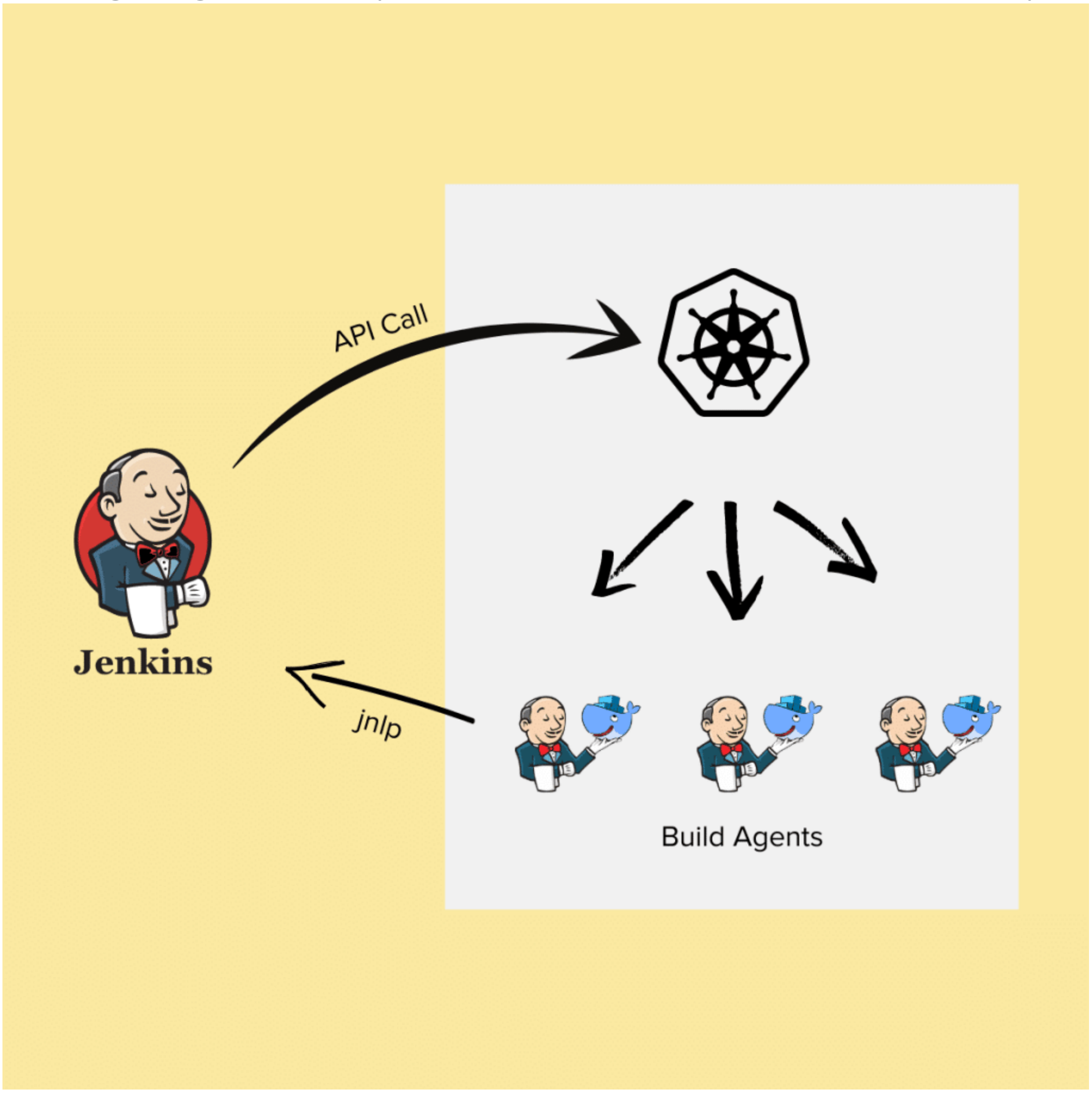
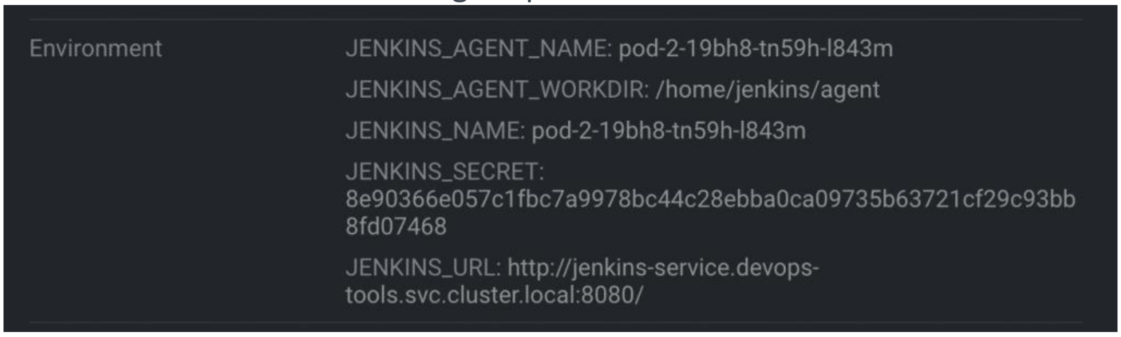
How Do Jenkins Kubernetes Pod Agents Work?

Before getting into the implementation, let’s understand how this setup works.



Whenever you trigger a Jenkins job, the Jenkins Kubernetes plugin will make an API call to create a Kubernetes agent pod.
Then, the Jenkins agent pod gets deployed in the kubernetes with a few environment variables containing the Jenkins server details and secrets.
When the agent pod comes up, it uses the details in its environment variables and talks back to Jenkins using the JNLP method. The following images show the environment variables of the agent pod.



```
Environment          JENKINS_AGENT_NAME: pod-2-19bh8-tn59h-l843m
                     JENKINS_AGENT_WORKDIR: /home/jenkins/agent
                     JENKINS_NAME: pod-2-19bh8-tn59h-l843m
                     JENKINS_SECRET:
                     8e90366e057c1fbc7a9978bc44c28ebba0ca09735b63721cf29c93bb
                     8fd07468
                     JENKINS_URL: http://jenkins-service.devops-
                     tools.svc.cluster.local:8080/
```

All the build steps from the Jenkinsfile run on that pod. Once the build is complete, the pod will get terminated automatically. There are also options to retain the build pod.
The Jenkins Kubernetes plugin takes care of all the communication from Jenkins to the Kubernetes cluster.
Also, as long as your Kubernetes cluster scales, you can scale your Jenkins build agents without any issues.

Setting Up Jenkins Build Pods On Kubernetes

To work on this setup we need the following.
A working Kubernetes cluster.
Kubernetes admin user to create Kubernetes deployments and service accounts
A running Jenkins master
Also, I am considering two scenarios here.
Jenkins master running inside the Kubernetes cluster.
Jenkins master running outside the Kubernetes cluster.
We will look at both scenarios and their configurations.
Overfall, here is what we are going to do.
Create a namespace `devops-tools`
Create a Kubernetes service account named `jenkins-admin` with permissions to manage pods in `devops-tools` namespace. This service account will be used by Jenkins to deploy the agent pods. (Both internal & external Jenkins)
Deploy Jenkins in `devops-tools` namespace with the `jenkins-admin` service account. (If you don't have an existing Jenkins)
Configure Kubernetes Jenkins Plugin for Jenkins to interact with Kubernetes cluster and deploy build agents.

Setting Up Kubernetes Namespace & Service Account

Let's get started with the setup.
Step 1: Create a namespace called `devops-tools`

```
kubectl create namespace devops-tools
```

Step 2: Save the following manifest as `service-account.yaml`. It contains the role and role-binding for the service account with all the permission to manage pods in the `devops-tools` namespace.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: jenkins-admin
  namespace: devops-tools
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: jenkins
  namespace: devops-tools
  labels:
    "app.kubernetes.io/name": 'jenkins'
rules:
- apiGroups: [""]
```

```
  resources: ["pods"]
  verbs: ["create","delete","get","list","patch","update","watch"]
- apiGroups: [""]
  resources: ["pods/exec"]
  verbs: ["create","delete","get","list","patch","update","watch"]
- apiGroups: [""]
  resources: ["pods/log"]
  verbs: ["get","list","watch"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: jenkins-role-binding
  namespace: devops-tools
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: jenkins
subjects:
- kind: ServiceAccount
  name: jenkins-admin
  namespace: devops-tools
```

Create the service account.

