins is still relevant
  - Extreme flexibility
  - Large plugin ecosystem
  - Strong support for legacy and complex systems
- Challenges
  - Requires operational effort
  - Plugin compatibility management – many plugins are maintained by volunteers
  - Competition from managed CI/CD platforms like GitLab
- Despite newer tools, Jenkins remains widely used in
  - Large scale enterprises
  - Regulated environments
  - Complex, heterogeneous systems

GitLab and GitHub

- GitHub Actions and GitLab CI/CD are modern alternatives to Jenkins pipelines
  - These are integrated pipeline tools built directly into their Git hosting services
- These repository tools treat CI/CD as a native feature of the repository
  - Jenkins = external automation engine
  - GitHub/GitLab = pipelines embedded in the SCM platform
- The same basic pipeline concepts apply to these as to Jenkins
  - The syntax and format of the configuration files differs
  - GitLab/GitHub use yaml configuration files
  - Jenkins uses groovy scripting

GitHub Actions

- GitHub Actions
  - GitHub’s built-in automation and CI/CD platform
  - Pipelines are defined using YAML workflow files stored in the repository
- Key characteristics
  - Tight integration with GitHub repositories
  - Event-driven (push, pull request, tag, issue events)
  - Pipelines live in the directory “.github/workflows”
  - Uses hosted runners or self-hosted runners
- Conceptual flow
  - GitHub event → Action workflow → Jobs → Steps
- Strengths
  - Zero infrastructure to manage
  - Simple setup for GitHub users
  - Strong marketplace of reusable actions

GitLab CI/CD

- GitLab CI/CD is a first-class feature of GitLab, not an add-on.
  - Pipelines are defined in a file: “.gitlab-ci.yml”
- Key characteristics
  - CI/CD tightly integrated with GitLab repos
  - Built-in artifact storage and environment tracking
  - Native support for Docker and Kubernetes
  - GitLab runners execute jobs
- Conceptual flow
  - Git commit → GitLab pipeline → Stages → Jobs
- Strengths
  - Very cohesive developer experience
  - Strong DevOps lifecycle integration
  - Minimal configuration overhead

Comparison

- Configuration style contrast
- Jenkins
  - Highly flexible
  - Scriptable pipelines (Groovy)
  - Powerful but verbose
  - Requires plugin management
- GitHub / GitLab
  - Opinionated YAML syntax
  - Simpler mental model
  - Less customization, faster onboarding
  - Platform constraints apply

Jenkins vs GitHub Actions vs GitLab CI/CD

Feature	Jenkins	GitHub Actions	GitLab CI/CD
Hosting	Self-managed	SaaS	SaaS / Self
Pipeline Language	Jenkinsfile (Groovy)	YAML	YAML
Plugin Ecosystem	Very large	Marketplace actions	Built-in features
Customization	Extremely high	Moderate	High
Enterprise Control	Full	Limited	Strong

Jenkins vs GitHub Actions vs GitLab CI/CD

- Jenkins is preferred for
  - Complex, multi-tool workflows
  - Hybrid or on-prem environments
  - Advanced customization needs
  - Regulated industries
  - Long-running or specialized builds
- GitHub/GitLab are preferred for
  - Simple to moderate pipelines
  - Tight integration with SCM
  - Minimal infrastructure management
  - Fast onboarding

Comparison

- Jenkins is often chosen when there is a need for
  - Extreme customization
  - Supporting many SCM systems
  - Supporting complex legacy workflows
  - Infrastructure is managed and controlled
- GitHub / GitLab pipelines are often chosen when
  - Repositories are already hosted on the platform
  - Teams want fast setup
  - Minimal operational overhead is desired
  - Standard CI/CD patterns are sufficient

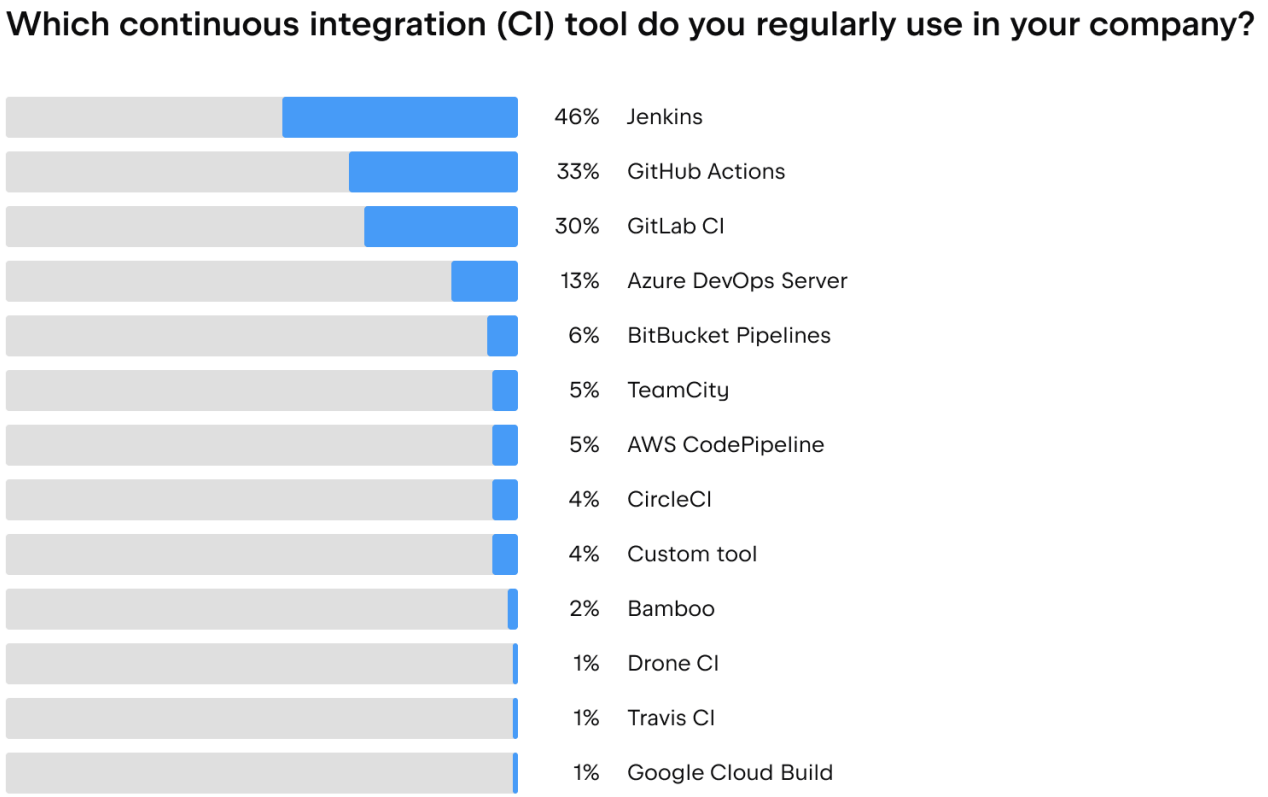


Image Credit: https://blog.jetbrains.com/teamcity/2023/08/how-to-choose-cicd-tool/

Modern DevOps Ecosystems

- Jenkins is an automation engine that
  - Listens to SCM events
  - Executes pipelines
  - Orchestrates execution of builds, tests, scans, and deployments
  - Integrates with many external tools
    - *SCM: GitHub, GitLab, Bitbucket*
    - *Build tools: Maven, Gradle, npm, pip*
    - *Containers: Docker, Podman*
    - *Kubernetes: Jenkins Kubernetes plugin*
    - *Quality & Security: SonarQube, Snyk, Trivy*
    - *Secrets: Vault, cloud KMS systems*
- Jenkins is usually the best choice when
  - The pipeline needs to integrate across a variety of tool chains
  - Full control over pipeline workflows is required
  - Operational needs span hybrid or multi-cloud environments

Jenkins Release Model

- Jenkins has two release lines
- Weekly releases
  - New features
  - Faster change
  - Higher risk
- LTS (Long-Term Support)
  - Stability-first
  - Security patches backported from the weekly releases
  - Recommended for production

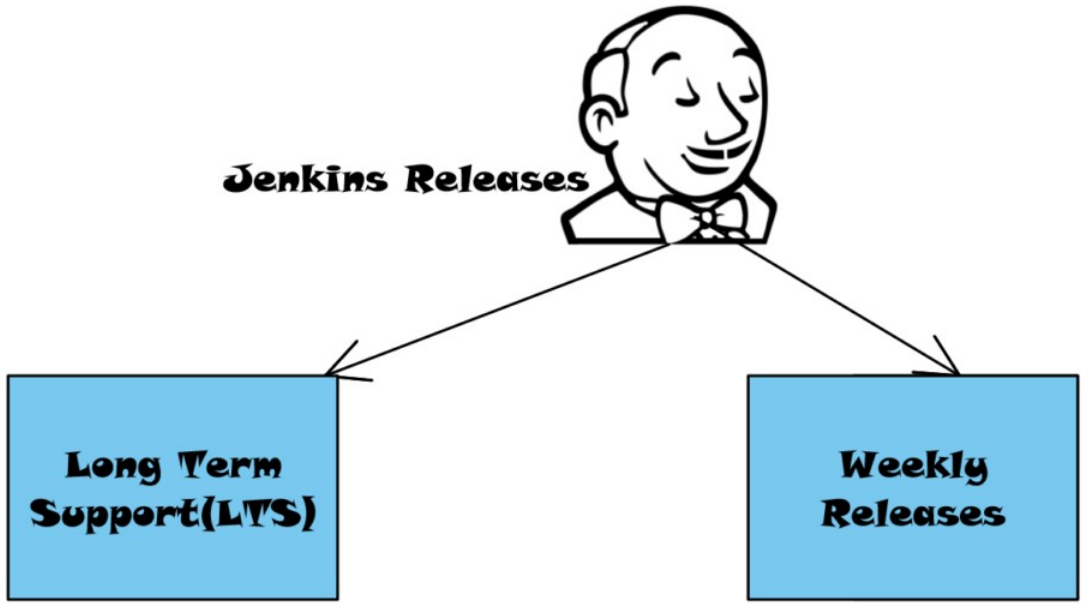


Image Credit: https://www.testingdocs.com/category/jenkins/

Jenkins Weekly Releases

- Represent the latest state of Jenkins development
- Typically used by
  - Plugin developers
  - Jenkins core contributors
  - Early adopters testing new features
  - Teams validating future LTS upgrades
- Not meant for
  - Production controllers
  - Regulated environments

