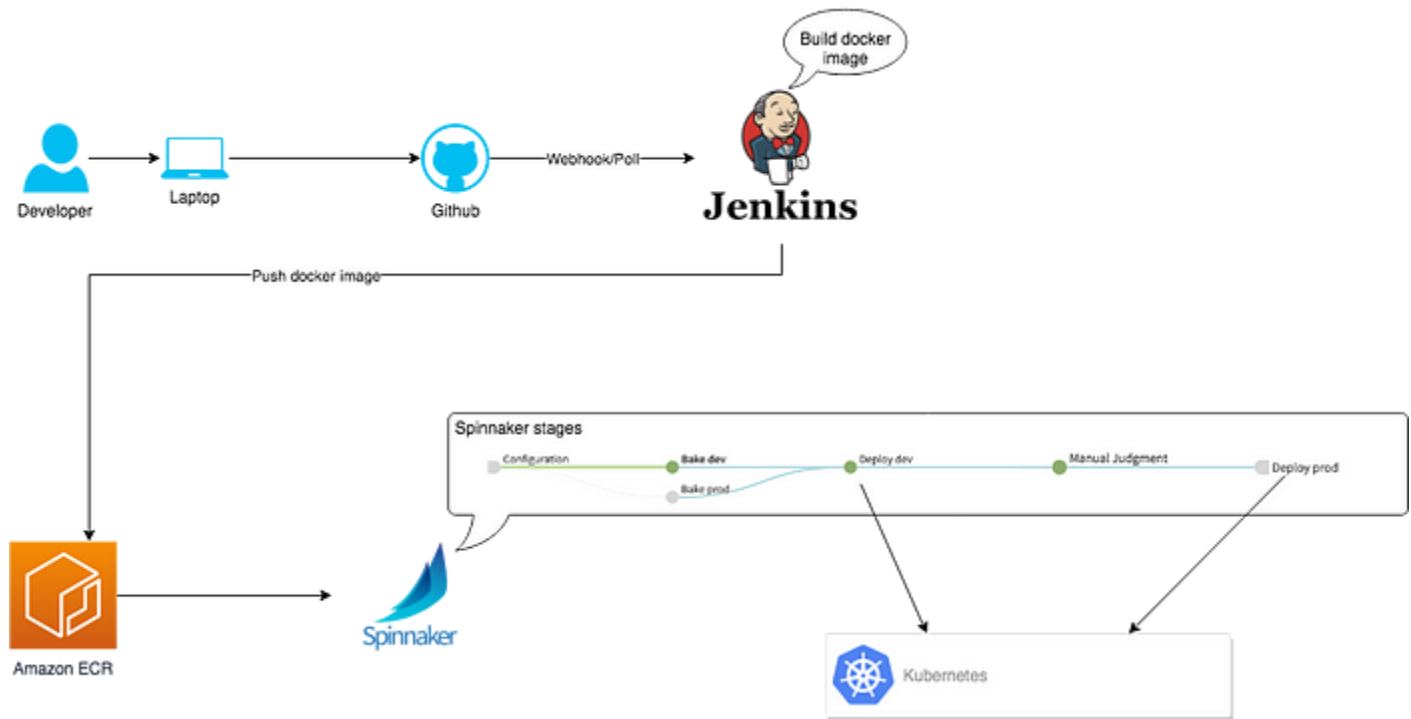


# Sample CI/CD with Spinnaker to deploy on EKS

[Spinnaker](#) is a continuous delivery platform, originally developed by Netflix, for releasing software changes rapidly and reliably. Spinnaker makes it easier for developers to focus on writing code without having to worry about the underlying cloud infrastructure. It integrates seamlessly with Jenkins and other popular build tools.

In this post we will discuss on how to install Spinnaker and build a continuous delivery pipeline for your workloads running on Kubernetes. Our workflow will look like:



These steps are covered in the diagram:

1. Developer pushes code to GitHub.
2. GitHub triggers Jenkins.
3. Jenkins builds a Docker image, tags and pushes it to Amazon Elastic Container Registry (Amazon ECR).
4. The Spinnaker pipeline is triggered when Amazon ECR receives the new Docker image.
5. Spinnaker then does following:
6. Generate (bake) Kubernetes deployment files (dev and prod) using Helm.

7. Deploy Kubernetes to the dev environment.
8. Manual judgement: Our pipeline configuration requires a manual confirmation by a human before it can deploy the app to production. It will wait at this step before pipeline execution can continue.
9. Deploy the code to the production environment.

## Prerequisites

1. A running Kubernetes cluster. If you don't already have one running, use [eksctl](#) to [get an EKS cluster up and running with one command](#).
2. At least eight GB of free memory and two vCPU in the Kubernetes cluster for Spinnaker microservices. An m5.large instance should do the job.
3. [kubectl installed](#), configured, and working on your machine.
4. [Helm](#) installed. To install it, follow the [Kubernetes Helm instructions](#).
5. [Jenkins](#) installed. To install it, follow the instructions in the [documentation on Jenkins on AWS](#).
6. [Docker](#) and the [Amazon ECR plugin](#) installed for Jenkins and configured to work.