```
kubectl apply -f service-account.yaml
```

Jenkins Master Setup in Kubernetes

In this setup, we will have both the Jenkins master and agents deploying in the same Kubernetes cluster.

We will set up the Jenkins master server on the Kubernetes cluster.

Note: If you have an existing setup, you can use that as well. Ensure it has a service account with permissions to deploy pods in the namespace where Jenkins is deployed.

Save the following manifest as `deployment.yaml`. This manifest contains persistent volume, deployment, and service definitions.

Note: Ensure that your Kubernetes cluster setup supports persistent volumes. If you deploy Jenkins without a persistent volume, you will lose the Jenkins data on every restart or pod deletion.

```
# Persistent Volume Claim
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: jenkins-pv-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 50Gi

# Deployment Config
---
apiVersion: apps/v1
kind: Deployment
metadata:
```

```yaml
  name: jenkins-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jenkins
  template:
    metadata:
      labels:
        app: jenkins
    spec:
      serviceAccountName: jenkins-admin
      securityContext:
            fsGroup: 1000
            runAsUser: 1000
      containers:
        - name: jenkins
          image: jenkins/jenkins:lts
          resources:
            limits:
              memory: "2Gi"
              cpu: "1000m"
            requests:
              memory: "500Mi"
              cpu: "500m"
          ports:
            - name: httpport
              containerPort: 8080
            - name: jnlpport
              containerPort: 50000
          livenessProbe:
            httpGet:
              path: "/login"
              port: 8080
            initialDelaySeconds: 90
            periodSeconds: 10
            timeoutSeconds: 5
            failureThreshold: 5
          readinessProbe:
            httpGet:
              path: "/login"
              port: 8080
            initialDelaySeconds: 60
            periodSeconds: 10
            timeoutSeconds: 5
            failureThreshold: 3
          volumeMounts:
            - name: jenkins-data
              mountPath: /var/jenkins_home
      volumes:
        - name: jenkins-data
          persistentVolumeClaim:
              claimName: jenkins-pv-claim

# Service Config
---
apiVersion: v1
kind: Service
metadata:
  name: jenkins-service
  annotations:
      prometheus.io/scrape: 'true'
      prometheus.io/path:   /
      prometheus.io/port:   '8080'
spec:
  selector:
    app: jenkins
  type: NodePort
  ports:
    - name: httpport
      port: 8080
      targetPort: 8080
      nodePort: 32000
    - name: jnlpport
      port: 50000
      targetPort: 50000
```

Create the deployment.