Jenkins LTS

- Recommended for production environments
  - Plugins are tested primarily against LTS
  - Predictable upgrade cycles
    - *Allows for enterprise planning for moving to newer versions*
  - Security patches are backported from the weekly build
    - *Ensures that security issues are addressed in LTS*
- Design assumes long running controller
  - Optimized for production environments

Long Running Controller

- The controller
  - Maintains persistent state on disk
  - Accumulates configuration over time
- Retains
  - Job definitions
  - Build history
  - Security credentials
  - Plugin data
  - User configuration
- Different design architecture than
  - Stateless web services
  - Ephemeral CI runners
  - Serverless architectures

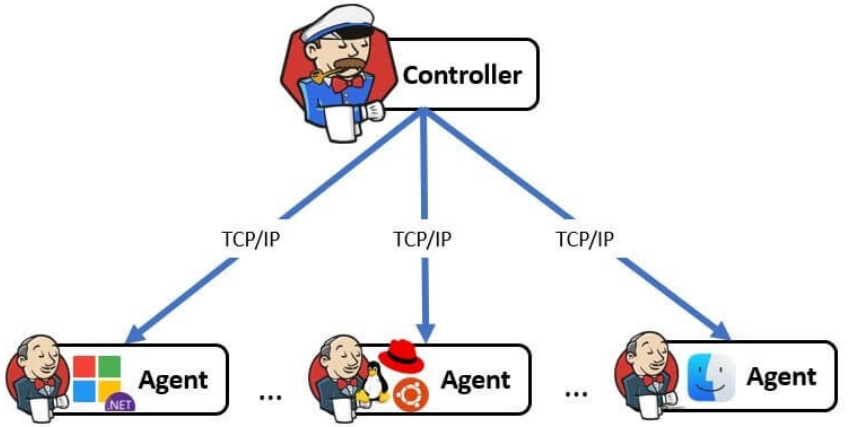


Image Credit: https://naiwaen.debuggingsoft.com/2022/06/jenkins-controller-and-agents-architecture/

Long Running Controller

- Jenkins controllers
  - Retain long build histories
  - Preserve logs for auditing
  - Track trends over time
  - Examples
    - *Test flakiness trends*
    - *Build duration changes*
    - *Deployment frequency*
- This historical continuity only works if
  - The controller persists
  - The file system remains stable

Jenkins Architecture

- Jenkins controller
  - Central brain of Jenkins
  - Responsibilities
    - *Web UI*
    - *REST API*
    - *Job and pipeline definitions*
    - *Build scheduling and queue management*
    - *Plugin management*
    - *Credential storage*
    - *User authentication and authorization*
    - *Should not run heavy builds*
  - Referred to as the master node in older versions of Jenkins

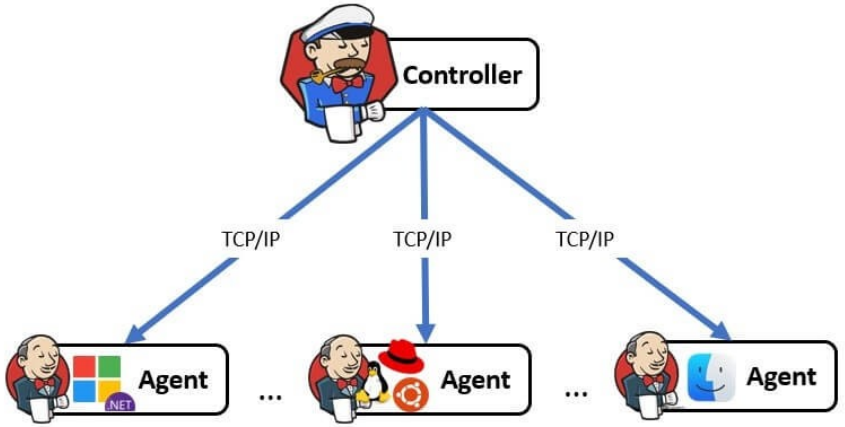


Image Credit: https://naiwaen.debuggingsoft.com/2022/06/jenkins-controller-and-agents-architecture/

Jenkins Architecture

- Jenkins controller should NOT
  - Run heavy builds
  - Compile large projects
  - Execute tests
- Best practice is the builds are never executed on the controller node
  - Builds are done by agents
  - Jenkins controller is idle most of the time
  - But running builds might disrupt the scheduling processes that are the core of the controller functionality

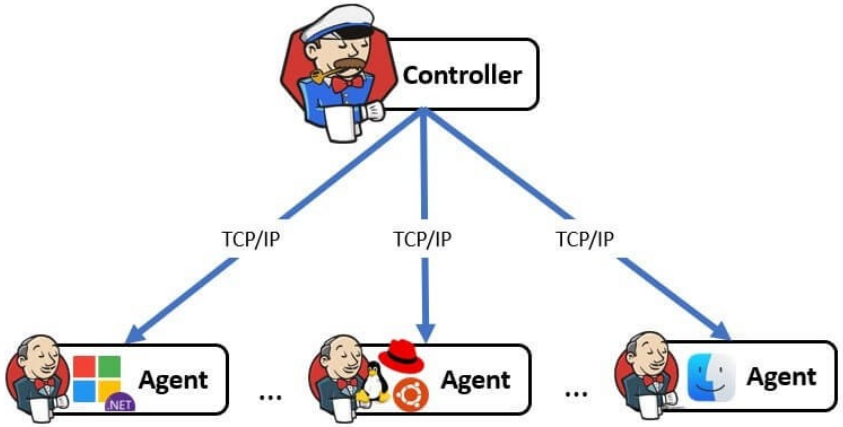


Image Credit: https://naiwaen.debuggingsoft.com/2022/06/jenkins-controller-and-agents-architecture/

Internal Controller Components

- Web application
  - Runs as a Java web app
  - Embedded Jetty servlet container
  - UI + API share the same backend
- Job configuration store
  - Jobs stored as XML on disk
  - Pipelines stored either:
    - *Inline (legacy)*
    - *From SCM (preferred)*
- Build queue
  - Pending builds waiting for executors
  - Queue logic decides
    - *When a job runs*
    - *On which agent*
    - *With which constraints*
- Executor management
  - Executors represent the capacity of an agent to run concurrent builds
  -

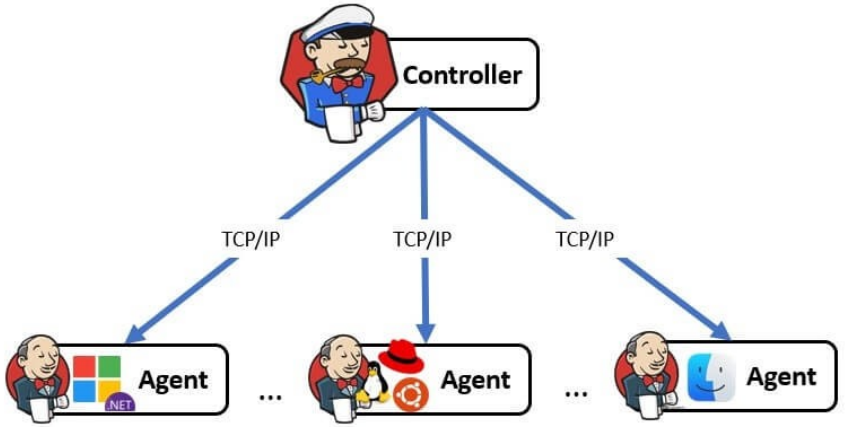


Image Credit: https://naiwaen.debuggingsoft.com/2022/06/jenkins-controller-and-agents-architecture/

Plugin Ecosystem

- Jenkins controller core is intentionally small
  - Most functionality comes from plugins
  - Thousands of plugins are available
  - Examples
    - *Git / GitHub / GitLab integration*
    - *Pipeline features*
    - *Kubernetes agents*
    - *Credentials and RBAC*
  - Plugin Risks
    - *Plugins run with high privileges*
    - *Outdated plugins = security risk*
    - *Version compatibility matters*