7. A Docker registry account. If you don't have one, you can use [Amazon ECR](#), as we will be doing in this post. You could also use [Docker Hub](#).
8. An authentication provider (LDAP/SAML/Oauth2). In this post we will be using Active Directory (LDAP) authentication. If you don't already have one, follow the [AWS Managed Microsoft AD documentation](#).

## Steps

Once you have all the prerequisites in place, you can begin the actual steps to set up the pipeline. We will go through each of these steps in detail; here's an overview of what we'll be doing:

1. Build a sample application: [Hello world sample microservice](#).
2. Install Spinnaker on EKS using Helm.
3. Set up LDAP/AD authentication.
4. Expose Spinnaker by setting up an ingress controller.
5. Add a GitHub account to Spinnaker.
6. Configure Amazon ECR in your AWS account to store Docker images pushed by Jenkins.
7. Configure Jenkins for Docker image build and ECR push.

8. Build the CI/CD pipeline in Spinnaker — automated build using web-hook from GitHub, manual approval for deployment to production.
9. Run the pipeline and deploy the application.
10. Test.
11. Teardown.

## Step 1: Build a sample application

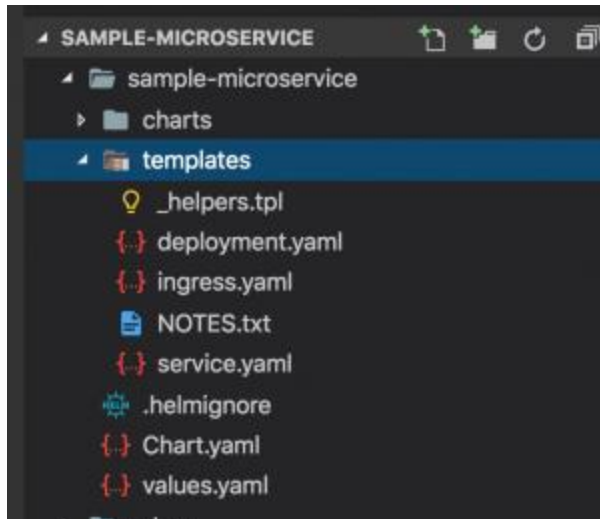
To demonstrate for this post, we will use the [sample application](#) that our pipeline will build and deploy. Please fork the sample application to continue with the next step.

This repo includes a Helm chart that will be used for deployment by Spinnaker. Items mentioned in the rest of the section have already been completed for this repo which you can use right away. If you are using the sample application, skip to Step 2!!! Otherwise, if you are using your own:

### If you are using your own application

If you prefer to use your own application, you will need to create your own Helm chart and package it. Follow the steps below to create and package a Helm chart for your application.

```
helm create sample-microservice
```



Open sample-microservice/templates/deployment.yaml and make the following changes:

## 1.1 Add namespace

Add `namespace: {{ .Release.Namespace }}` to Helm template deployment. This helps Spinnaker to deploy the Kubernetes deployments in the specific namespaces mentioned in the deployment stage

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: {{ include "sample-microservice.fullname" . }}
  namespace: {{ .Release.Namespace }}
```

## 1.2 Change image

Change : `"{{ .Values.image.repository }}:{{ .Values.image.tag }}"` in deployment.yaml in the Helm template to `{{ .Values.image.repository }}` (this allows Spinnaker to replace the tags for deployment) from:

```
spec:
  containers:
```

```
- name: {{ .Chart.Name }}
  image: {{ .Values.image.repository }}
  imagePullPolicy: "{{ .Values.image.pullPolicy }}:{{
.Values.image.tag }}"
```

to:

```
spec:
  containers:
    - name: {{ .Chart.Name }}
      image: {{ .Values.image.repository }}
      imagePullPolicy: {{ .Values.image.pullPolicy }}
```

Package the Helm chart by running the command:

```
helm package sample-microservice
```

This command will create sample-microservice-0.1.0.tgz which will be used by Spinnaker.

## Step 2: Install Spinnaker using Helm

To install Spinnaker with the default configuration, run this command:

```
helm install stable/spinnaker --name=spinnaker --
namespace=spinnaker
```

You can learn more about [Helm charts for Spinnaker on GitHub](#).

While installation is in progress, let's talk about some Spinnaker components that we will be using in this post. For a detailed architecture and components of Spinnaker you can refer to [Spinnaker's reference architecture](#).

1. [Deck](#) — This is the front-end component of Spinnaker that provides the browser-based UI.
2. [Gate](#) — This service provides the Spinnaker REST API, servicing scripting clients as well as all actions from Deck.
3. [Halyard](#) — is the CLI for configuring, installing, and updating Spinnaker.

Spinnaker uses [Redis](#) as caching infrastructure to store information related to live execution; you can learn more about this on the [Spinnaker Redis configuration page](#). We will be using a Redis installation that the Helm chart depends on and installs within the Kubernetes cluster. For a production setup of Spinnaker you would want to [externalize Redis](#). You'll also want to read [productionize Spinnaker documentation](#).

Spinnaker also needs a data store ([S3](#), [Minio](#), or other object store). Default installation uses Minio. For production you would want to use S3 by enabling S3 in [values.yaml](#) instead of Minio.

To verify your Spinnaker installation:

```
kubectl -n spinnaker get pods:
```

Output like this confirms the successful installation of Spinnaker:

NAME	READY	STATUS	
RESTARTS	AGE		
spin-clouddriver-945c95564-8w152	1/1	Running	0
2h			
spin-deck-6c4bf6c4f6-wqgmK	1/1	Running	0



2h				
spin-echo-646f6c4b76-p29tl	1/1	Running	0	
2h				
spin-front50-7cc5575457-qcvtd	1/1	Running	1	
2h				
spin-gate-84dc696d7c-zqctg	1/1	Running	0	
2h				
spin-igor-885f8bf5c-xprkc	1/1	Running	0	
2h				
spin-orca-7bfd8fd4d6-28dks	1/1	Running	0	
2h				
spin-rosco-844b85888-sggkk	1/1	Running	0	
2h				
spinnaker-install-using-hal-qlvfj	0/1	Completed	0	
2h				
spinnaker-Minio-df54fb68d-h4ld9	1/1	Running	0	
2h				
spinnaker-Redis-master-0	1/1	Running	0	
2h				
spinnaker-spinnaker-halyard-0	1/1	Running	0	
2h				

**To list the services, run:**

```
kubectl -n spinnaker get services
```

**Output:**

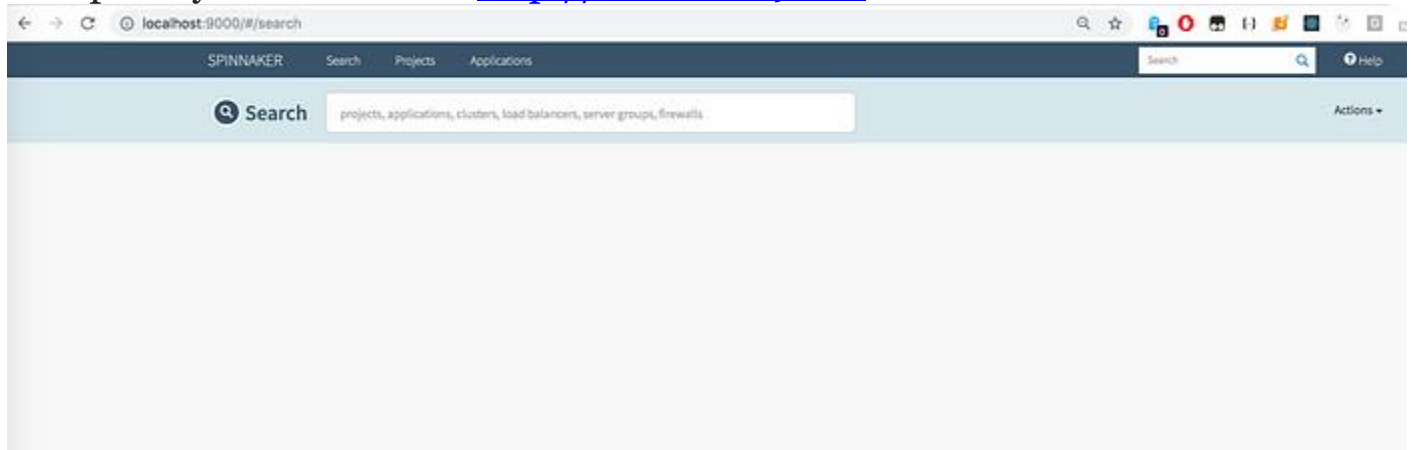
NAME	EXTERNAL-IP	PORT(S)	AGE	TYPE	CLUSTER-IP	
spin-clouddriver				ClusterIP	172.20.135.53	<none>
7002/TCP		2h				
spin-deck				ClusterIP	172.20.167.104	<none>
9000/TCP		2h				
spin-echo				ClusterIP	172.20.46.99	<none>
8089/TCP		2h				
spin-front50				ClusterIP	172.20.234.34	<none>
8080/TCP		2h				
spin-gate				ClusterIP	172.20.132.82	<none>
8084/TCP		2h				
spin-igor				ClusterIP	172.20.87.99	<none>
8088/TCP		2h				
spin-orca				ClusterIP	172.20.241.201	<none>
8083/TCP		2h				
spin-rosco				ClusterIP	172.20.136.62	<none>
8087/TCP		2h				

spinnaker-Minio	ClusterIP	None	<none>
9000/TCP 2h			
spinnaker-Redis-master	ClusterIP	172.20.80.211	<none>
6379/TCP 2h			
spinnaker-spinnaker-halyard	ClusterIP	None	<none>
8064/TCP 2h			

To launch the Spinnaker UI, run:

```
kubectl -n spinnaker port-forward svc/spin-deck 9000:9000
```

And point your browser to <http://localhost:9000>. You should see:



## Step 2.1: Set up LDAP/AD authentication

Get the URL for your Active Directory server. I have an AD server running in my AWS account in the same VPC as my Kubernetes cluster. If you don't already have one, head over to [AWS Managed AD](#) and get one for yourself.

Create a file named `gate-local.yaml` as shown below. This will hold the configuration of Active Directory for Spinnaker.

