Jenkins + Docker + Kubernetes Environment Setup

▼ 1️⃣ Architecture Overview

Purpose

This setup provides a **local CI/CD environment** where Jenkins pipelines can build Docker images, and deploy applications to a **local Kubernetes (Kind) cluster**, using Docker Compose and an inbound Jenkins agent.

Architecture Summary

The system follows a **Jenkins controller–agent model**, fully containerized using Docker Compose.

Jenkins Controller

- Runs in a Docker container
- Responsible for:
  - Jenkins UI
  - Job scheduling
  - Pipeline orchestration
  - Agent authentication
- Exposes:
  - **8081** → **8080** (Jenkins UI)
  - **50000** → **50000** (Inbound JNLP agents)
- Persists state using a **named Docker volume** mounted at:

  ```
  /var/jenkins_home
  ```

This stores jobs, pipelines, plugins, users, and secrets across restarts.

- Connected to a Docker bridge network (`jenkins-net`) for isolation.

Jenkins Agent

- Runs in a separate Docker container
- Connects to the controller using the **inbound (JNLP) agent mechanism**
- Configured with:

  ```
  JENKINS_URL=http://localhost:8081
  JENKINS_AGENT_NAME=jenkins-agent
  JENKINS_SECRET=<agent-secret>
  ```

- Uses **`network_mode: host`**, allowing:
  - Direct access to host services
  - Access to the Kubernetes API exposed on host `localhost`
- Removed from Docker bridge networking because Docker DNS is unavailable in host network mode.

Docker Integration

- The host Docker socket is mounted into the agent:

  ```
  /var/run/docker.sock
  ```

- The `jenkins` user inside the agent is added to the Docker group
- Pipelines can:
  - Build Docker images
  - Tag and push images
  - Run Docker commands without Docker-in-Docker

Docker Registry Authentication

- Docker registry credentials (e.g., Docker Hub) are configured in **Jenkins Credentials**
- Credentials are injected into pipelines at runtime using Jenkins' credential binding mechanisms
- This enables secure authentication for:
  - `docker login`
  - Pushing images to remote registries
- No Docker-specific Jenkins plugins are required; the standard Docker CLI is sufficient

---

Kubernetes Integration

- Kubernetes cluster is provided by **Kind (Kubernetes in Docker)**
- Kind exposes the Kubernetes API on the **host loopback interface**
- The Jenkins agent:
  - Uses host networking to reach the API
  - Mounts the host kubeconfig file:

    ```
    ~/.kube/config → /home/jenkins/.kube/config (read-only)
    ```

- `kubectl` is installed in the agent image
- Jenkins pipelines can execute:

  ```
  kubectl apply
  kubectl rollout status
  ```

- Kubernetes credentials are **not stored in Jenkins**, as access is inherited from the host kubeconfig.

Networking Rationale

- **Controller** uses Docker bridge networking for isolation
- **Agent** uses host networking to:
  - Reach the Kind API on `localhost`
  - Reach the Jenkins controller via host-exposed ports
- Docker DNS names are intentionally not used by the agent

Scope & Limitations

- Optimized for:
  - Local development
  - Learning and experimentation
- Not production-ready:
  - Uses host kubeconfig
  - Uses host networking
- Production evolution would require:
  - Kubernetes ServiceAccounts
  - Jenkins-managed kubeconfig credentials
  - Routable cluster endpoints

▼ 2️⃣ `docker-compose.yaml`

🔹 Controller service

- `build: ./controller` Builds a custom Jenkins controller image using the Dockerfile in `controller/`. This allows installing the Docker CLI and adjusting Linux groups.
- Jenkins inbound (JNLP) agents require **two separate connections**:
  1. **HTTP connection (port 8080)** `8081:8080`

     Used for initial communication and controller discovery.

2. **JNLP agent connection (port 50000)** `50000:50000`

   Ensures that the Jenkins controller can accept inbound agent connections.

   Used to establish and maintain the agent execution channel.

- `jenkins-data:/var/jenkins_home` Persists Jenkins configuration, jobs, plugins, and credentials across container restarts.
- `/var/run/docker.sock:/var/run/docker.sock` Allows Jenkins to communicate with the **host Docker daemon**. This enables building and running Docker images from pipelines.
- `networks: jenkins-net` Places the controller on a dedicated Docker bridge network so agents can reach it by name.