```
kubectl apply -f deployment.yaml
```

After a couple of minutes, the Jenkins deployment will be up and you will be able to access any Kubernetes node on port `32000`

Step 4: Access the Jenkins dashboard over the node port and unlock it using the password from the pod logs. Install the suggested plugins and create a Jenkins user.

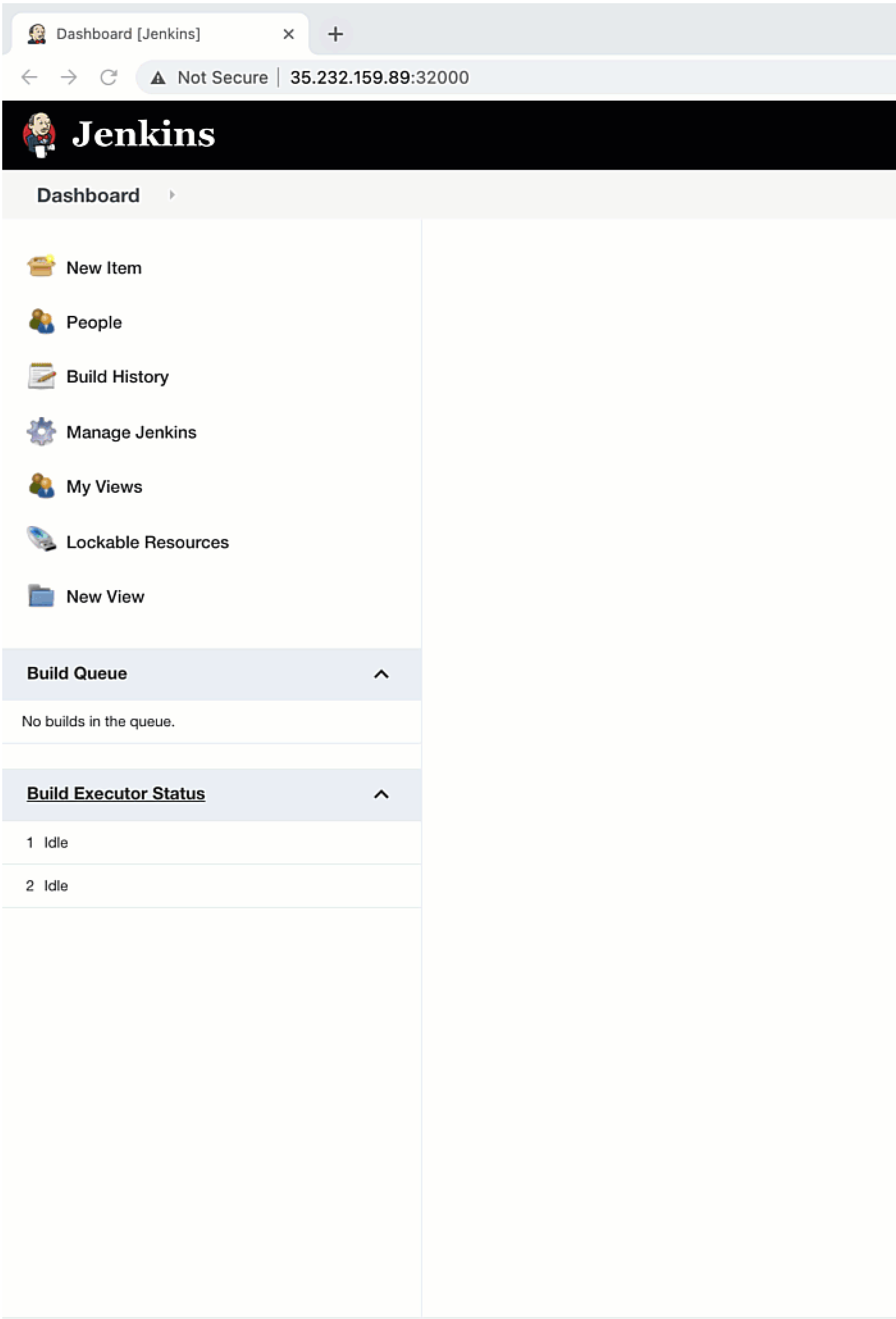
Please follow the Jenkins on Kubernetes blog if you have any doubts.

Jenkins Kubernetes Plugin Configuration

Jenkins Kubernetes plugin is required to set up Kubernetes-based build agents. Let's configure the plugin.

Step 1: Install Jenkins Kubernetes Plugin

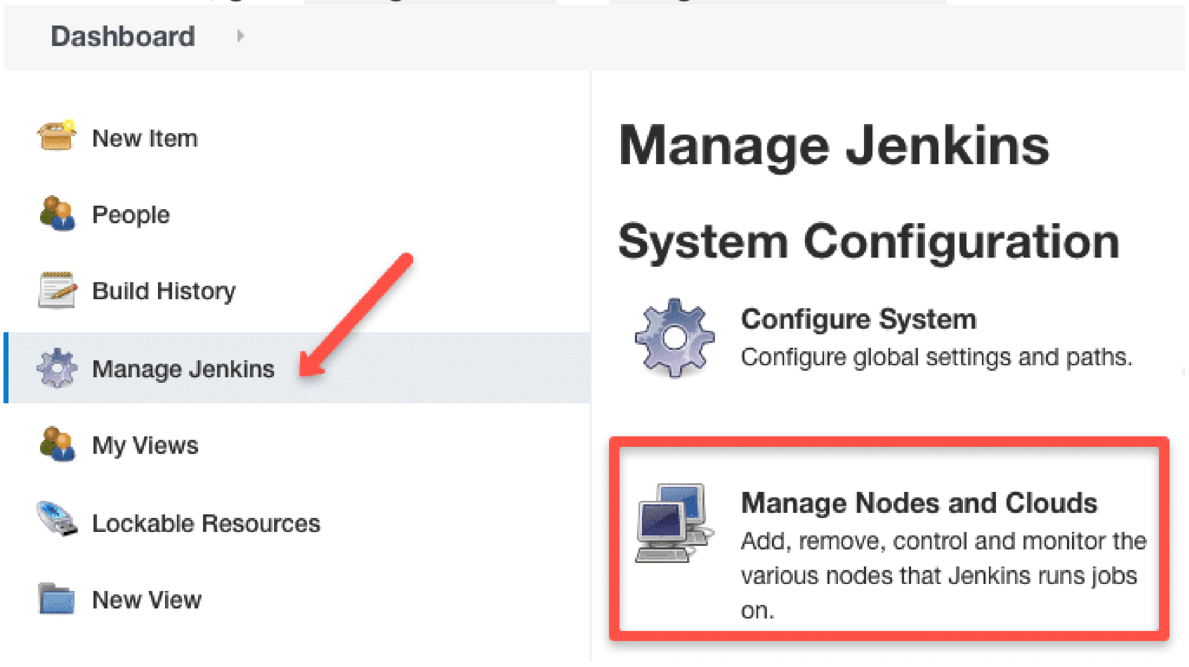
Go to `Manage Jenkins` –> `Manage Plugins`, search for Kubernetes Plugin in the available tab, and install it. The following Gif video shows the plugin installation process.



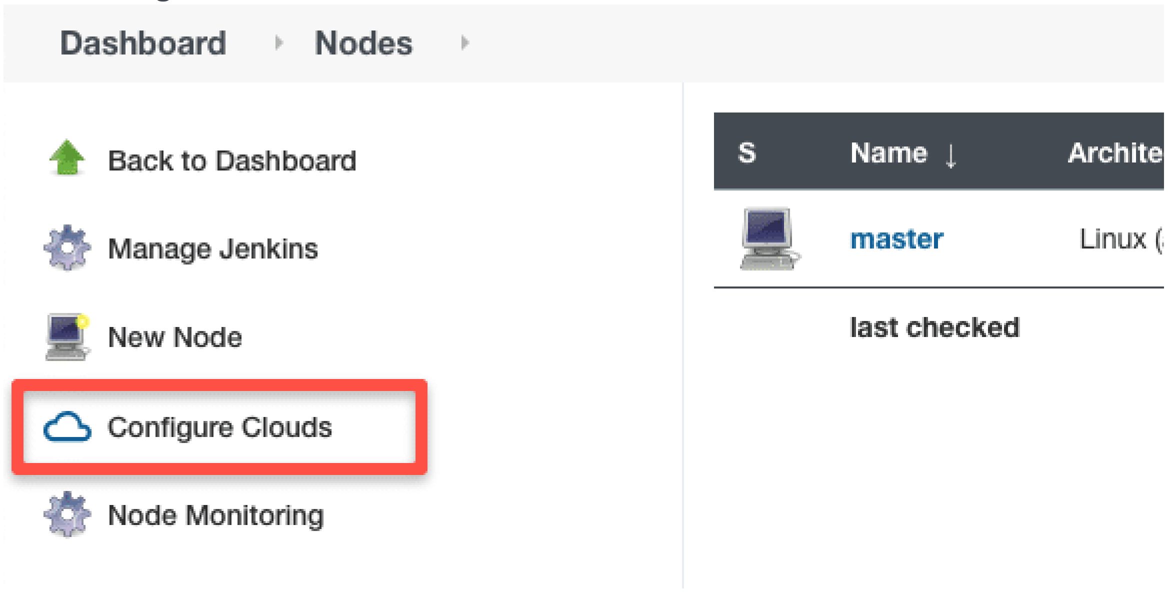
No builds in the queue.

Step 2: Create a Kubernetes Cloud Configuration

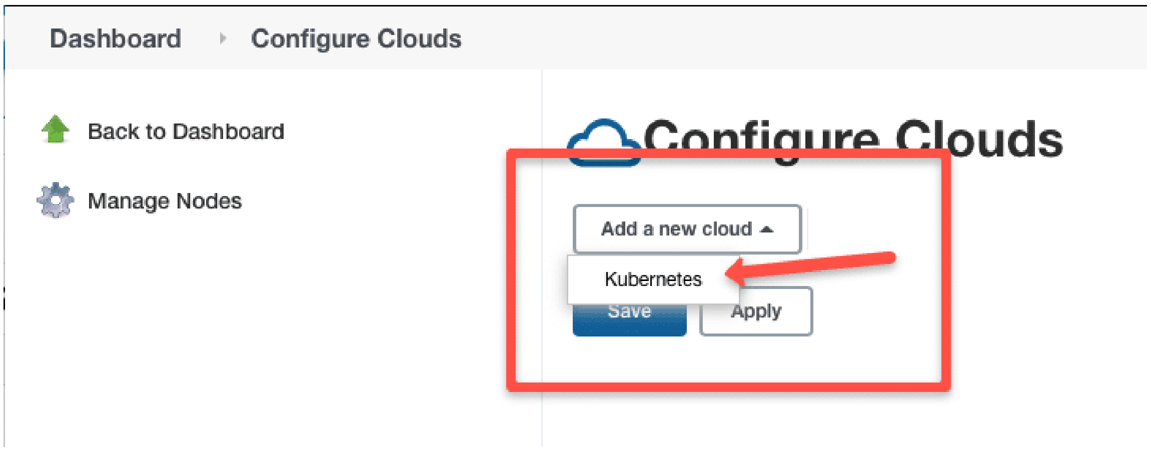
Once installed, go to `Manage Jenkins` –> `Manage Node & Clouds`



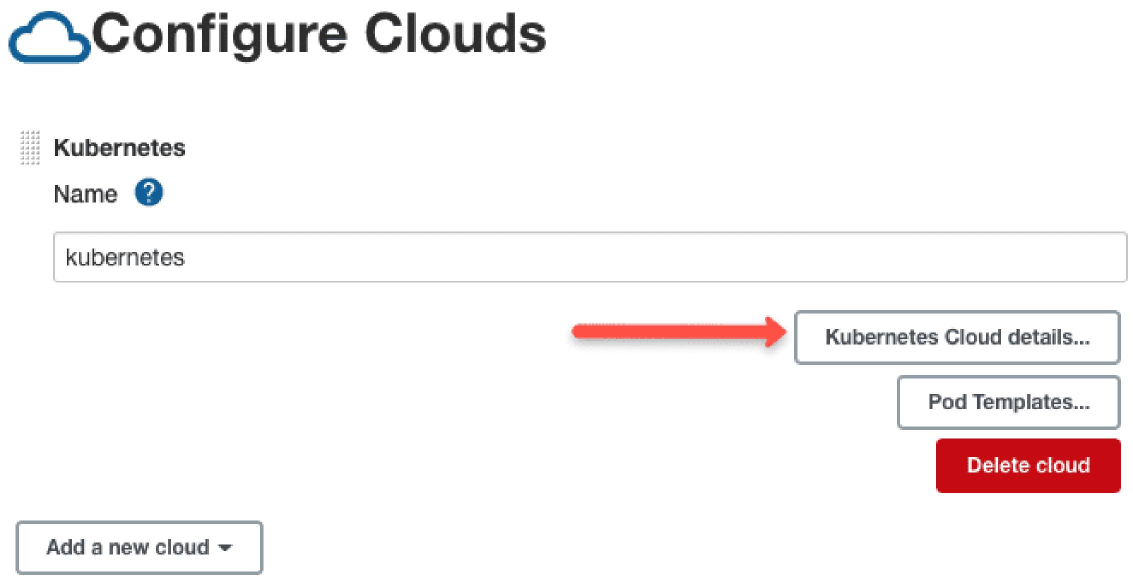
Click Configure Clouds



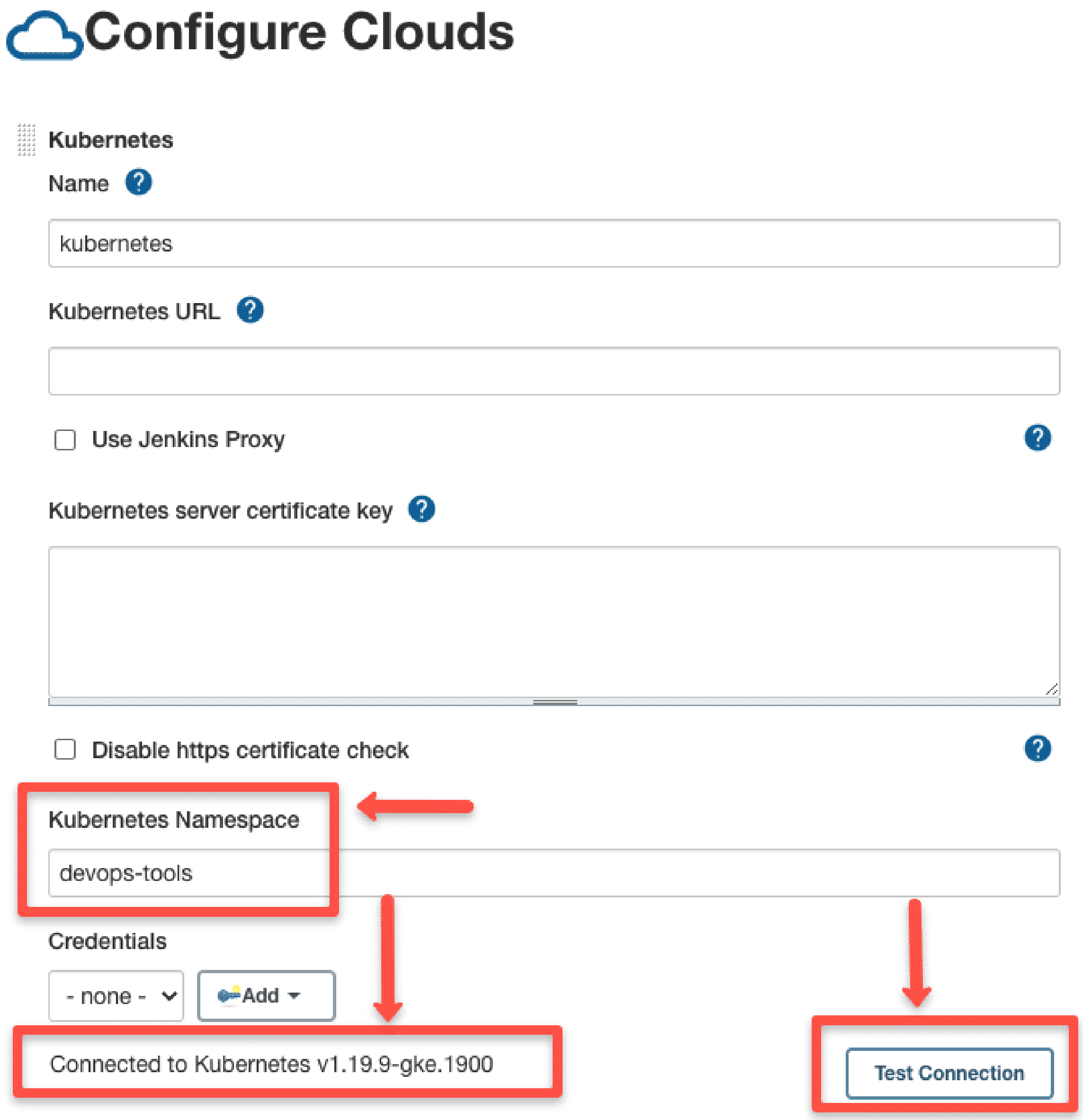
"Add a new Cloud" select Kubernetes.



Select Kubernetes Cloud Details



Step 3: Configure Jenkins Kubernetes Cloud
Here we have two scenarios.
Jenkins server running inside the same Kubernetes cluster
Jenkins server running out of the Kubernetes cluster.
Let’s look at configurations for both scenarios.
Jenkins server running inside the same Kubernetes cluster
Since we have Jenkins inside the Kubernetes cluster with a service account to deploy the agent pods, we don’t have to mention the Kubernetes URL or certificate key.