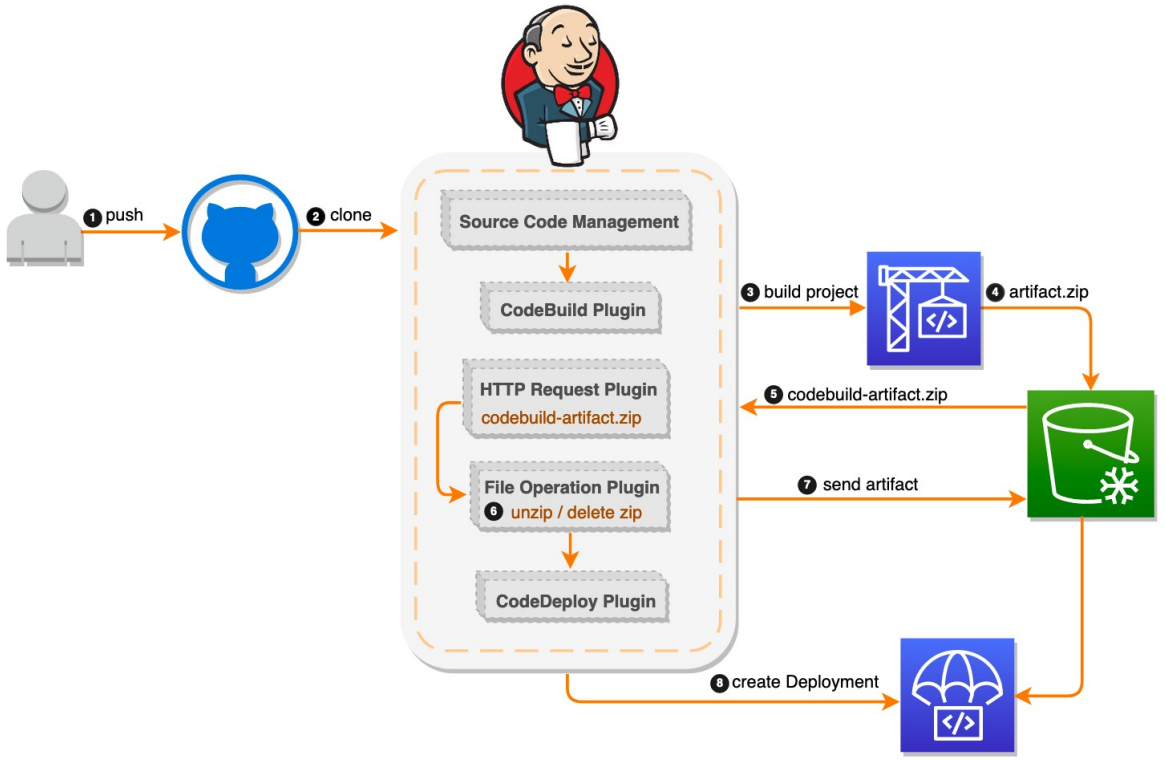


Image Credit: https://thinkwithwp.com/blogs/devops/setting-up-a-ci-cd-pipeline-by-integrating-jenkins-with-aws-codebuild-and-aws-codedeploy/

Security Model

- Authentication: local users, LDAP, SSO
- Authorization: role-based access control (RBAC)
  - Credentials stored centrally and injected securely
  - Fine-grained permissions per job, folder, or pipeline
- Governance considerations
  - Who can install plugins?
  - Who can modify pipelines?
  - How are credentials managed?
  - How are updates tested and rolled out?

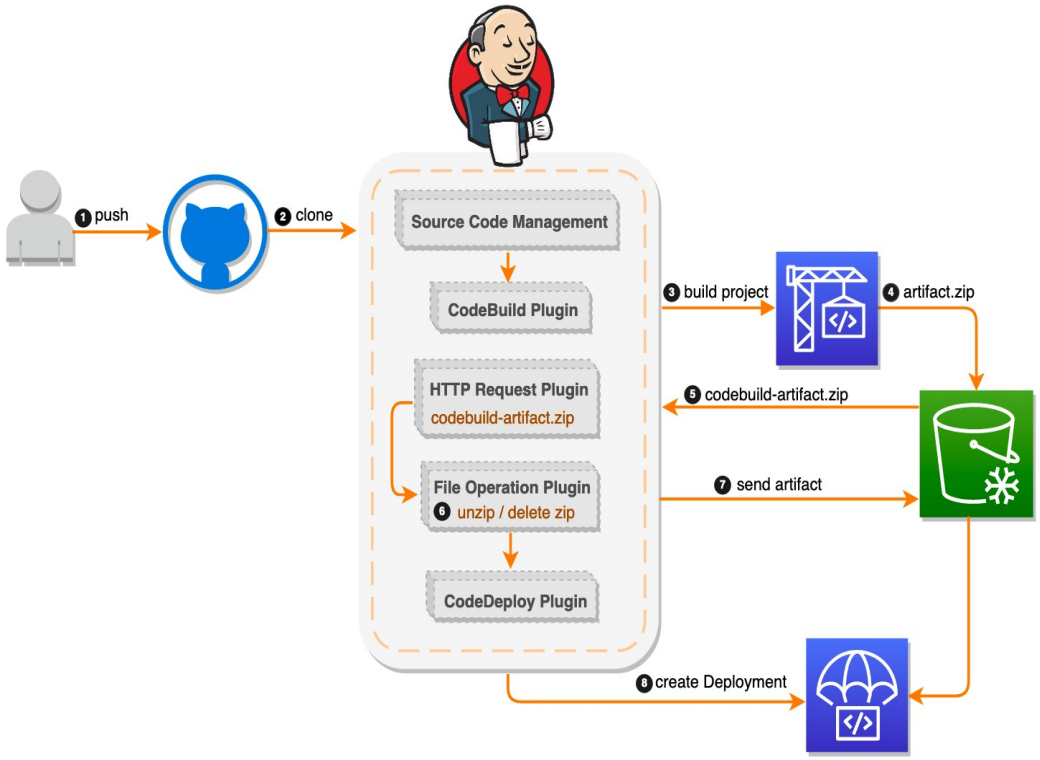


Image Credit: https://thinkwithwp.com/blogs/devops/setting-up-a-ci-cd-pipeline-by-integrating-jenkins-with-aws-codebuild-and-aws-codedeploy/

Controller/Agent Architecture

- Using agents provides
  - Separation of concerns
    - *For example, doing builds in different OS environments*
  - Scalability – can add more agents to manage workloads
  - Security isolation – each agent can have its own security profile
- Static agents
  - Always running
  - Simple but resource-heavy
  - Common in older Jenkins setups
  - Usually a physical host or a virtual machine

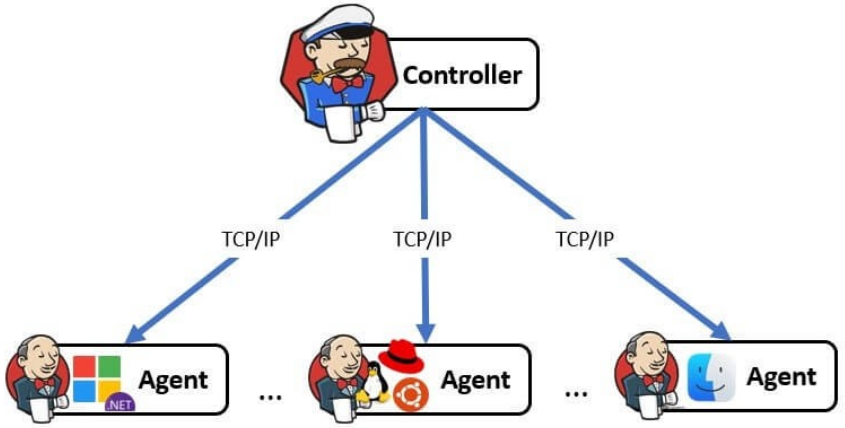


Image Credit: https://naiwaen.debuggingsoft.com/2022/06/jenkins-controller-and-agents-architecture/

Controller/Agent Architecture

- Ephemeral agents
  - Created on demand and destroyed after job completion
  - Often container-based or Kubernetes-based
  - Better isolation and scalability than static agents
  - Considered to be a modern best practice
- Execution flow of ephemeral agents
  - SCM event triggers pipeline
  - Controller schedules work to be done
  - Agent spins up
  - Pipeline stages execute the work
  - Agent is destroyed

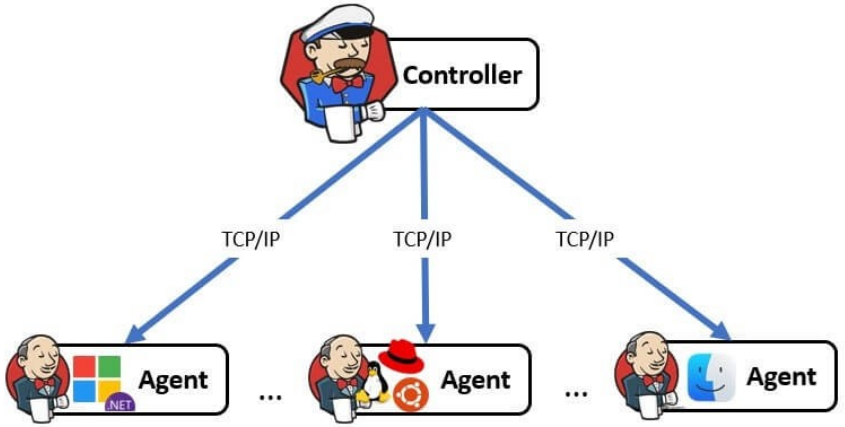


Image Credit: https://naiwaen.debuggingsoft.com/2022/06/jenkins-controller-and-agents-architecture/

SCM-Driven Automation

- Source control as the single source of truth
  - For both the application code and pipeline code (Jenkinsfile)
  - Pipelines ingest application code
  - Jenkinsfile is an IaC specification that defines the CI/CD steps and flow of execution
  - Jenkinsfile is versioned with the application
    - *Similar to how unit and integration tests are versioned*
  - Changes to Jenkinsfile code
    - *Follow code review processes*
    - *Subject to the same QA testing as application code*

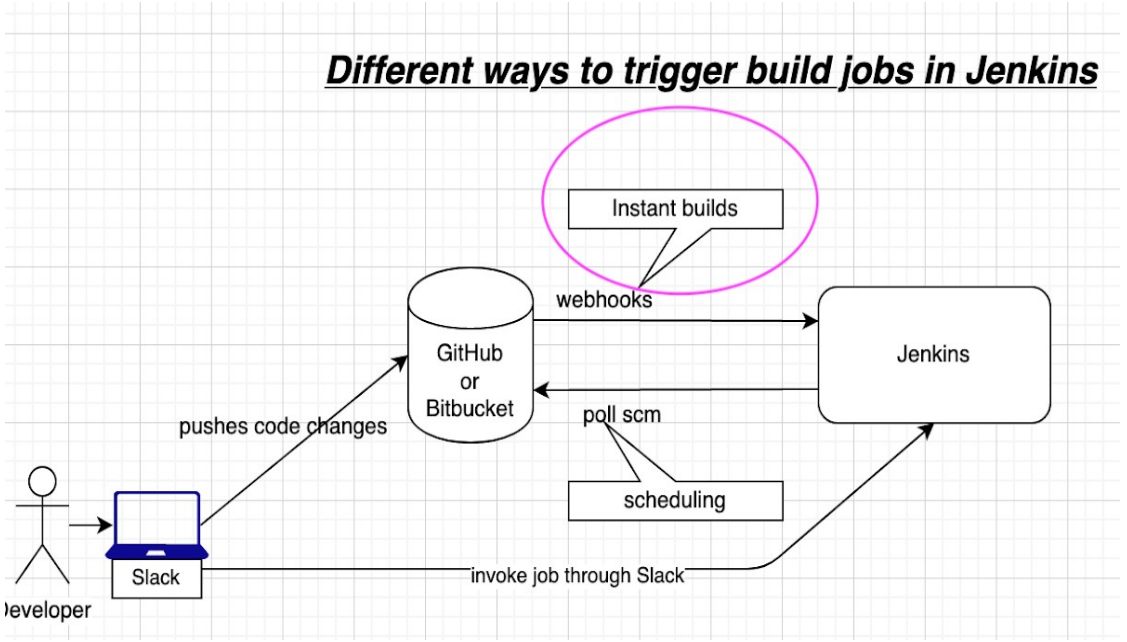


Image Credit: https://www.coachdevops.com/2020/06/how-to-configure-webhooks-in-bitbucket.html