```
ldap:
  enabled: true
  url: ldap://10.0.157.236:389/dc=ad,dc=prabhatsharma,dc=com
  userSearchBase: OU=users,OU=ad
```

```
userSearchFilter: (sAMAccountName={0})
managerDn:
CN=prabhat,OU=users,OU=ad,dc=ad,dc=prabhatsharma,dc=com
managerPassword: MySuper#StrongPassword
```

Copy `gate-local.yaml` to Halyard:

```
kubectl cp gate-local.yaml spinnaker-spinnaker-halyard-0:/home/spinnaker/.hal/default/profiles/
```

Apply the Halyard configuration:

```
kubectl exec spinnaker-spinnaker-halyard-0 -- bash hal deploy apply
```

## Step 3: Expose Spinnaker — set up ingress controller

This is an optional step which is needed only if you want to expose Spinnaker external to your Kubernetes cluster. You must have the [NGINX ingress controller installed](#) in order for this step to work.

I have a public wildcard domain configured in Route53 which is pointed to my NGINX ingress ELB. You will need to use your own domain for this by replacing [yourcustomdomain.com](#) with your own domain.

Create a file `spinnaker-ingress.yaml`:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: spinnaker
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  tls:
  - hosts:
    - spinnaker.yourcustomdomain.com
    - spin-gate.yourcustomdomain.com
```

```
rules:
- host: spinnaker.yourcustomdomain.com
  http:
    paths:
      - path: /
        backend:
          serviceName: spin-deck
          servicePort: 9000
- host: spin-gate.yourcustomdomain.com
  http:
    paths:
      - path: /
        backend:
          serviceName: spin-gate
          servicePort: 8084
```

### Deploy the ingress:

```
kubectl -n spinnaker apply -f spinnaker-ingress.yaml
```

At this point you have both deck (Spinnaker UI endpoint) and gate (Spinnaker API endpoint) exposed.

You now need to tell Spinnaker to use the new Spinnaker endpoints that we just deployed. We will be using Halyard to do that. To run Halyard, log in to the Halyard pod:

```
kubectl -n spinnaker exec -it spinnaker-spinnaker-halyard-0 bash
```

This will drop you into the shell in the container.

```
spinnaker@spinnaker-spinnaker-halyard-0:/workdir$
```

Run the commands below to configure Spinnaker to use the new endpoints. You can always refer to the complete list of [Halyard commands](#) for any other configuration needed.

```
hal config security api edit --override-base-url https://spin-gate.yourcustomdomain.com
```

```
hal config security ui edit --override-base-url  
https://spinnaker.yourcustomdomain.com  
hal deploy apply
```

After this you will be able to access Spinnaker  
at <https://spinnaker.yourcustomdomain.com>

## Step 4: Add a GitHub account to Spinnaker

We will use Halyard to add a GitHub account

To access the Halyard pod:

```
kubectl -n spinnaker exec -it spinnaker-spinnaker-halyard-0 bash  
hal config artifact github account add aws-samples  
hal deploy apply
```

Note: Please change aws-samples to your own Github account name.

The above commands will allow Spinnaker to access GitHub.

## Step 5: Configure the Amazon ECR repository for Jenkins image push

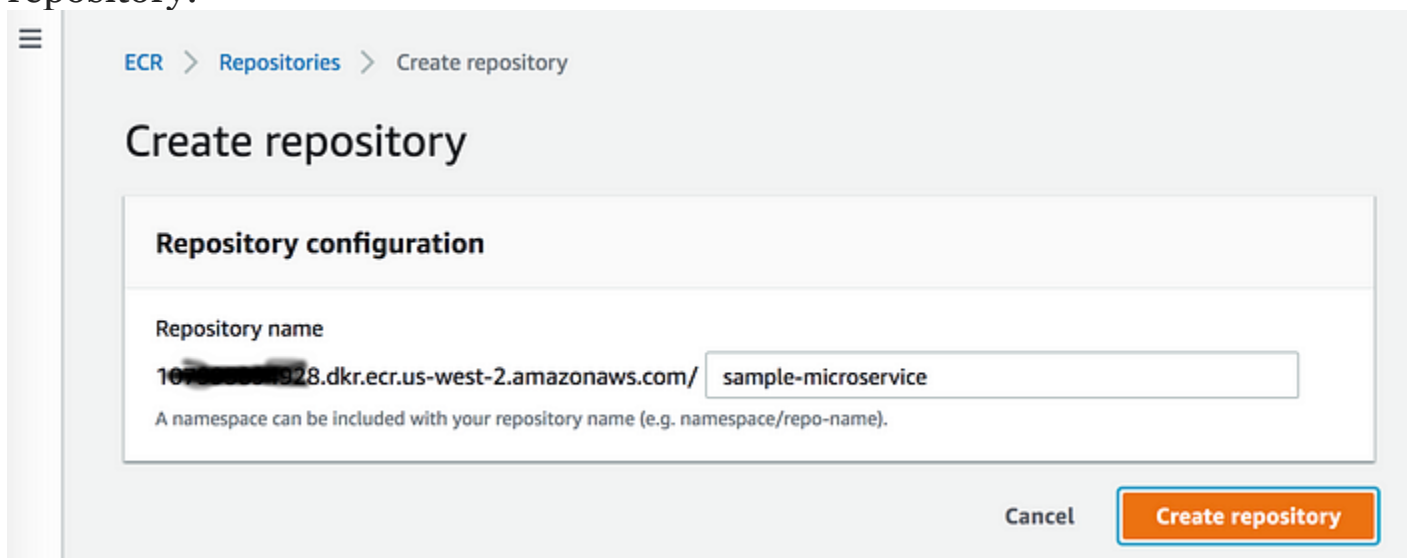
You need a Docker repository to store your microservice Docker images. For that we will create an Amazon ECR repository.