However, to validate the connection using the service account, use the Test connection button as shown below. It should show a connected message if the Jenkins pod can connect to the Kubernetes master API


Configure Clouds

Kubernetes

Name

kubernetes

Kubernetes URL

Use Jenkins Proxy

Kubernetes server certificate key

Disable https certificate check

Kubernetes Namespace

devops-tools

Credentials

- none -

Connected to Kubernetes v1.19.9-gke.1900

Jenkins server running Outside the Kubernetes cluster

If your Jenkins server is running outside the Kubernetes cluster, you need to specify the following.

Kubernetes URL: This is the Kubernetes master API endpoint. If is it https enabled, use the https url.

Kubernetes Server Certificate key: If you have a Kubernetes Cluster CA certificate, you can add it for secure connectivity. You can get the certificate from the pod location `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt` .If you do not have the certificate, you can enable the "`disable https certificate check`" option.

Credentials: For Jenkins to communicate with the Kubernetes cluster, we need a service account token with permission to deploy pods in the `devops-tools` namespace.

Note: If you use managed services like GKE cluster, you can get all the cluser details from the GKE dashboard.

We have already created the service account in the `devops-tools` namespace. We need to get the token from the service account.

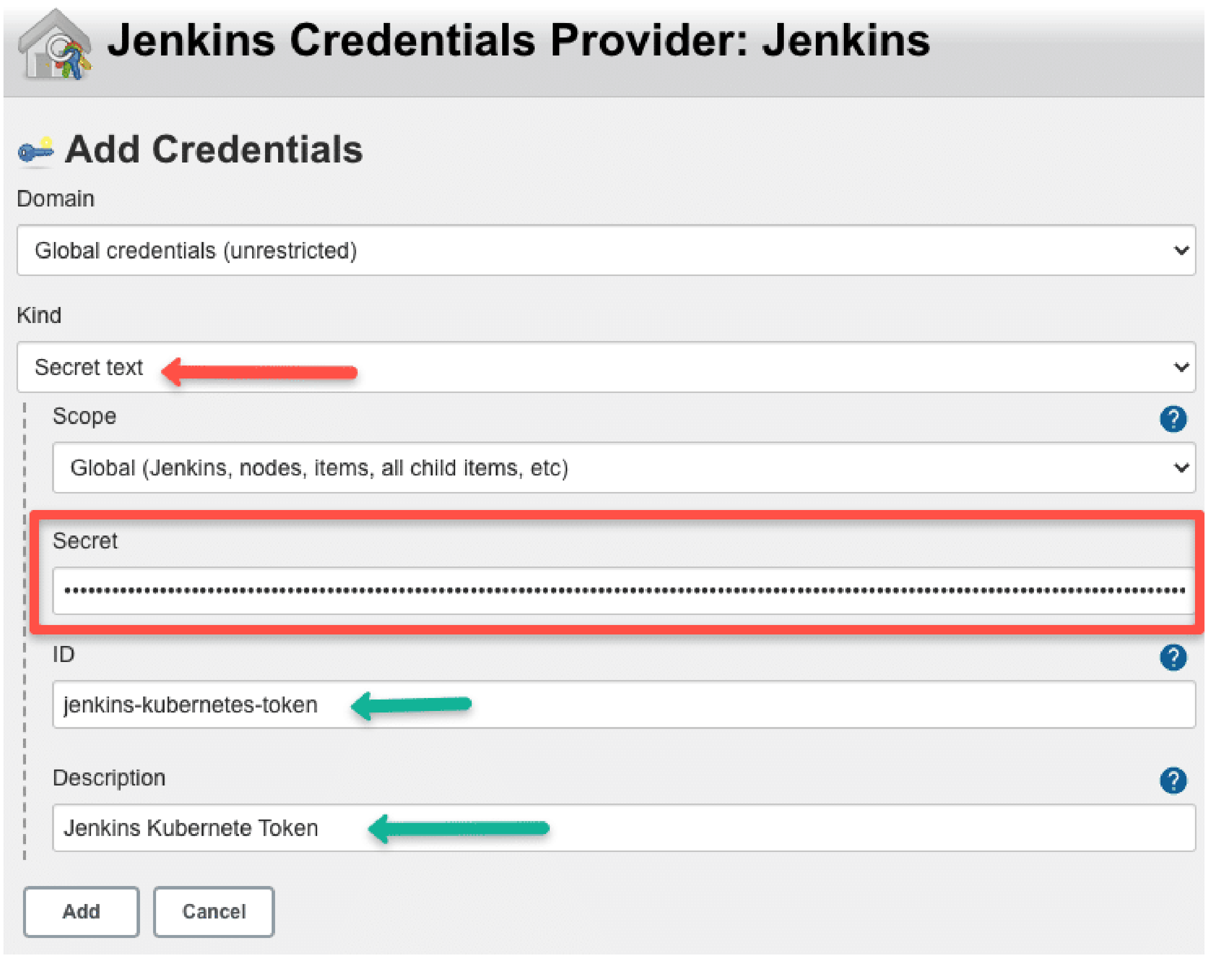
Execute the following commands to retrieve the secret name from the service account.