SCM-Driven Automation

- Webhooks
  - GitHub/GitLab notify Jenkins on:
    - *Push*
    - *Pull/Merge Request*
    - *Tag creation*
  - Enables event-driven automation

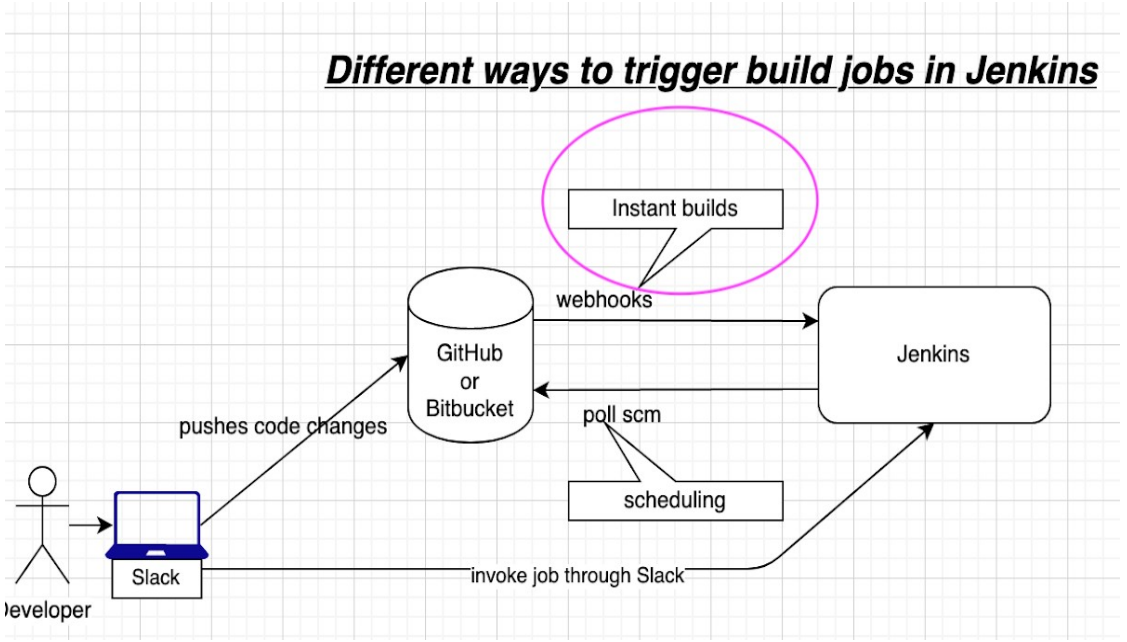
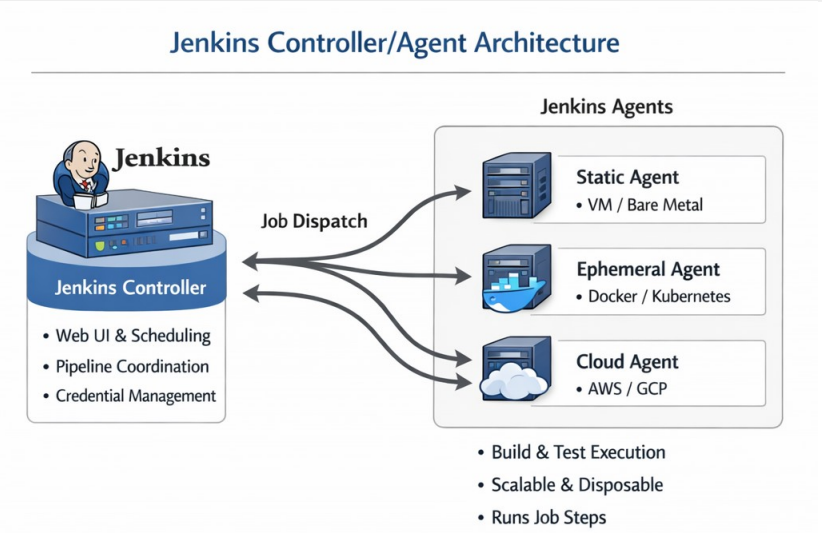
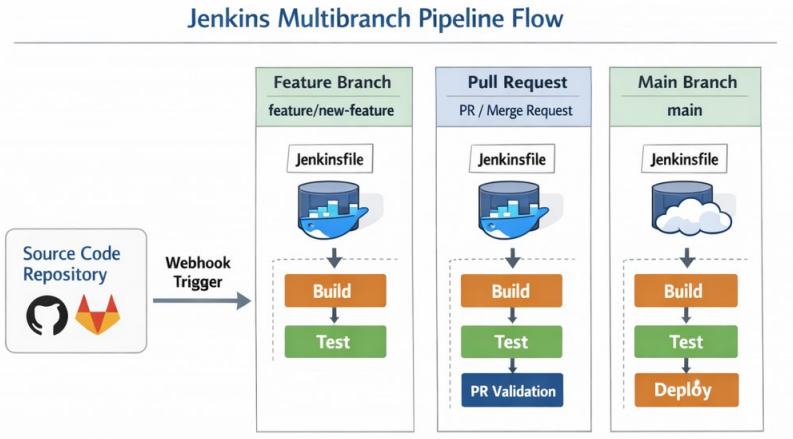


Image Credit: https://www.coachdevops.com/2020/06/how-to-configure-webhooks-in-bitbucket.html

SCM-Driven Automation

- Multibranch Pipelines
  - Jenkins automatically discovers branches
  - Each branch can have its own Jenkinsfile
- Enables
  - Pull request validation
  - Feature branch testing
  - Main branch deployments





Installing Jenkins LTS

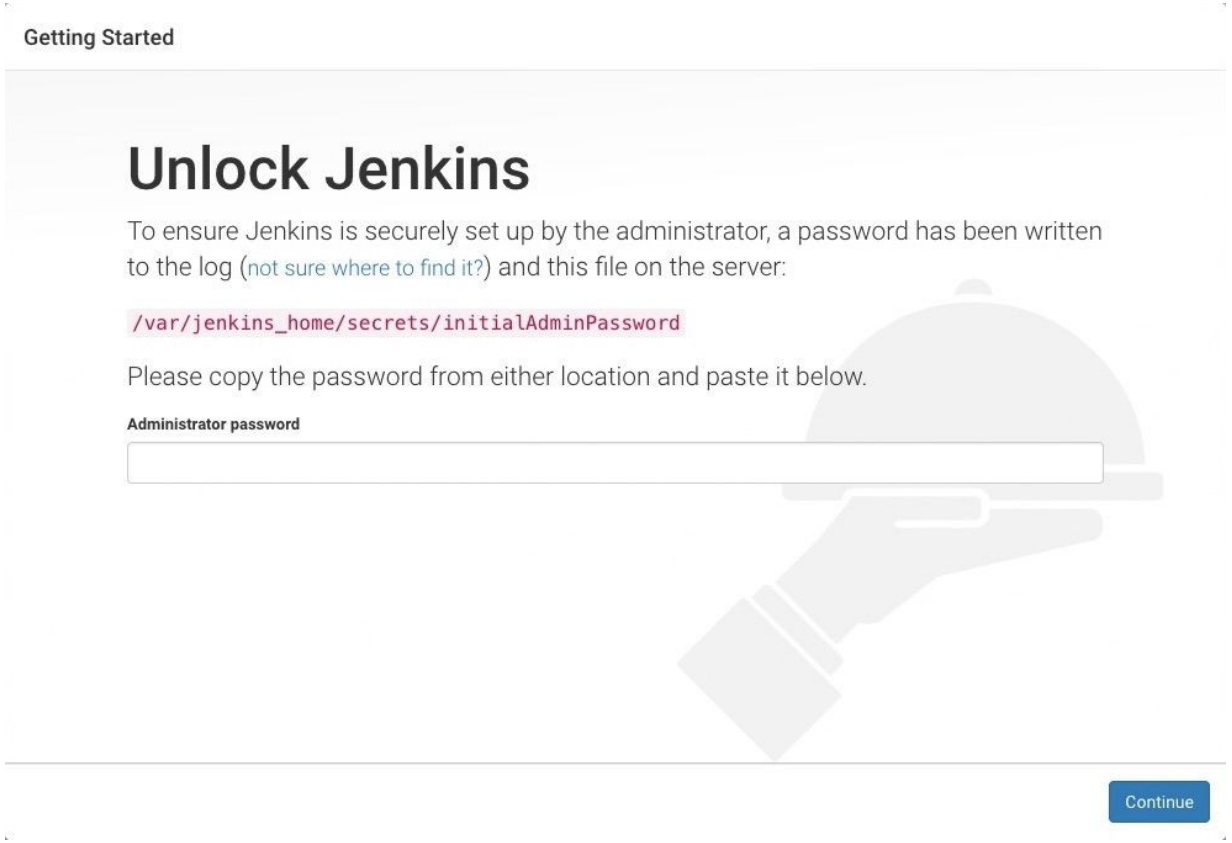
- Installing Jenkins means:
  - Installing the Jenkins controller
  - Running it as a long-lived service
  - Preparing it to manage jobs, pipelines, and agents
- Jenkins is a Java application distributed as
  - Native packages (RPM, DEB) or Windows MSI
  - WAR file
  - Docker image
- Always start with LTS unless there is a reason for using the weekly build