🔹 Agent service

- `build.context: ./agent` Builds a dedicated Jenkins agent image with build tools (Maven, Docker CLI).
- `DOCKER_GID` Must match the **host Docker group GID** (retrieved running `getent group docker` on host).This ensures non-root Docker access inside the container.
- `depends_on` Ensures the controller container starts before the agent.
- `JENKINS_URL`

  ~~`JENKINS_URL=http://jenkins-controller:8080`~~ This URL relied on **Docker bridge networking and Docker DNS.**

  However, the Jenkins agent was later configured to run with: `network_mode: host`

  Updated configuration: `JENKINS_URL=http://localhost:8081`

  - The agent connects to the Jenkins controller via the **host network.**
  - `localhost` correctly resolves to the host when using host networking.
  - Port `8081` maps to the controller's internal port `8080` via Docker port publishing.
- `JENKINS_AGENT_NAME` Must match the node name configured in Jenkins UI.

- `JENKINS_SECRET` Authentication token generated by Jenkins for inbound agent connections.
- `/var/run/docker.sock` Allows the agent to execute Docker commands against the host daemon.
- `~/.kube/config` Mounts the Kubernetes **kubeconfig file from the host machine** into the Jenkins agent container.
- `network_mode: host`

  Justification:

  The Jenkins agent is configured to use **host network mode** to enable direct access to the local Kubernetes API when deploying to a **Kind (Kubernetes in Docker)** cluster.

  Kind exposes the Kubernetes API server on the host machine's loopback interface (`127.0.0.1:<port>`).

  Under normal Docker bridge networking, containers have their **own isolated network namespace**, meaning `127.0.0.1` inside a container refers to the container itself, not the host. As a result, `kubectl` inside the Jenkins agent cannot reach the Kubernetes API using the default kubeconfig.

  By enabling `network_mode: host`:

  - The Jenkins agent shares the host's network namespace
  - `127.0.0.1` inside the agent resolves to the host loopback interface
  - The Kubernetes API exposed by Kind becomes reachable without modifying the kubeconfig
  - `kubectl apply` and other deployment commands can be executed successfully from Jenkins pipelines

```
services:
  jenkins-controller:
    build: ./controller
    container_name: jenkins-controller
    ports:
      - "8081:8080"
      - "50000:50000"
    volumes:
      - jenkins-data:/var/jenkins_home
      - /var/run/docker.sock:/var/run/docker.sock
    networks:
      - jenkins-net

  jenkins-agent:
    build:
      context: ./agent
      args:
        DOCKER_GID: 1001   # must match host docker group
GID
    container_name: jenkins-agent
    depends_on:
      - jenkins-controller
    environment:
      - JENKINS_URL=http://jenkins-controller:8080
      - JENKINS_AGENT_NAME=docker-agent
      - JENKINS_SECRET=30b6da28714e95752404273bddec5c2b9ce
80fb8bf5ca2f22484e6b3ea00277a
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ~/.kube/config:/home/jenkins/.kube/config:ro
    #networks:
    #  - jenkins-net
    network_mode: host

volumes:
  jenkins-data:

networks:
  jenkins-net:
    driver: bridge
```

▼ 3️⃣ Controller `Dockerfile`

- The controller image installs **only minimal tooling** (Docker CLI and certificates).
- Build tools are intentionally excluded to keep the controller lightweight and focused on orchestration.
- The Docker socket is owned by a group with a host-specific GID. Linux permissions are enforced by numeric IDs, not group names. Therefore, the Docker group inside the container must be modified to use the **same GID as the host**.

```
# Use official Jenkins LTS image as controller base
FROM jenkins/jenkins:lts

# Temporarily switch to root to install OS packages
USER root

# Install minimal required packages
RUN apt-get update && \
    apt-get install -y \
        ca-certificates \
        curl \
        gnupg \
        lsb-release

# Install Docker CLI
RUN curl -fsSL https://get.docker.com | sh

# Create docker group if it does not exist. Otherwise, MOD
IFY its GID to match the host. Then add jenkins to it
# IMPORTANT: match docker group GID with host
ARG DOCKER_GID=1001
RUN if getent group docker; then \
        groupmod -g ${DOCKER_GID} docker; \
    else \
        groupadd -g ${DOCKER_GID} docker; \
    fi && \
    usermod -aG docker jenkins

# Switch back to jenkins user
USER jenkins
```