```
SECRET_NAME=$(kubectl get serviceaccount jenkins-admin  -o=jsonpath='{.secrets[0].name}' -n devops-tools)
```

Now, we will use the SECRET_NAME to get the base64 encoded service account token and then decode it. You will get the token as an output.

```
kubectl get secrets $SECRET_NAME  -o=jsonpath='{.data.token}' -n devops-tools | base64 -D
```

Now click the `Add` button under credentials and create a credential type "`Secret text`". Enter the service account token in the secret box and add other details as shown below. Finally, save the credential.



Jenkins Credentials Provider: Jenkins

Add Credentials

Domain: Global credentials (unrestricted)

Kind: Secret text

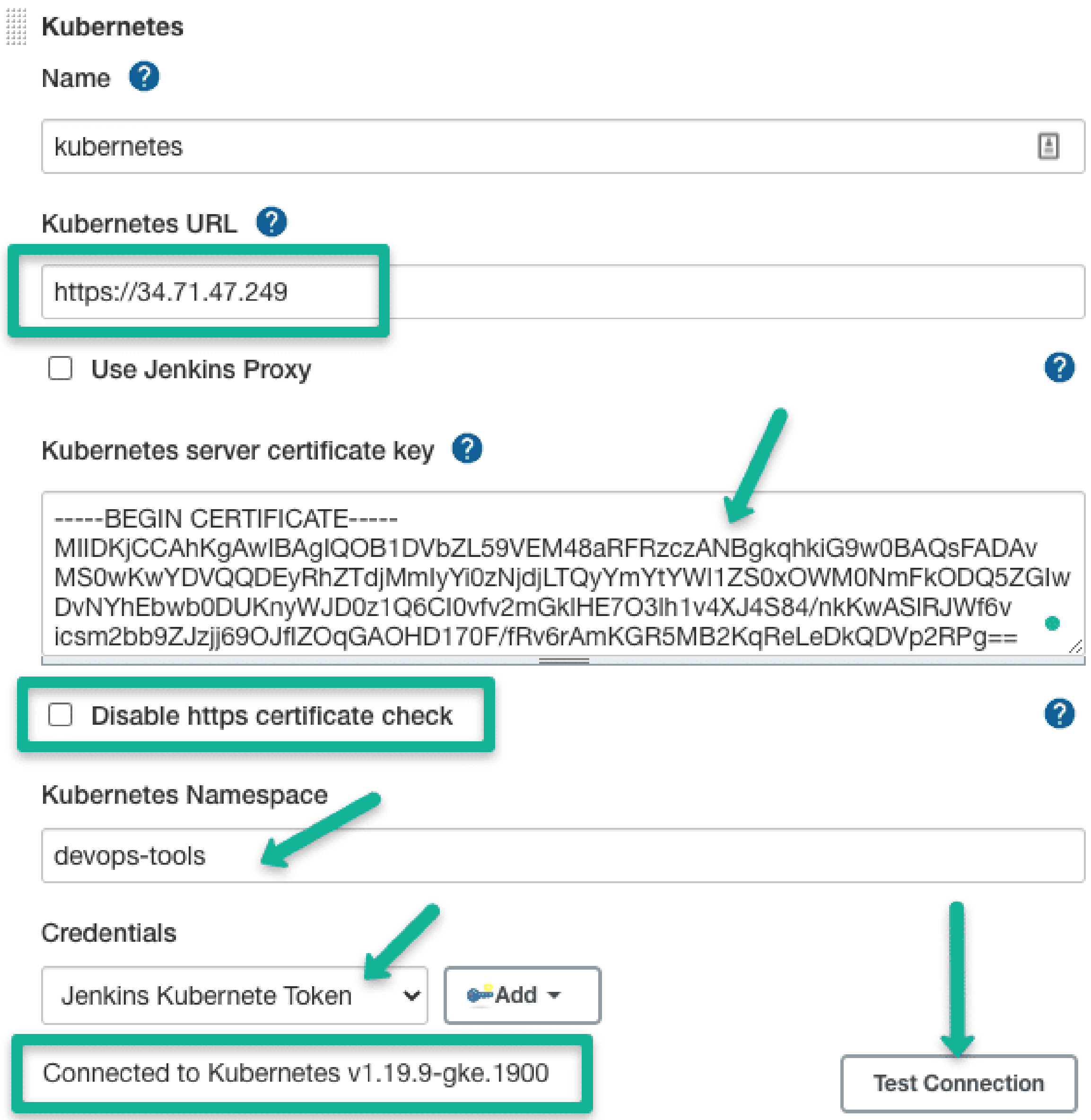
Scope: Global (Jenkins, nodes, items, all child items, etc)

Secret: ••

ID: jenkins-kubernetes-token

Description: Jenkins Kubernete Token

The kubernetes cloud configuration would look like the following.



Kubernetes

Name

kubernetes

Kubernetes URL

https://34.71.47.249

☐ Use Jenkins Proxy

Kubernetes server certificate key

```
-----BEGIN CERTIFICATE-----
MIIDKjCCAhKgAwIBAgIQOB1DVbZL59VEM48aRFRzczANBgkqhkiG9w0BAQsFADAv
MS0wKwYDVQQDEyRhZTdjMmIyYi0zNjdjLTQyYmYtYWI1ZS0xOWM0NmFkODQ5ZGIw
DvNYhEbwb0DUKnyWJD0z1Q6CI0vfv2mGkIHE7O3Ih1v4XJ4S84/nkKwASIRJWf6v
icsm2bb9ZJzjj69OJfIZOqGAOHD170F/fRv6rAmKGR5MB2KqReLeDkQDVp2RPg==
```

☐ Disable https certificate check

Kubernetes Namespace

devops-tools

Credentials

Jenkins Kubernete Token

Connected to Kubernetes v1.19.9-gke.1900

After filling in all the details, you can test the connection to validate the Kubernetes cluster connectivity.

Step 4: Configure the Jenkins URL Details

For Jenkins master running inside the cluster, you can use the Service endpoint of the Kubernetes cluster as the Jenkins URL because agents pods can connect to the cluster via internal service DNS.

The URL is derived using the following syntax.

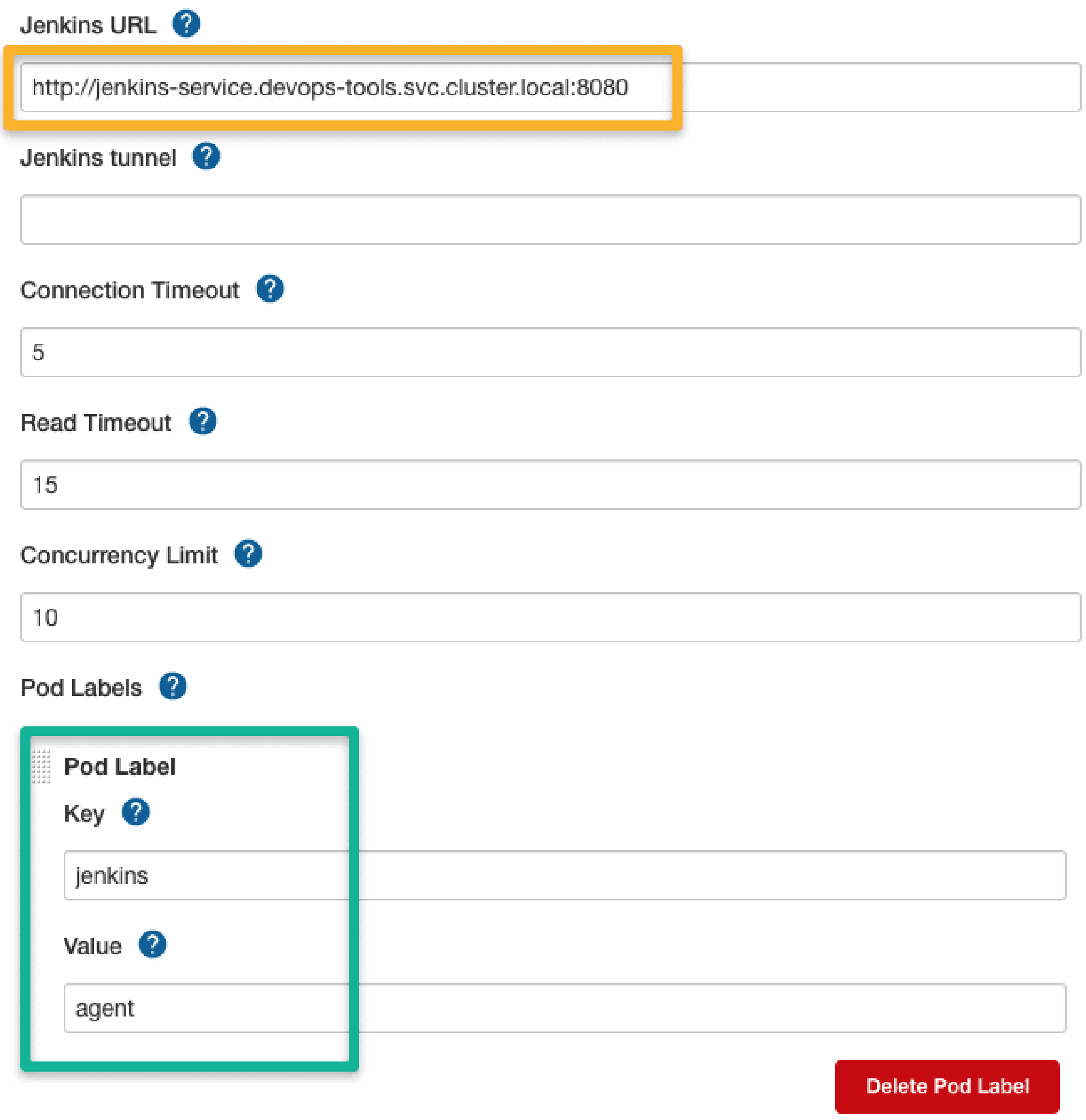
```
http://<service-name>.<namespace>.svc.cluster.local:8080
```

In our case, the service DNS will be,