Installing Jenkins LTS

- Native Package Install
  - Installed as a system service
  - Runs on a VM or server
  - Direct access to filesystem
  - Traditional enterprise model
- WAR File
  - Portable
  - Manual startup
  - Often used for testing or learning
- Docker
  - Containerized Jenkins controller

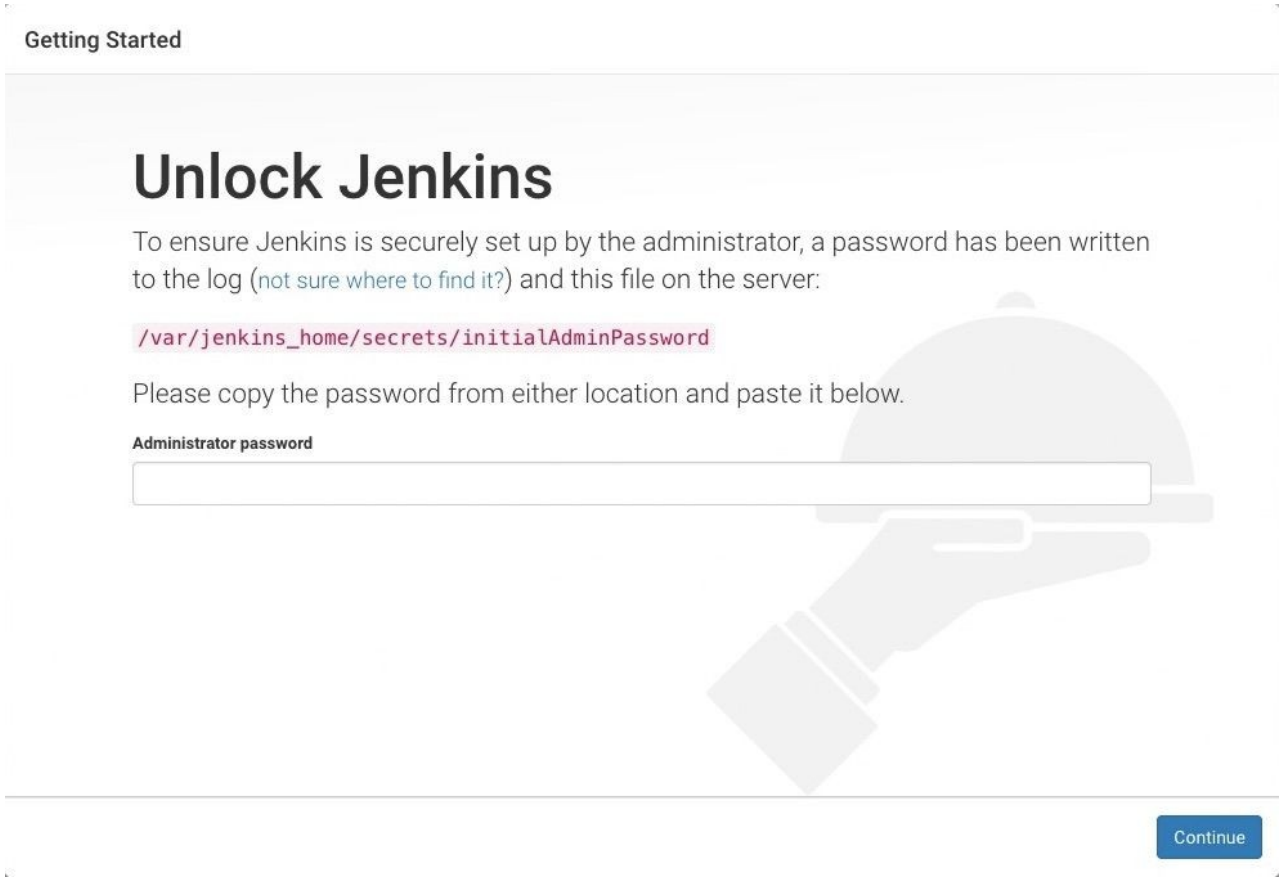
Installing Jenkins LTS

- During the first startup
  - Jenkins creates $JENKINS_HOME
  - Default configuration files are generated
  - Initial admin password is created
  - Setup wizard is triggered



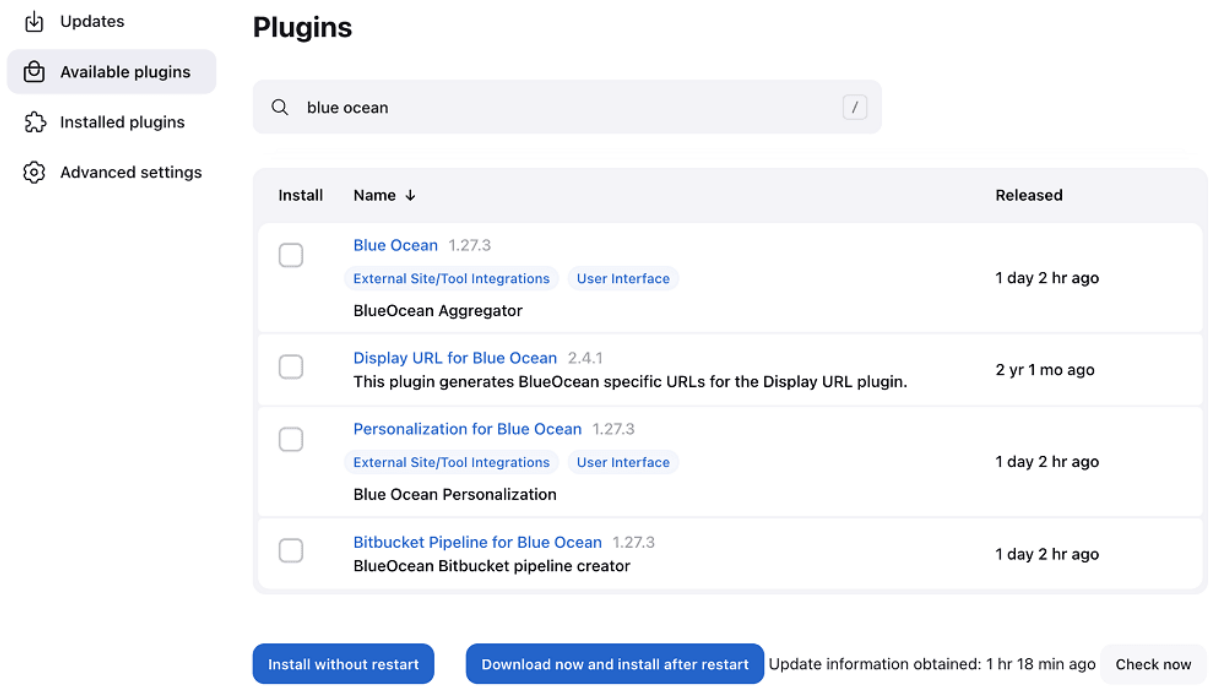
Initial Setup Wizard

- On first startup, Jenkins
  - Locks itself
  - Requests an admin password
  - Prompts for plugin installation
  - Creates the first admin user
  - Sets basic configuration



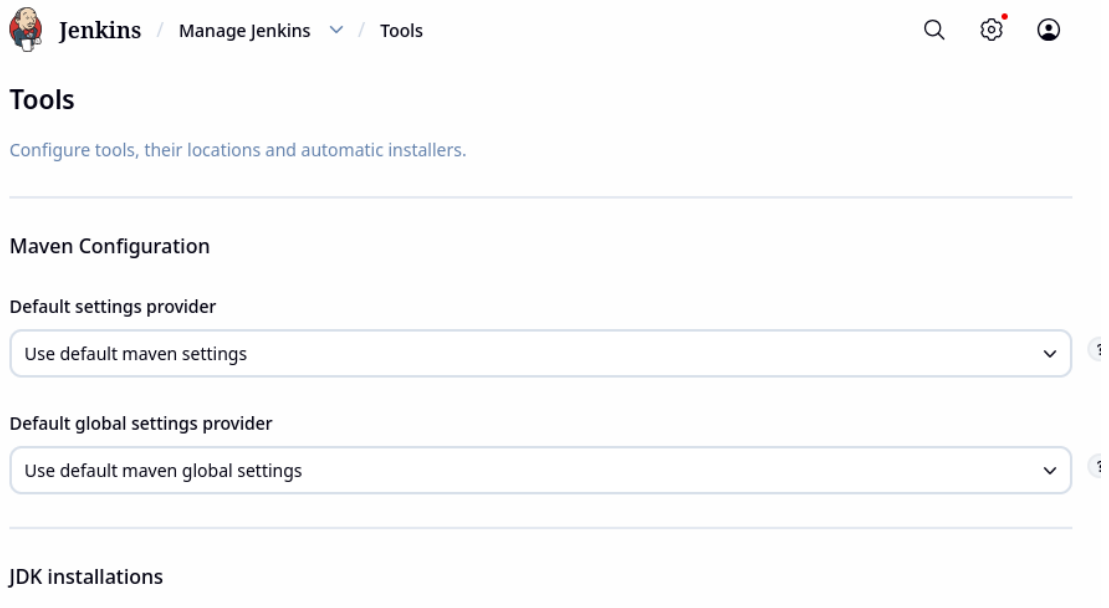
Plugin Selection

- Two common choices:
  - Suggested plugins
  - Custom plugin selection
- For most installations
  - Suggested plugins are sufficient
  - Plugins can always be added later
  - However, removing them is harder