Navigate to AWS console > Compute > ECR.

Click Create repository.



Type in the repository name (sample-microservice) and click Create repository.



This repository will hold the Docker images of our microservice that are pushed by Jenkins.

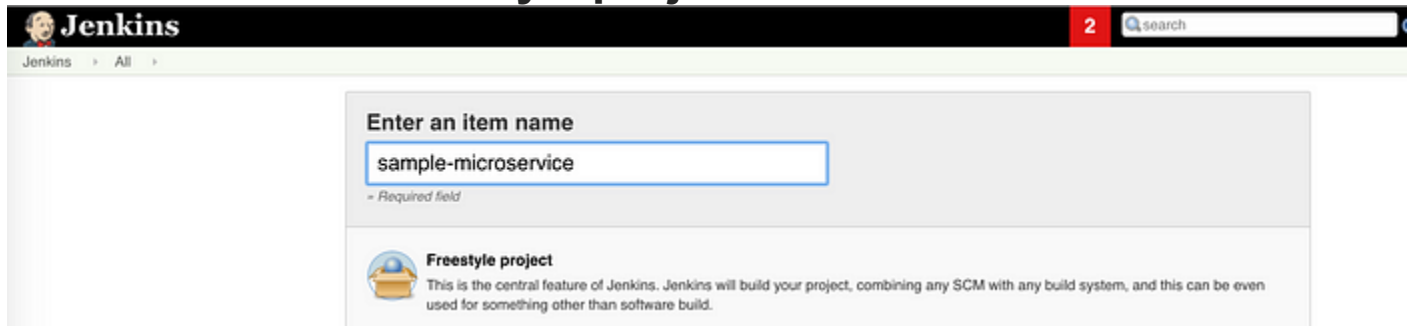
## Step 6: Configure Jenkins for Docker image build and ECR push

Note: You must have the [Amazon ECR plugin](#) installed and configured for this to work. You can verify that the plugin is installed by going to Jenkins > Plugin Manager > Installed and searching for Amazon ECR.

We will configure a Jenkins job that will be triggered by a push to code in GitHub. This job will build a Docker image and push the image to Amazon ECR.

Now log in to your Jenkins installation and:

## 6.1 Create a new freestyle project



The screenshot shows the Jenkins web interface. At the top, there's a header with the Jenkins logo and a search bar. Below the header, the main content area is titled 'Enter an item name'. A text input field contains the text 'sample-microservice'. Below the input field, there's a small text indicating it's a required field. Underneath, there's a section for 'Freestyle project' with a description: 'This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.'

## 6.2 Configure source code management

Change the path of your forked GitHub repo and replace aws-samples with your own username, e.g., replace <https://github.com/aws-samples/sample-microservice-with-spinnaker> with [https://github.com/\[your GitHub handle\]/sample-microservice-with-spinnaker](https://github.com/[your GitHub handle]/sample-microservice-with-spinnaker).

The screenshot shows the Jenkins configuration page for Source Code Management. The tabs at the top are General, Source Code Management (selected), Build Triggers, Build Environment, Build, and Post-build Actions. There is an Advanced... button on the right. Under Source Code Management, the 'Git' radio button is selected. In the 'Repositories' section, the 'Repository URL' is 'https://github.com/aws-samples/sample-microservice-with-spinnaker.git' and 'Credentials' is '- none -'. There are 'Add', 'Advanced...', and 'Add Repository' buttons. In the 'Branches to build' section, the 'Branch Specifier (blank for 'any')' is '\*/master', with an 'Add Branch' button. The 'Repository browser' is set to '(Auto)'.

## 6.3 Configure build trigger

You could use a [webhook](#) or polling. We are using a webhook for this blog. Configuration of the Jenkins webhook for GitHub is out of scope of this post.

The screenshot shows the Jenkins configuration page for Build Triggers. The 'Build Triggers' section has five options: 'Trigger builds remotely (e.g., from scripts)', 'Build after other projects are built', 'Build periodically', 'GitHub hook trigger for GITScm polling' (which is checked), and 'Poll SCM'. Each option has a help icon. Below this is the 'Build Environment' section.

## 6.4 Configure build phase

We will be using the Jenkins build number as the Docker image tag:



General Source Code Management Build Triggers Build Environment **Build** Post-build Actions

**Build / Publish Docker Image**

Directory for Dockerfile: .  
Location to look for the Dockerfile in, which is used to build the image.

Cloud: docker  
Cloud to use to build image

Image:  
123456789123.dkr.ecr.us-west-2.amazonaws.com/sample-microservice  
123456789123.dkr.ecr.us-west-2.amazonaws.com/sample-microservice:v\${BUILD\_NUMBER}

Push image: ☒

Registry Credentials: AWS/\*\*\*\*\* (Amazon ECR Registry:96a42c02-5ac0-4d58-b06b-3be337e92418-US\_W) Add

Clean local images: ☒

Attempt to remove images when Jenkins deletes the run: ☒

The Jenkins variable `BUILD_NUMBER` will be used as a tag for the newly-created image.