```
http://jenkins-service.devops-tools.svc.cluster.local:8080
```

Also, add the POD label that can be used for grouping the containers if required in terms of bulling or custom build dashboards.

Note: If the Jenkins master is outside the Kubernetes cluster, use the Jenkins IP or DNS in the Jenkins URL configuration.



Jenkins URL

http://jenkins-service.devops-tools.svc.cluster.local:8080

Jenkins tunnel

Connection Timeout

5

Read Timeout

15

Concurrency Limit

10

Pod Labels

Pod Label

Key

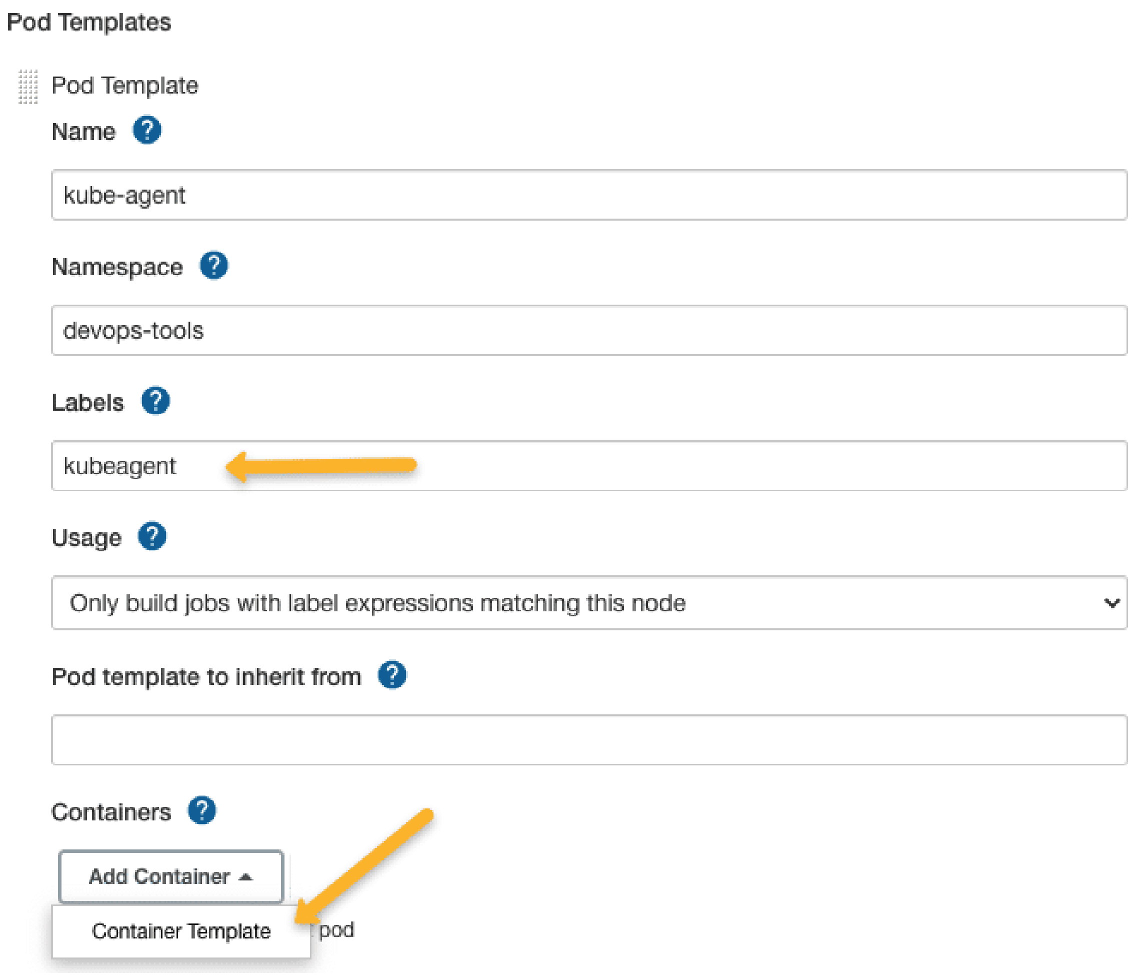
jenkins

Value

agent

Step 5: Create POD and Container Template

Next, you need to add the POD template with the details, as shown in the image below. The label `kubeagent` will be used in the job as an identifier to pick this pod as the build agent. Next, we need to add a container template with the Docker image details.


Pod Templates

Pod Template

Name: kube-agent

Namespace: devops-tools

Labels: kubeagent

Usage: Only build jobs with label expressions matching this node

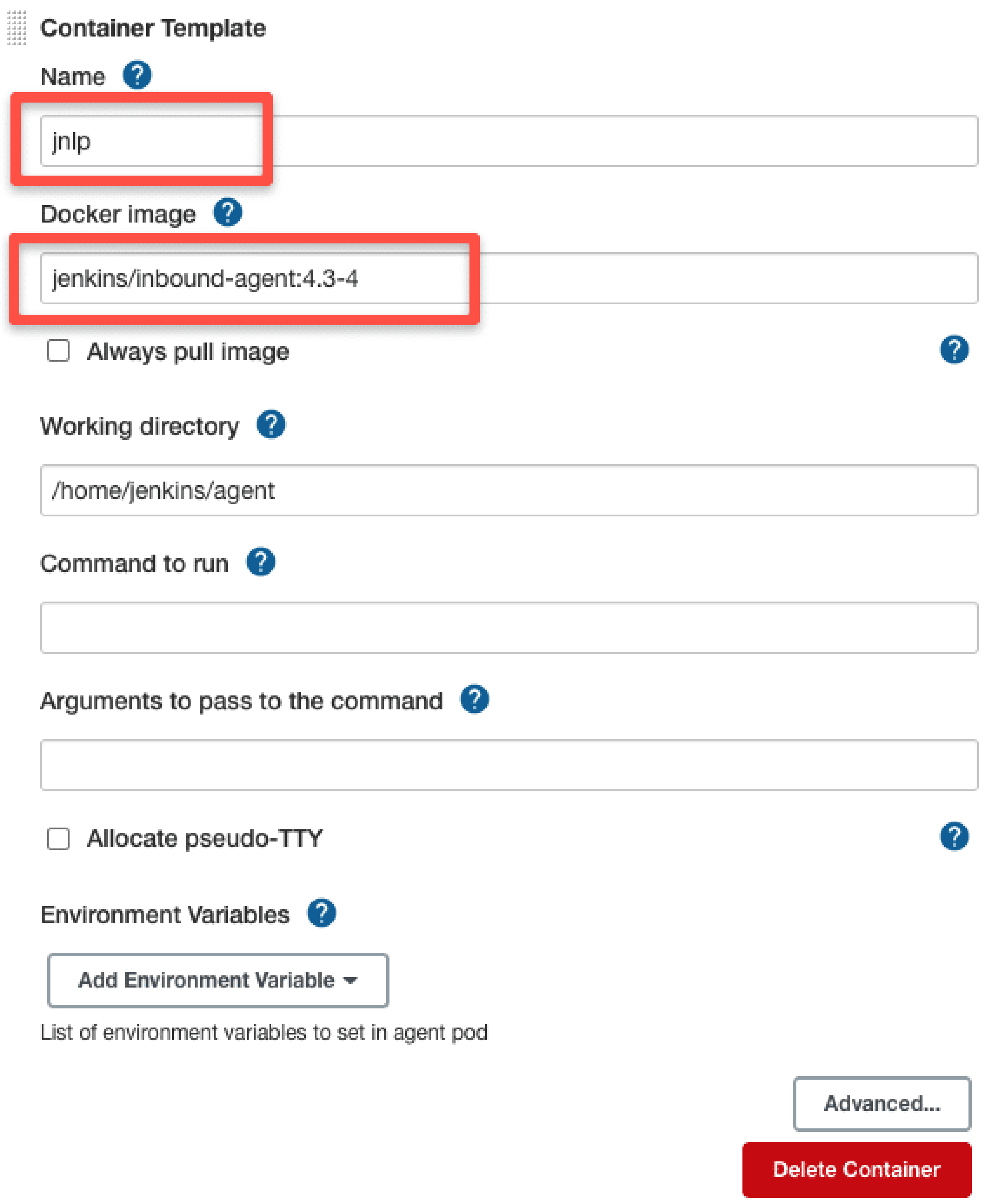
Pod template to inherit from:

Containers

The next configuration is the container template. If you don't add a container template, the Jenkins Kubernetes plugin will use the default JNLP image from the Docker hub to spin up the agents. ie, jenkins/inbound-agent

If you are on the corporate network and don't have access to the Docker hub, you will have to build your own `jnlp` image and override the default with the same name as shown below assuming `jenkins/inbound-agent:latest` is the custom jnlp image.

Ensure that you remove the `sleep` and `9999999` default argument from the container template.


Container Template

Name: jnlp

Docker image: jenkins/inbound-agent:4.3-4

☐ Always pull image

Working directory: /home/jenkins/agent

Command to run:

Arguments to pass to the command:

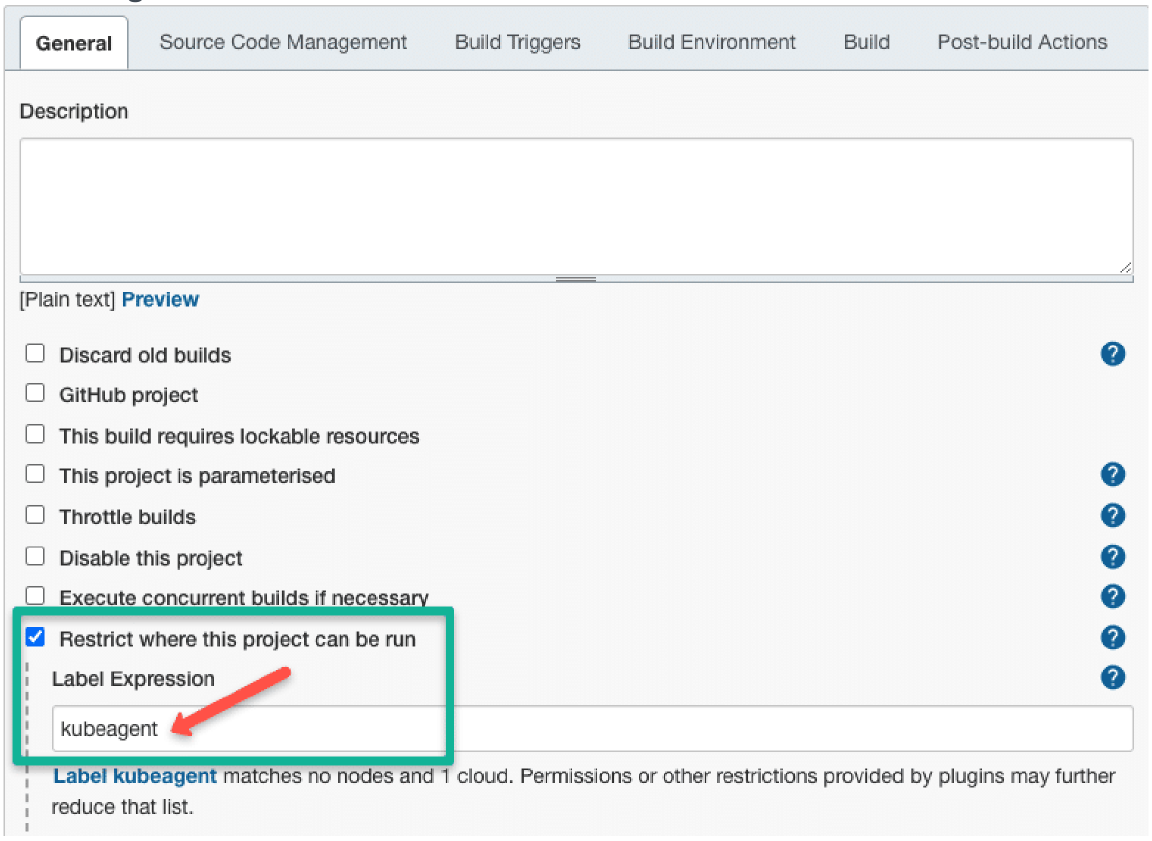
☐ Allocate pseudo-TTY

Environment Variables

List of environment variables to set in agent pod

We can add multiple container templates to the POD template and use them in the pipeline. I have explained that in the next section with `Jenkinsfile` examples. This is the base minimum configuration required for the agent to work. I will explain a few use cases of volumes and other options later in the pipeline examples.
Now save all the configurations and let’s test if we can build a job with a pod agent.
Step 6: Go to Jenkins home –> New Item and create a freestyle project.

In the job description, add the label `kubeagent` as shown below. It is the label we assigned to the pod template. This way, Jenkins knows which pod template to use for the agent container.



Description

[Plain text] Preview

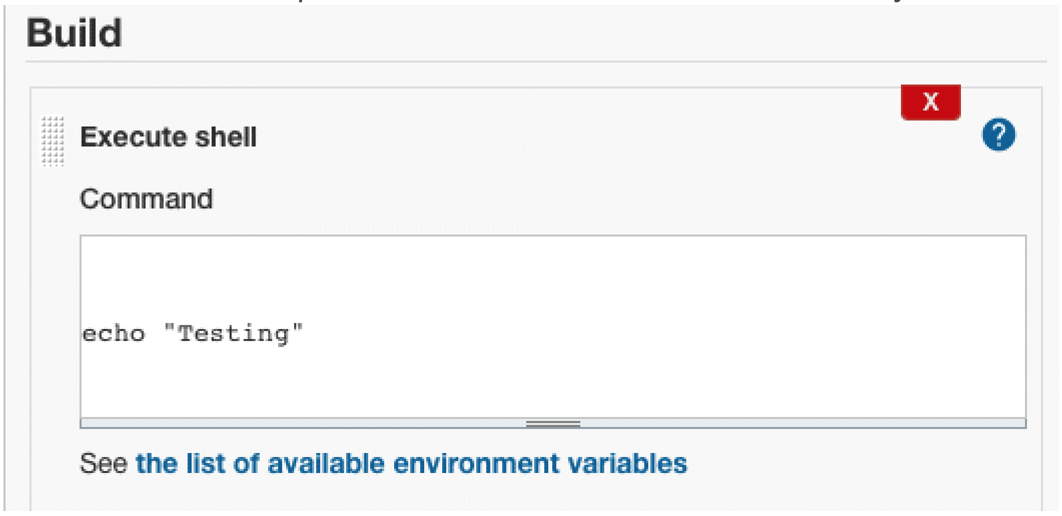
- [ ] Discard old builds
- [ ] GitHub project
- [ ] This build requires lockable resources
- [ ] This project is parameterised
- [ ] Throttle builds
- [ ] Disable this project
- [ ] Execute concurrent builds if necessary
- [x] Restrict where this project can be run

Label Expression

kubeagent

Label kubeagent matches no nodes and 1 cloud. Permissions or other restrictions provided by plugins may further reduce that list.

Add a shell build step with an echo command to validate the job as shown below.



Build

Execute shell

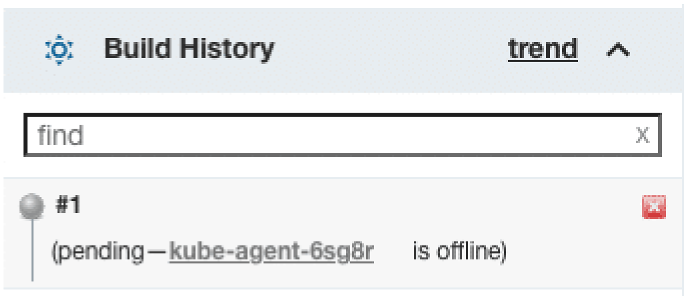
Command