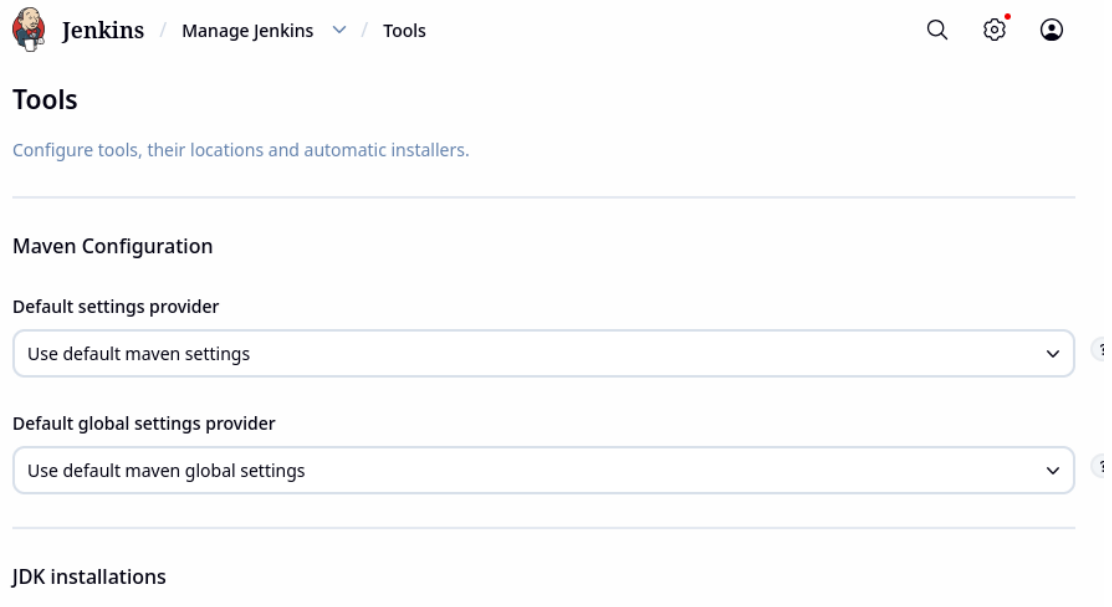
Global Tools Configuration

- Global tools:
  - Are *not* binaries
  - Are *not* installed on the controller
  - Are *not* automatically present on agents
  - Are named tool definitions
  - Are stored in Jenkins configuration
- Used by pipelines as references
  - Global tools are labels, not software



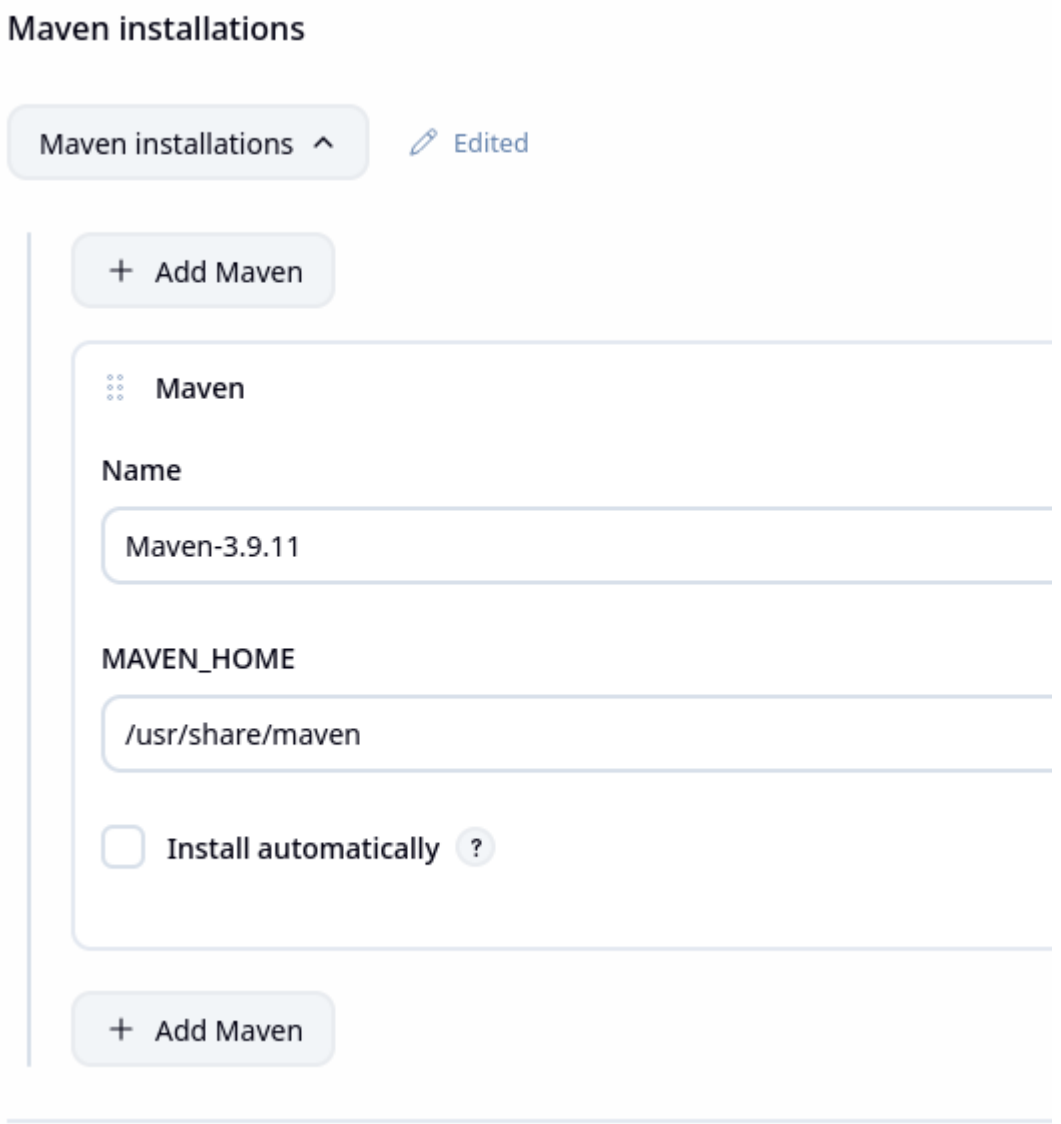
Reference Tools by Name

- In a Jenkinsfile:
  - “Use Maven version maven-3.9”
  - The name “maven-3.9” is defined globally in Jenkins
  - Abstracts away
    - *Version numbers*
    - *Installation paths*
    - *Agent OS differences*
- Benefits
  - Pipelines become portable
  - Jenkins controls versions centrally
  - Developers don’t hardcode paths to build tools in pipeline code



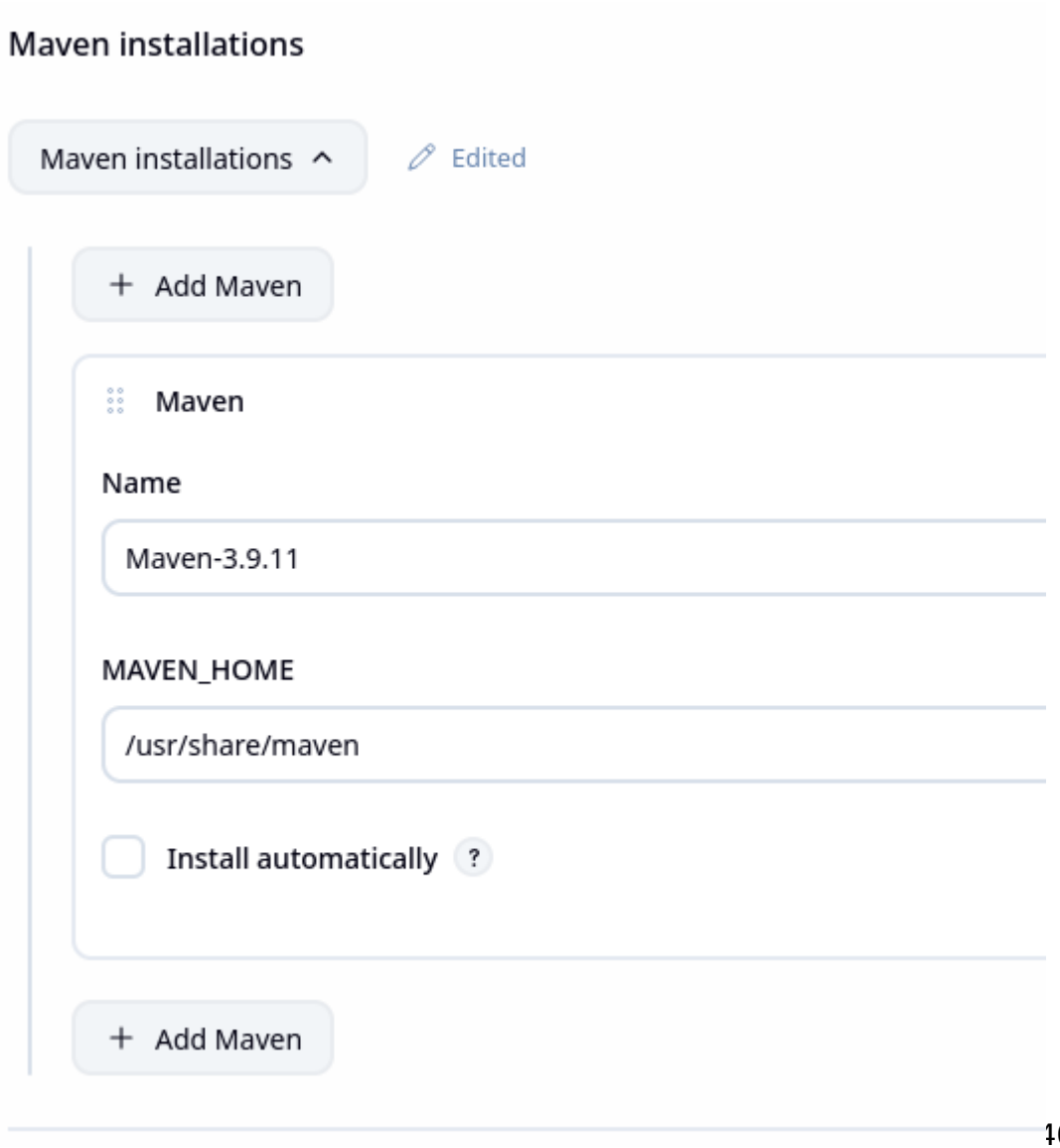
What Happens at Runtime

- Step 1: Pipeline starts
  - Controller reads the Jenkinsfile
  - Sees: “This pipeline needs Maven maven-3.9”
- Step 2: Agent is selected
  - Controller assigns an agent
  - Agent may be:
    - *A VM*
    - *A Docker container*
    - *A Kubernetes pod*
- Step 3: Tool resolution
  - Controller sends the tool definition to the agent
  - Definition includes:
    - *Tool type (Maven, JDK, Node)*
    - *Version*
    - *Installation strategy*