## Step 7: Configure Amazon ECR for Spinnaker

Note: For this to work, your Kubernetes nodes must have an appropriate IAM role assigned to allow access to ECR. You can find [sample IAM policies](#) in the documentation that can be assigned to your Kubernetes worker node IAM role.

This configuration will allow you to configure the Spinnaker pipeline to be triggered when a container is pushed to ECR.

```
ADDRESS=123456789123.dkr.ecr.us-west-2.amazonaws.com
REGION=us-west-2
hal config provider docker-registry account add my-ecr-registry \
--address $ADDRESS \
--username AWS \
```

```
--password-command "aws --region $REGION ecr get-authorization-  
token --output text --query  
'authorizationData[].authorizationToken' | base64 -d | sed  
's/^AWS:/'"hal deploy apply
```

More information about managing Docker registries can be found at [Spinnaker's Docker registry documentation](#).

## Step 8: Build the CI/CD pipeline in Spinnaker

Before you start building the pipeline, you need to understand certain Spinnaker concepts:

[Application](#) — An application represents the service you are going to deploy using Spinnaker, all configuration for that service, and all the infrastructure on which it will run. You will typically create a different application for each service, though Spinnaker does not enforce that.

[Pipeline](#) — A pipeline is a sequence of stages provided by Spinnaker, ranging from functions that manipulate infrastructure (deploy, resize, disable) to utility scaffolding functions (manual judgment, wait, run Jenkins job). All of these together precisely define the runbook for managing your deployments.

[Stage](#) — A Stage in Spinnaker is an atomic building block for a pipeline, describing an action that the pipeline will perform. You can sequence stages in a Pipeline in any order, though some stage sequences are more common than others. Spinnaker provides a number of stages such as Deploy, Resize, Disable, Manual Judgment, and many more.

[Artifact](#) — In Spinnaker, an artifact is an object that references an external resource. That resource could be:

- a Docker image
- a file stored in GitHub
- an Amazon Machine Image (AMI)
- a binary blob in S3, GCS, etc.

Spinnaker uses Helm v2 for managing deployments to Kubernetes. You must specify the base Helm template and an override document for each environment to which you want to push the deployment.

Now let's proceed with the required steps for setting up the pipeline:

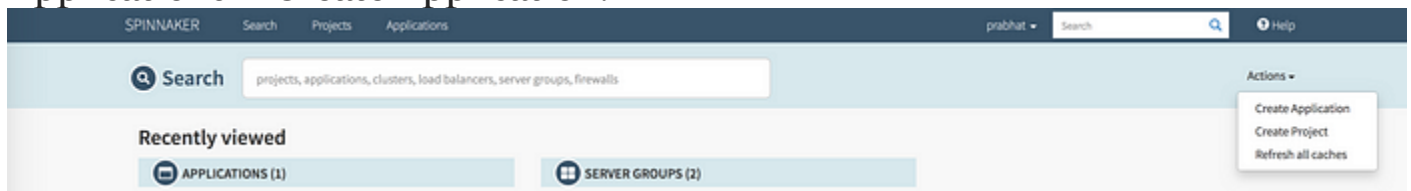
1. Create application
2. Create pipeline
3. Set up configuration
4. Set up artifacts
5. Helm template — sample-microservice-0.1.0.tgz
6. Helm dev override — values/dev.yaml
7. Helm prod override — values/prod.yaml

8. Docker image — 123456789123.dkr.ecr.us-west-2.amazonaws.com/sample-microservice
9. Set up pipeline trigger
10. Create stages
11. Bake dev
12. Bake prod
13. Deploy to dev
14. Manual judgement
15. Deploy to prod