▼ 4️⃣ Agent `Dockerfile`

“Why the agent has more tools than the controller”

The agent image contains:

- Maven (for builds)
- Docker CLI (for image build and run)

This separation follows Jenkins best practices:

- Controller = orchestration
- Agent = execution

The agent runs as the **jenkins user**, not root, to follow the principle of least privilege.

```
FROM jenkins/inbound-agent:latest

# Switch to root to install packages
USER root

# Install required tools
RUN apt-get update && \
    apt-get install -y \
        ca-certificates \
        curl \
        gnupg \
        lsb-release \
        maven
```

```
# Install latest Docker CLI (not daemon)
RUN curl -fsSL https://get.docker.com | sh

# Install kubectl
RUN curl -LO https://dl.k8s.io/release/$(curl -L -s http
s://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl
&& \
    chmod +x kubectl && \
    mv kubectl /usr/local/bin/

# Create docker group if it does not exist. Otherwise, MOD
IFY its GID to match the host. Then add jenkins to it
# IMPORTANT: match docker group GID with host
ARG DOCKER_GID=1001
RUN if getent group docker; then \
        groupmod -g ${DOCKER_GID} docker; \
    else \
        groupadd -g ${DOCKER_GID} docker; \
    fi && \
    usermod -aG docker jenkins

# Switch back to jenkins user
USER jenkins
```

▼ 5️⃣ Startup & Verification Steps

1- Build both controller and agent images using the custom Dockerfiles. Then, Start the Jenkins controller and agent containers in detached mode.

```
docker compose build
docker compose up -d
docker ps
```

2- Open Jenkins in browser.

```
http://localhost:8081
```

3- Retrieve the one-time administrator password generated on first startup.

```
docker exec -it jenkins-controller cat /var/jenkins_home/s
ecrets/initialAdminPassword
```

4- Install suggested plugins.

5- Create admin account (username: `admin`, password: `admin`)

6- Verify docker is working in agent:

```
docker exec -it jenkins-agent docker ps
```

7- Verify kubectl is working in agent:

```
docker exec -it jenkins-agent kubectl version --client
```

8- Verify kubeconfig inside the agent.

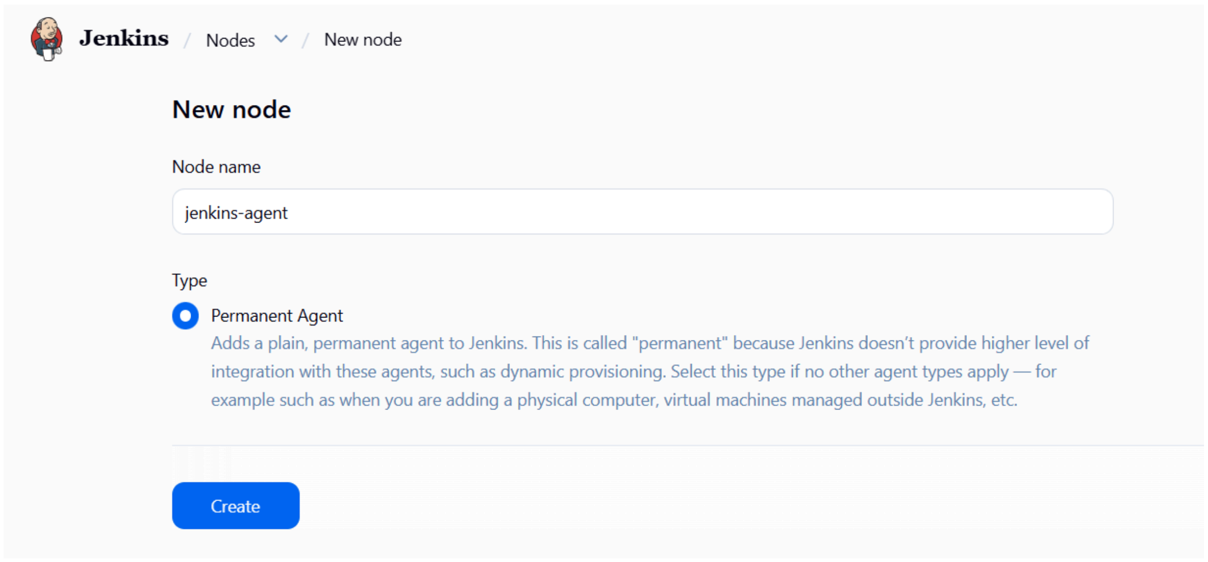
```
docker exec -it jenkins-agent ls -l /home/jenkins/.kube
```