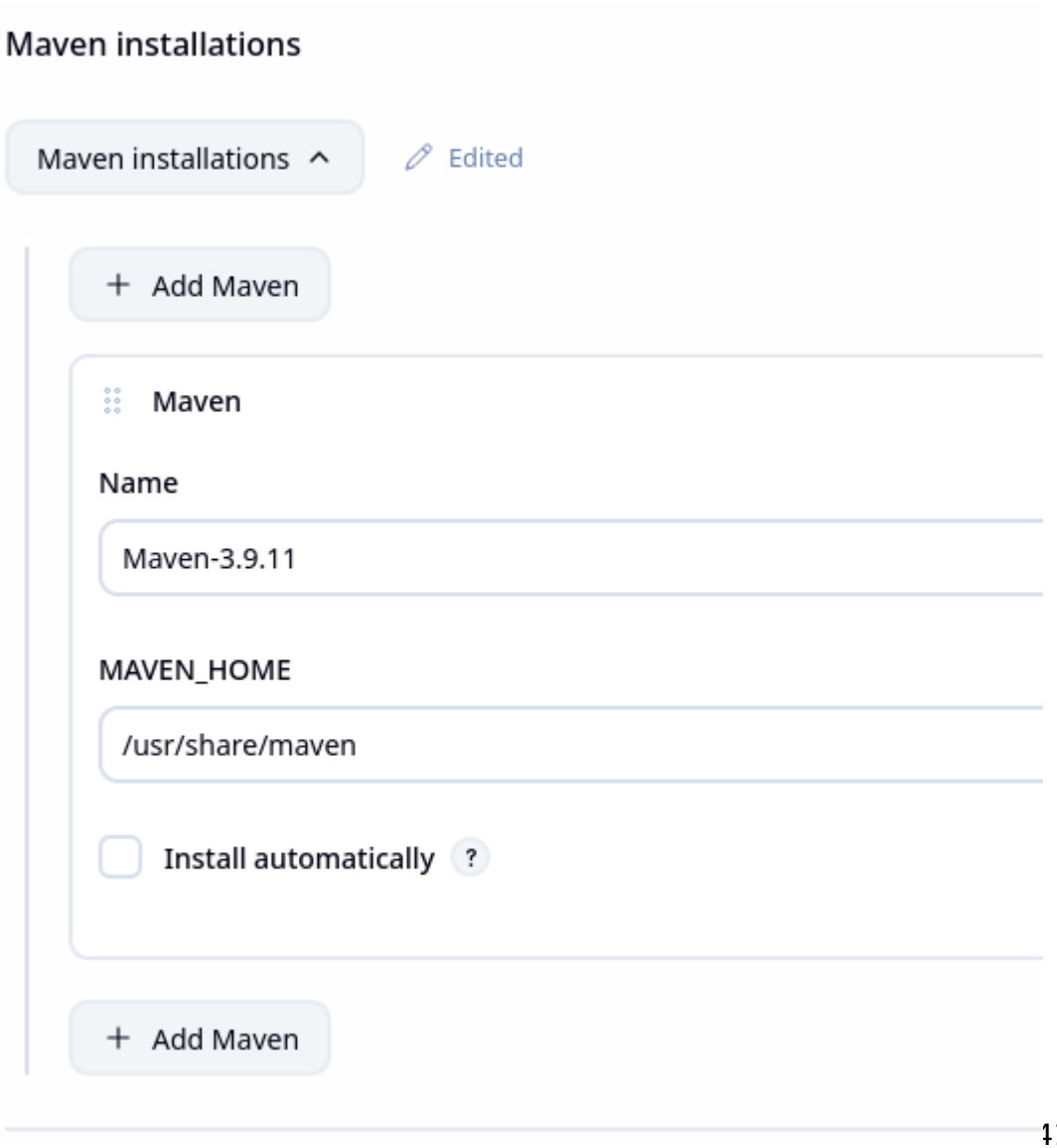
What Happens at Runtime

- Step 4: Tool installation (if needed)
  - On the agent, if tool is already installed then it is used
  - If not
    - *Jenkins downloads it*
    - *Installs it locally*
    - *Caches it for reuse (if agent persists)*
- Step 5: Tool execution
  - Agent updates PATH and environment variables
  - Pipeline steps run
  - Controller receives logs and results



What Auto-Install Means

- Jenkins can download tools automatically
- Controlled by:
  - Tool definition settings
  - Plugin support
  - Agent permissions
- Benefits
  - No manual agent setup
  - New agents work immediately
  - Ephemeral agents become practical
- Pitfalls
  - Builds depend on external downloads
  - Tool versions can change unexpectedly
  - Network failures can break builds



Ensuring Consistent Builds

- Without global tools
  - Each agent might have different versions
  - Pipelines hardcode paths
  - “Works on my agent” problems appear
- With global tools
  - Jenkins enforces version consistency
  - Pipelines are environment-agnostic
  - Upgrades are centralized and controlled
- Critical when
  - Multiple teams share Jenkins
  - Agents are ephemeral
  - Jenkins uses containers or Kubernetes

```
pipeline {
    agent any

    tools {
        // These refer to Global Tool Configuration names
        jdk 'JDK17'
        maven 'Maven3'
    }

    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/example/my-java-app.git'
            }
        }

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

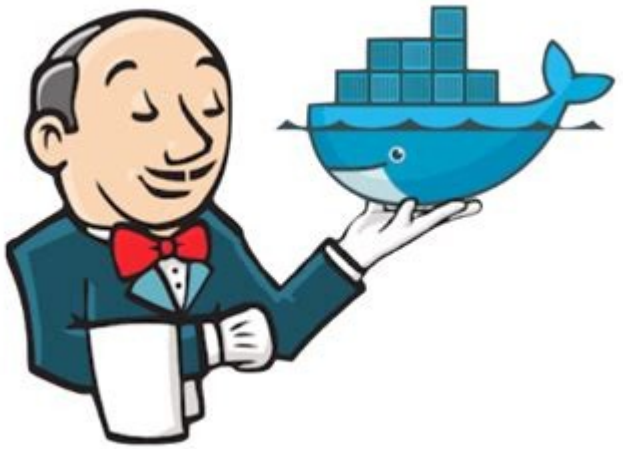
Controller vs Agent Tooling

- Controller responsibilities
  - Stores tool definitions
  - Defines names and versions
  - Decides what tools are needed
  - Never runs builds
- Agent responsibilities
  - Installs tools
  - Executes commands
  - Runs builds and tests
  - Cleans up afterward
- The controller is the planner
- The agent is the worker

```
pipeline {
    agent any

    tools {
        // These refer to Global Tool Configuration names
        jdk 'JDK17'
        maven 'Maven3'
    }

    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/example/my-java-app.git'
            }
        }

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

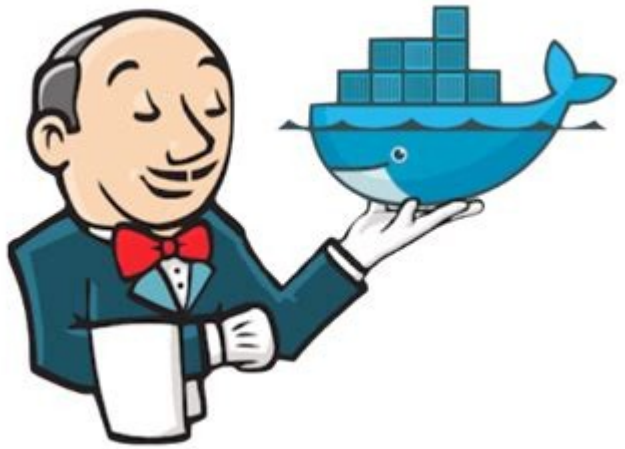
Running Jenkins in Docker

- Docker helps solve common Jenkins problems
  - Consistent environments
  - Easier upgrades
  - Simplified rollback
  - Faster setup for labs and training
- Jenkins in Docker is NOT Jenkins as a stateless service
  - The controller still needs persistent storage



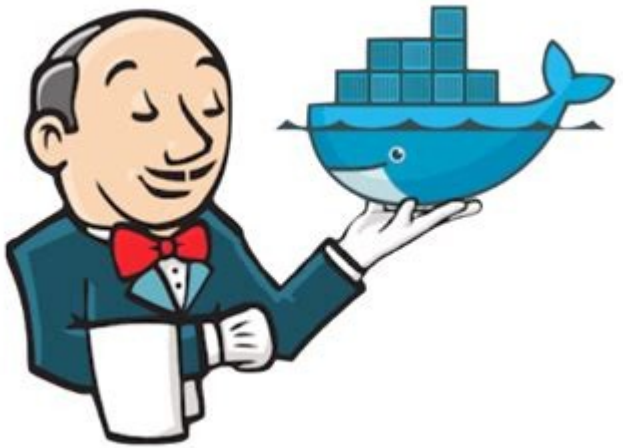
Running Jenkins in Docker

- Persistent volumes
  - Jenkins state must survive container restarts
  - $JENKINS_HOME is mounted to a volume
  - Losing this volume = losing Jenkins
- Container Lifecycle
  - Container can be destroyed and recreated
  - Jenkins data remains intact
  - Encourages clean operational practices