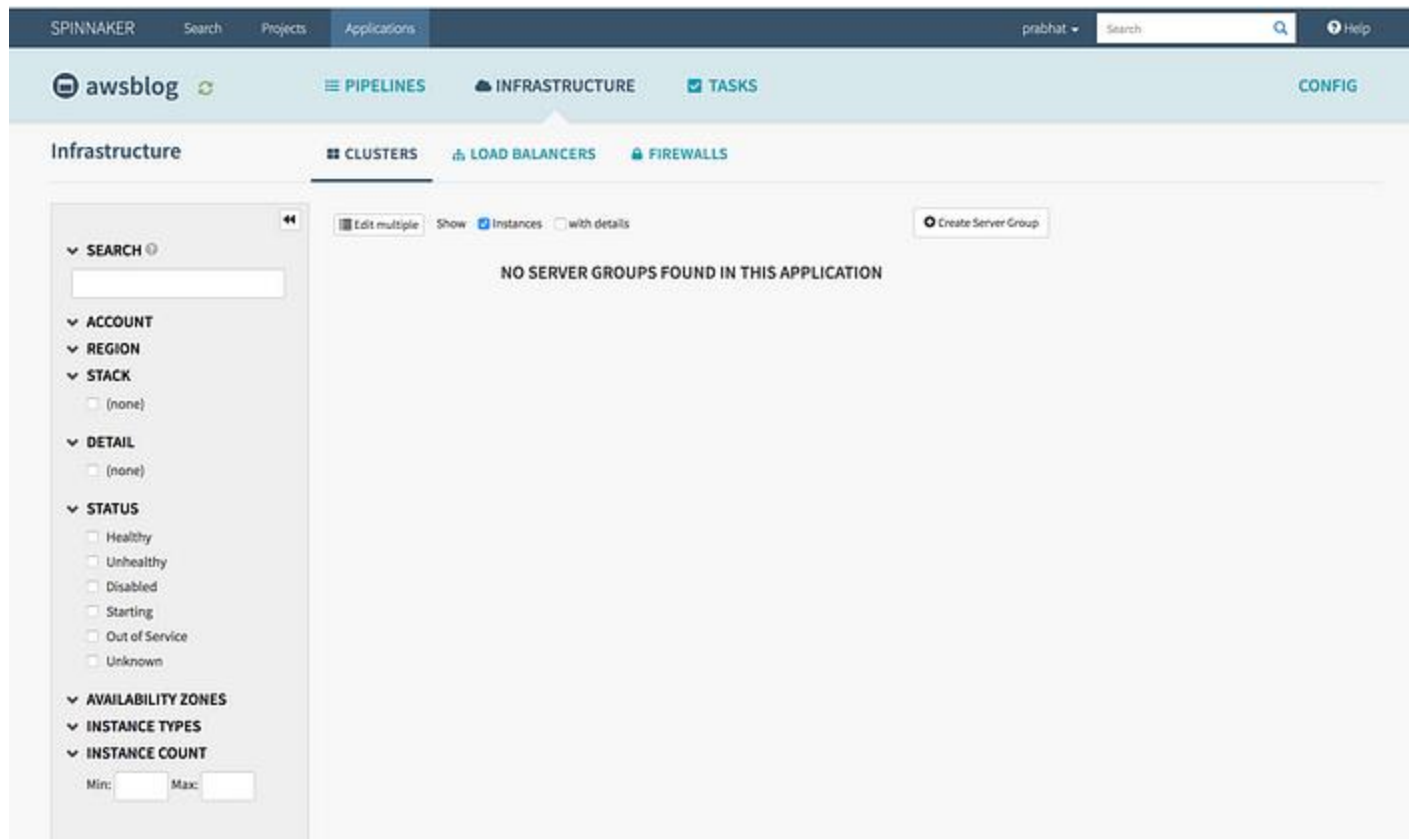
## 8.1 Create application

Our application will be the placeholder in Spinnaker for the service for which we are building the pipeline.

Once you are logged in to Spinnaker, create a new application from Applications > Create Application.







## 8.2 Create a pipeline

Head over to Pipelines and click Configure a new pipeline.



Create New Pipeline

Type

Pipeline

Pipeline Name

sample-microservice

Cancel

Create

## 8.3 Set up pipeline configuration

You should now be at:

sample-microservice

Permalink

Create

Configure

Pipeline Actions

Configuration

Add stage

Copy an existing stage

CONCURRENT EXECUTIONS

EXPECTED ARTIFACTS

AUTOMATED TRIGGERS

PARAMETERS

NOTIFICATIONS

DESCRIPTION

Concurrent Executions

☒ Disable concurrent pipeline executions (only run one at a time).

☐ Do not automatically cancel pipelines waiting in queue.

Expected Artifacts

Declare artifacts your pipeline expects during execution in this section.

You don't have any expected artifacts declared for sample-microservice.

Add Artifact

Automated Triggers

You don't have any triggers configured for sample-microservice.

Add Trigger

We will now configure Expected Artifacts:

## Helm Template artifact

Click Add Artifact.


Note: Replace aws-samples with your GitHub handle in the steps below.

Set default content URL to <https://api.github.com/repos/aws-samples/sample-microservice-with-spinnaker/contents/sample-microservice-0.1.0.tgz>

**Expected Artifacts**


Declare artifacts your pipeline expects during execution in this section. ?

Match against ?

 GitHub

▼

A file stored in git, hosted by GitHub.

 Remove artifact

File path ?


sample-microservice-0.1.0.tgz

If missing ?

Use Prior Execution ☐

Use Default Artifact ☒

Default artifact ?

 GitHub

▼

A file stored in git, hosted by GitHub.

Content URL ?

https://api.github.com/repos/prabhatsharma/sample-microservice-with-spinnaker/contents/s

Commit/Branch ?

master



Artifacts are passed on to Spinnaker through a trigger by a webhook whenever it kicks. If your pipeline requires a particular artifact for executing the pipeline that it has not received with the trigger, you can specify a default artifact for use. In this case we would specify default artifact since our pipeline is not being triggered by GitHub and therefore won't be passed the artifact which we need for execution.

Similarly configure other artifacts:

### **Dev override artifact**

Artifact type — GitHub

File path — values/dev.yaml

Default artifact content URL — <https://api.github.com/repos/aws-samples/sample-microservice-with-spinnaker/contents/values/dev.yaml>

### **Prod override artifact**

Artifact type — GitHub

File path— values/prod.yaml

Default artifact content URL — <https://api.github.com/repos/aws-samples/sample-microservice-with-spinnaker/contents/values/prod.yaml>

### **Docker image artifact**

Artifact type — Docker

Docker image — 123456789123.dkr.ecr.us-west-

2.amazonaws.com/sample-microservice

Default artifact Docker image — 123456789123.dkr.ecr.us-west-

2.amazonaws.com/sample-microservice:latest

Now we will configure Automated Triggers:


Automated triggers can start a pipeline whenever a specific event happens (e.g., Docker image push to a registry, code push to GitHub, etc.). We want the pipeline to start when a new Docker image becomes available in our ECR repository.