9- Verify cluster access from the agent.

```
docker exec -it jenkins-agent kubectl get nodes
```

▼ 6️⃣ Configure Jenkins Agent

1- Configure a new node on Jenkins UI

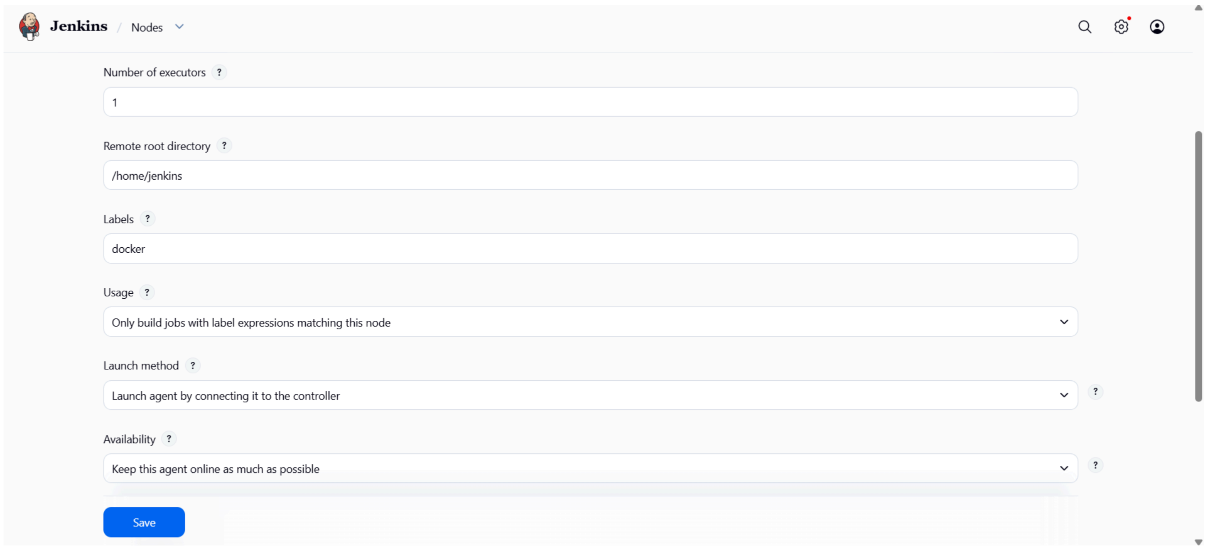


New node

Node name: jenkins-agent

Type

Permanent Agent
Adds a plain, permanent agent to Jenkins. This is called "permanent" because Jenkins doesn't provide higher level of integration with these agents, such as dynamic provisioning. Select this type if no other agent types apply — for example such as when you are adding a physical computer, virtual machines managed outside Jenkins, etc.