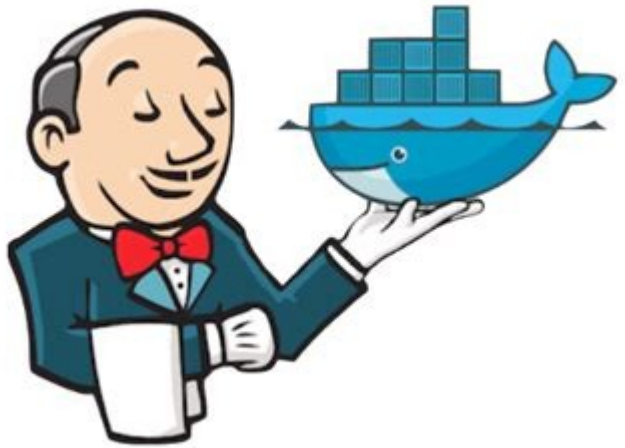
Running Jenkins in Docker

- Running Jenkins in Docker means
  - Jenkins runs inside a container
  - Jenkins data lives in a mounted volume
  - The container can be replaced without losing Jenkins state
  - Provides control, consistency, and safety without changing Jenkins' architecture



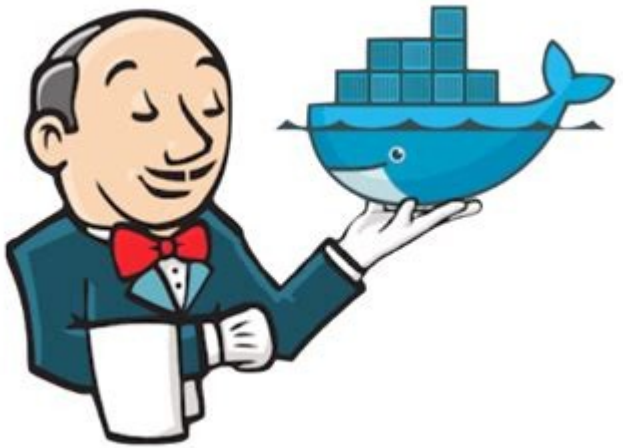
Docker: Fast and Predictable Setup

- Without Docker
  - Install Java
  - Configure OS packages
  - Manage service startup
  - Troubleshoot environment differences
- With Docker
  - Pull the Jenkins LTS image
  - Start a container
  - Jenkins is immediately usable
  - Everyone gets the same Jenkins
  - Avoids setup chaos



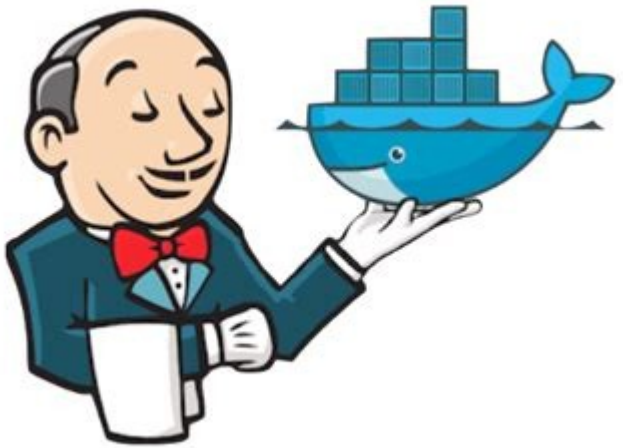
Docker: Environment Consistency

- Docker ensures
  - Same Jenkins version
  - Same Java version
  - Same default configuration
  - Same filesystem layout
- Eliminates
  - “Works on my machine”
  - OS-specific surprises
  - Hidden dependencies



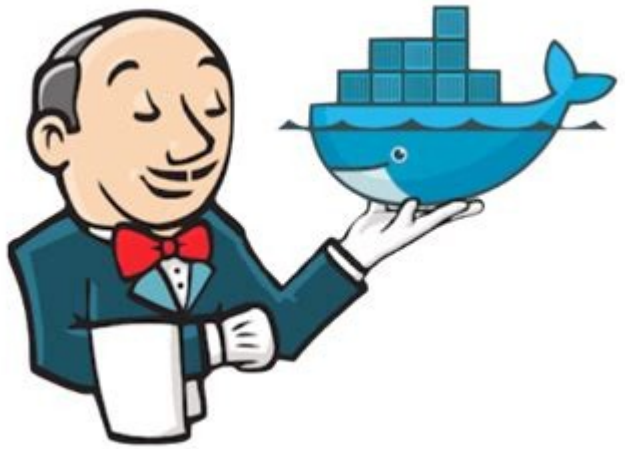
Docker: Modern Jenkins Practices

- Jenkins has evolved to support
  - Configuration-as-Code (JCasC)
  - Ephemeral agents
  - Cloud deployments
  - Kubernetes deployments



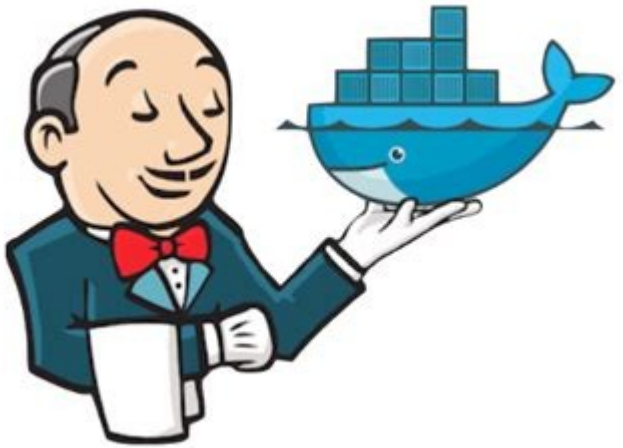
Docker: Reduced Operational Risk

- Avoids common Jenkins failures
  - Broken Java upgrades
  - OS package conflicts
  - Accidental system changes
  - Hard-to-reproduce environments
- Important because
  - Jenkins controllers are long-running
  - Downtime is disruptive
  - Recovery needs to be predictable



When to Avoid Docker

- Very high-scale enterprise setups
- Strict compliance environments
- Existing Jenkins infrastructure already standardized



Configuration-as-Code (JCasC)

- Configuration-as-Code (JCasC) is where
  - Jenkins configurations are defined in YAML files
  - Loads the configuration automatically at startup
  - Treats Jenkins configuration like application code
- Declarative
  - Describes how Jenkins should be configured
  - Replaces configuring Jenkins through the UI

```yaml
jenkins:
  systemMessage: "Welcome to the Jenkins Training Instance"

  securityRealm:
    local:
      allowsSignup: false
      users:
        - id: admin
          password: admin123

  authorizationStrategy:
    loggedInUsersCanDoAnything:
      allowAnonymousRead: false

tool:
  git:
    installations:
      - name: Default Git
        home: /usr/bin/git
```

Configuration-as-Code (JCasC)

- JcasC, can define
  - Security settings
  - Authentication and authorization
  - Users and roles
  - Credentials
  - Global tools (JDK, Maven, Node, etc.)
  - Plugin configuration
  - System settings
- But not
  - Individual pipeline logic
  - Build steps inside Jenkinsfiles
  - Pipelines are code, JCasC is configuration.

```yaml
jenkins:
  systemMessage: "Welcome to the Jenkins Training Instance"

  securityRealm:
    local:
      allowsSignup: false
      users:
        - id: admin
          password: admin123

  authorizationStrategy:
    loggedInUsersCanDoAnything:
      allowAnonymousRead: false

tool:
  git:
    installations:
      - name: Default Git
        home: /usr/bin/git
```

Configuration-as-Code (JCasC)

- JCasC YAML files are stored on disk
- Often mounted into Jenkins via:
  - Docker volumes
  - ConfigMaps (Kubernetes)
- JCasC is applied at
  - Jenkins startup
  - Configuration reload (optional)
- What Jenkins does
  - Jenkins starts
  - JCasC plugin reads YAML
  - Jenkins configures itself accordingly
  - UI reflects the defined state

```yaml
jenkins:
  systemMessage: "Welcome to the Jenkins Training Instance"

  securityRealm:
    local:
      allowsSignup: false
      users:
        - id: admin
          password: admin123

  authorizationStrategy:
    loggedInUsersCanDoAnything:
      allowAnonymousRead: false

tool:
  git:
    installations:
      - name: Default Git
        home: /usr/bin/git
```

Benefits

- Makes Jenkins reproducible
- Supports automation and version control
- Enables safer upgrades
- Helps avoid snowflake Jenkins which
  - Were configured manually over time
  - Have unique, undocumented settings
  - Cannot be easily recreated
  - No one fully understands anymore
  - Breaks when you try to upgrade or migrate
- Called a snowflake because it’s unique and melts under pressure

```
jenkins:
  systemMessage: "Welcome to the Jenkins Training Instance"

  securityRealm:
    local:
      allowsSignup: false
      users:
        - id: admin
          password: admin123

  authorizationStrategy:
    loggedInUsersCanDoAnything:
      allowAnonymousRead: false

tool:
  git:
    installations:
      - name: Default Git
        home: /usr/bin/git
```

Snowflake Instances

- Common causes
  - Clicking through the UI to configure Jenkins
  - Installing plugins “just to try something”
  - Making emergency fixes directly on the server
  - Manually editing config files
  - Installing tools directly on agents
  - No version control for configuration
- Over time
  - The original admins leave
  - Configuration knowledge is lost

```
jenkins:
  systemMessage: "Welcome to the Jenkins Training Instance"

  securityRealm:
    local:
      allowsSignup: false
      users:
        - id: admin
          password: admin123

  authorizationStrategy:
    loggedInUsersCanDoAnything:
      allowAnonymousRead: false

tool:
  git:
    installations:
      - name: Default Git
        home: /usr/bin/git
```

Questions