```
echo "Testing"
```

See the list of available environment variables

Now, save the job configuration and click “Build Now”

You should see a pending agent in the job build history as shown below.

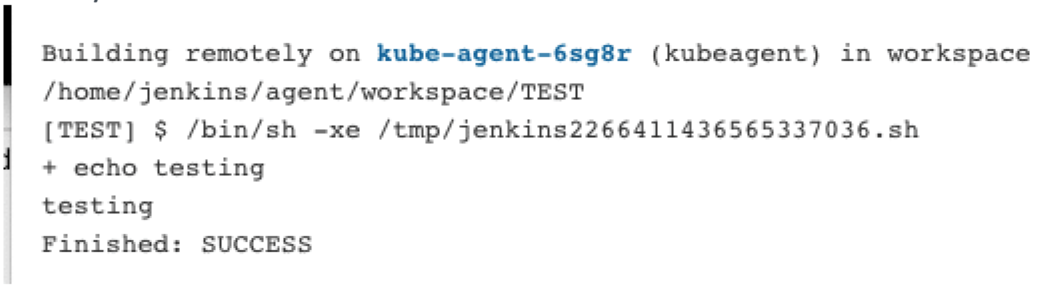

Build History trend

find

#1

(pending—kube-agent-6sg8r is offline)

In a couple of minutes, you will see a successful build. If you check the logs, it will show you the executed shell.