Number of executors: 1

Remote root directory: /home/jenkins

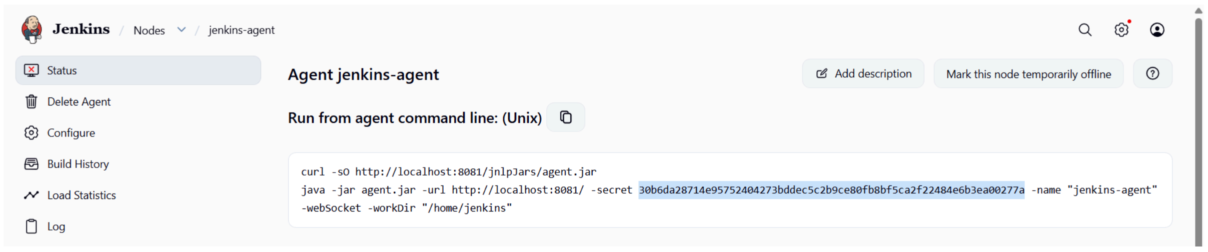
Labels: docker

Usage: Only build jobs with label expressions matching this node

Launch method: Launch agent by connecting it to the controller

Availability: Keep this agent online as much as possible

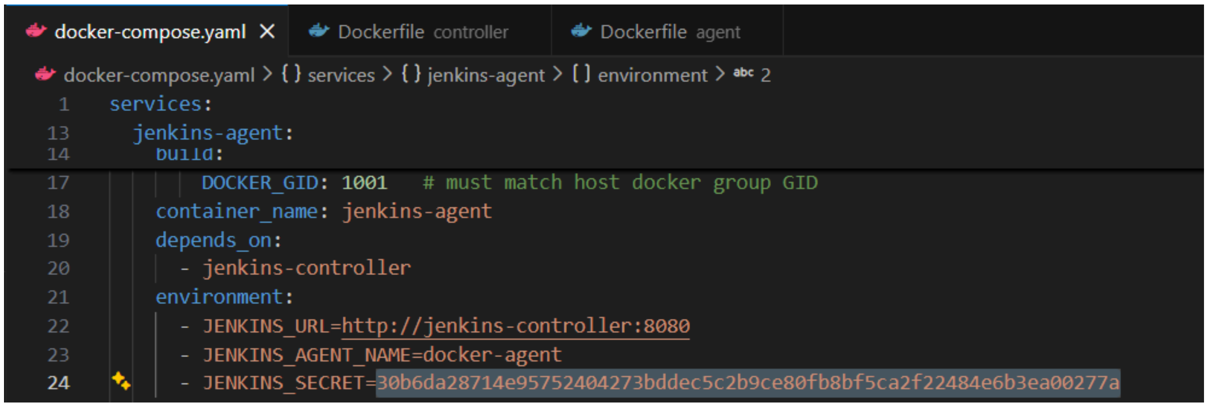
2- Add agent secret to docker compose file then, restart only the agent:



Agent jenkins-agent

Run from agent command line: (Unix)

```
curl -sO http://localhost:8081/jnlpJars/agent.jar
java -jar agent.jar -url http://localhost:8081/ -secret 30b6da28714e95752404273bddec5c2b9ce80fb8bf5ca2f22484e6b3ea00277a -name "jenkins-agent"
-webSocket -workDir "/home/jenkins"
```


docker-compose.yaml ✕ | Dockerfile controller | Dockerfile agent

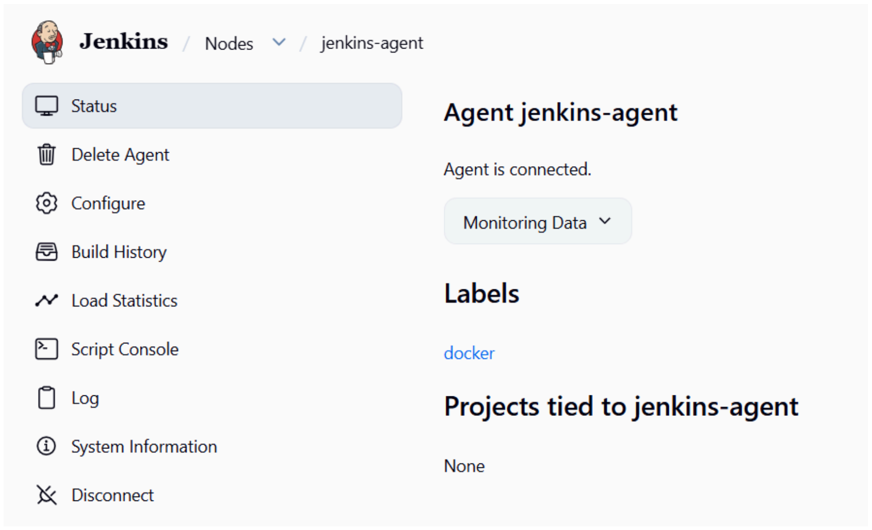
docker-compose.yaml > { } services > { } jenkins-agent > [ ] environment > abc 2

```yaml
services:
  jenkins-agent:
    build:
        DOCKER_GID: 1001    # must match host docker group GID
    container_name: jenkins-agent
    depends_on:
      - jenkins-controller
    environment:
      - JENKINS_URL=http://jenkins-controller:8080
      - JENKINS_AGENT_NAME=docker-agent
      - JENKINS_SECRET=30b6da28714e95752404273bddec5c2b9ce80fb8bf5ca2f22484e6b3ea00277a
```

```
docker compose up -d --no-deps jenkins-agent
```