```
Building remotely on kube-agent-6sg8r (kubeagent) in workspace
/home/jenkins/agent/workspace/TEST
[TEST] $ /bin/sh -xe /tmp/jenkins2266411436565337036.sh
+ echo testing
testing
Finished: SUCCESS
```

Jenkinsfile With Pod Template

Whatever we have seen till now is to understand and validate the Kubernetes Jenkins plugin setup.

When it comes to actual project pipelines, it is better to have the POD templates in the `Jenkinsfile`

Here is what you should know about the POD template.

By default, a JNLP container image is used by the plugin to connect to the Jenkins server. You can override with a custom JNLP image provided you give the name `jnlp` in the container template.

You can have multiple container templates in a single pod template. Then, each container can be used in different pipeline stages.

`POD_LABEL` will assign a random build label to the pod when the build is triggered. You cannot give any other names other than `POD_LABEL`

Here is an example `Jenkinsfile` with a POD template.

```
podTemplate {
  node(POD_LABEL) {
    stage('Run shell') {
      sh 'echo hello world'
    }
  }
}
```

If you build the above Jenkinsfile in a pipeline job, it will use the default JNLP image and execute the commands in the “Run Shell” stage. When I say default, the JNLP image from the docker hub will be used by the plugin if you don’t specify any.

Now, you can use your own `jnlp` image using a `containerTemplate` with all necessary build tools and use them in the pipeline as given below.

Here, instead of `jenkins/inbound-agent:latest`, you will have your own image.

```
podTemplate(containers: [
  containerTemplate(
      name: 'jnlp',
      image: 'jenkins/inbound-agent:latest'
      )
 ]) {

   node(POD_LABEL) {
      stage('Get a Maven project') {
         container('jnlp') {
            stage('Shell Execution') {
               sh '''
               echo "Hello! I am executing shell"
               '''
            }
         }
      }

   }
}
```

Multi Container Pod Template

You can use multiple container templates in a single POD template.

Here is a use case of this setup.

Let’s say you want to set up a single build pipeline that builds both Java and python project. In this case, you can use two container templates and use them in the build stages.

In the following example, in two separate stages, we are calling two different containers specified in the pod template.
One container contains all the maven dependencies for Java build and another contains Python build dependencies.

```
podTemplate(containers: [
  containerTemplate(
    name: 'maven',
    image: 'maven:3.8.1-jdk-8',
    command: 'sleep',
    args: '30d'
    ),
  containerTemplate(
    name: 'python',
    image: 'python:latest',
    command: 'sleep',
    args: '30d')
 ]) {

  node(POD_LABEL) {
    stage('Get a Maven project') {
      git 'https://github.com/spring-projects/spring-petclinic.git'
      container('maven') {
        stage('Build a Maven project') {
          sh '''
```

```
            echo "maven build"
            '''
          }
        }
      }

      stage('Get a Python Project') {
        git url: 'https://github.com/hashicorp/terraform.git', branch: 'main'
        container('python') {
          stage('Build a Go project') {
            sh '''
            echo "Go Build"
            '''
          }
        }
      }

  }
}
```

You can try building the above Jenkinsfile using the pipeline job.
While building, the above pipeline, if you check the kubernetes pods you will see three containers in the build agent pod as shown below.

Note: You cannot use the Docker hub images directly due to security compliance issues in actual projects. Sou you have to build your own Docker images and host them in the organization-approved container registry.

Using Shared Persistent Volumes With Jenkins Docker Agent Pods

To speed up the build process, it is better to attach a shared persistent volume to the build container.
For example, if you take a Java application, it has many Maven package dependencies.
When you build the Java apps, it downloads dependencies added in the pom.xml from the remote maven repository the first time, and it creates a local .m2 cache directory where the dependent packages are cached.
The `.m2` cache is not possible in Docker agent-based builds as it gets destroyed after the build.
We can create a persistent volume for the maven cache and attach it to the agent pod via the container template to solve this issue.
To demonstrate this, first, let’s create a PVC

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: maven-repo-storage
  namespace: devops-tools
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 50Gi
```

Here is an example `Jenkinsfile` with POD template that uses the `maven-repo-storage` persistent volume.

```
podTemplate(containers: [
  containerTemplate(
      name: 'maven',
      image: 'maven:latest',
      command: 'sleep',
      args: '99d'
      )
  ],

  volumes: [
  persistentVolumeClaim(
      mountPath: '/root/.m2/repository',
      claimName: 'maven-repo-storage',
      readOnly: false
      )
  ])

{
  node(POD_LABEL) {
    stage('Build Petclinic Java App') {
      git url: 'https://github.com/spring-projects/spring-petclinic.git', branch: 'main'
      container('maven') {
        sh 'mvn -B -ntp clean package -DskipTests'
      }
    }
  }
}
```

Building Docker Images On Kubernetes Cluster

If you are using Docker for deploying applications, you can integrate your CI Docker build pipeline on Kubernetes agents.

There are a few ways to run docker on docker for build use cases. However, due to the fact that Kubernetes removed docker runtimes, it is better to use alternative solutions.

For now, the best way to build docker images on the Kubernetes cluster is using Kaniko

Refer building docker image using kaniko to learn more about kaniko build pipeline using Jenkins pipeline.

Conclusion

If you are using Jenkins & kubernetes, you should definitely try out the container-based agents.

Scale your Jenkins agents on Kubernetes helps you from a lot of administrative overhead that you get with static build VMs. Even though there are dynamic VM build options are available, each build could take a long time compared to dynamic container agents.
You don’t have to worry about running out of resources for Jenkins builds.
Do give it a try and let me know if you face any issues.