3- Confirm agent is connected :

```
Manage Jenkins → Nodes
```


Jenkins / Nodes / jenkins-agent

Agent jenkins-agent

Agent is connected.

Monitoring Data

Labels

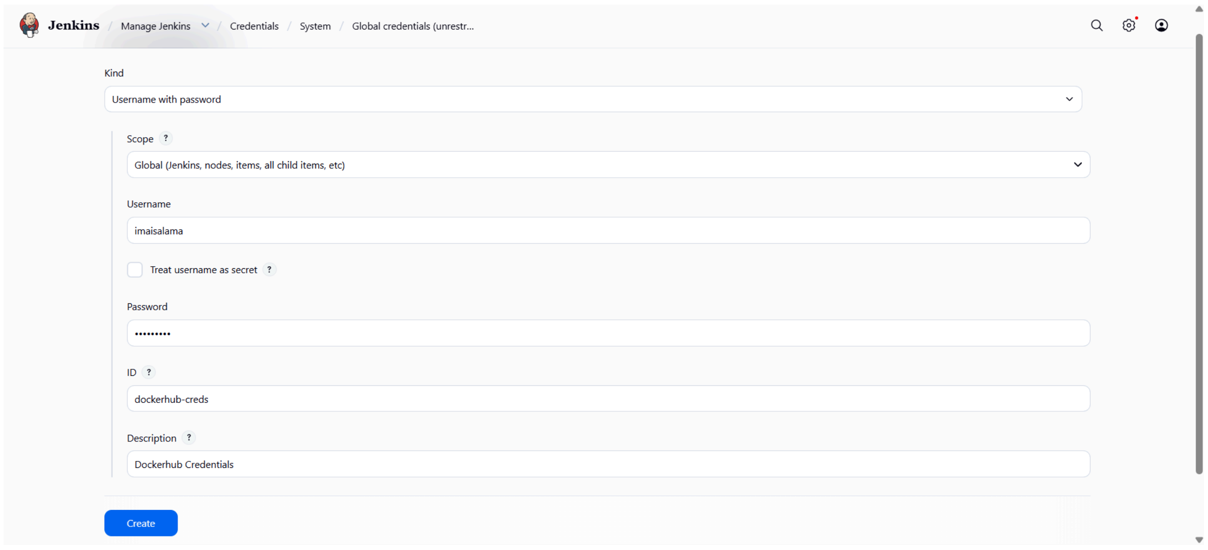
docker

Projects tied to jenkins-agent

None

▼ 7️⃣ Configuring Docker Hub Credentials

Configure Dockerhub Credentials


Jenkins / Manage Jenkins / Credentials / System / Global credentials (unrestr...

Kind
Username with password

Scope ?
Global (Jenkins, nodes, items, all child items, etc)

Username
imaisalama

Treat username as secret ?

Password
•••••••••

ID ?
dockerhub-creds

Description ?
Dockerhub Credentials

▼ 7️⃣ Final assessment

- Implemented a complete local CI/CD pipeline using **Jenkins, Docker, and Kubernetes** with a containerized controller–agent architecture.
- Configured a dedicated Jenkins agent with access to the **host Docker daemon** and Kubernetes tooling to execute build and deployment tasks.
- Enabled Kubernetes deployments to a local **Kind cluster** by using host networking and a mounted kubeconfig file.
- Ensured reliable agent connectivity by exposing the Jenkins UI and **JNLP agent port (50000)** and aligning controller URLs with host networking.
- Delivered a lightweight, minimal-plugin setup optimized for **local development and learning**, with a clear path to production-grade CI/CD enhancements.