Configure it by selecting the registry name and image from the drop down under Automated Triggers:

## Automated Triggers

Type


Docker Registry ▾ Executes the pipeline on an image update

 Remove trigger

Define Image ID

Select from list ▾

Registry Name

my-ecr-registry × ▾ 

Organization

No organization ▾

Image


sample-microservice × ▾

Tag ⓘ

Artifact Constraints ⓘ

Select...

☒ Trigger Enabled

 Add Trigger

Now save the changes by clicking Save Changes at bottom right.

## 8.4 Add bake stages

Now that our pipeline configuration is done, it's time to add a new stage

Bake dev — This stage will use Helm v2 to render the deployment template using the override values in dev.yaml.

Go to the top of the pipeline and click Add stage.

sample-microservice
Permalink
Create
Configure
Pipeline Actions

Configuration

Add stage

Copy an existing stage

Provide a name and a Kubernetes namespace where the deployment will take place. The namespace must already exist, or the pipeline will fail during execution.

Bake dev

Stage type: Bake (Manifest)  
 Bake a manifest (or multi-doc manifest set) using a template renderer such as Helm.

Type

Bake (Manifest)

Stage Name

Bake dev

Depends On

Select...

Remove stage

Edit stage as JSON

BAKE (MANIFEST) CONFIGURATION

EXECUTION OPTIONS

NOTIFICATIONS

COMMENTS

PRODUCES ARTIFACTS

Bake (Manifest) Configuration

Template Renderer

Render Engine

HELM2

Name

sample-microservice-dev

Namespace

sample-microservice-dev

Template Artifact

Expected Artifact

name: sample-microservice-0.1.0.tgz, type: github/file

Artifact Account

aws-samples

Overrides

Expected Artifact

name: values/dev.yaml, type: github/file

Artifact Account

aws-samples

Add value artifact

Overrides

Key

Value

Add override

This will also create a section called Produces Artifacts that you can scroll down to see:

## Produces Artifacts

Match against ⓘ

 Remove artifact

b64 Base64 ▼

An artifact that includes its referenced resource as part of its payload.

**Name**

sample-microservice-dev

If missing ⓘ

**Use Default Artifact**

☐

 Add Artifact

This produced artifact is a base64-encoded Kubernetes deployment (with service, ingress, etc.) file.

Similar to Bake dev stage above, create a stage Bake prod.

**Bake prod**  
**Stage type:** Bake (Manifest)  
Bake a manifest (or multi-doc manifest set) using a template renderer such as Helm.

Type:

Stage Name:

Depends On:

Remove stage

Edit stage as JSON

#### BAKE (MANIFEST) CONFIGURATION

EXECUTION OPTIONS

NOTIFICATIONS

COMMENTS

PRODUCES ARTIFACTS

#### Bake (Manifest) Configuration

##### Template Renderer

Render Engine:

Name:

Namespace:

##### Template Artifact

Expected Artifact:

Artifact Account:

##### Overrides

Expected Artifact:

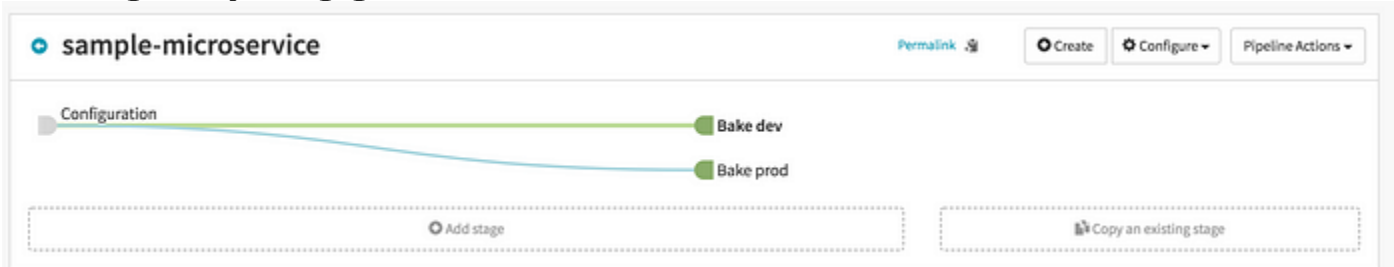
Artifact Account:

Add value artifact

Overrides

Key	Value
Add override	

At this point your pipeline would look like:



## 8.5 Add deploy dev stage

After bake dev and bake prod stages are completed, we have Kubernetes deployment files ready which we can use for deployment. Now create a stage Deploy dev which will deploy to dev environment.

With Bake dev stage selected, click Add stage. Add Bake prod as a dependency along with Bake dev.

**sample-microservice** [Permalink](#) [Create](#) [Configure](#) [Pipeline Actions](#)

Configuration → Bake dev → Bake prod → Deploy dev

[Add stage](#) [Copy an existing stage](#)

**Deploy dev**  
Stage type: Deploy (Manifest)  
Deploy a Kubernetes manifest yaml/json file.

Type:

Stage Name:

Depends On:

[Remove stage](#) [Edit stage as JSON](#)

**DEPLOY (MANIFEST) CONFIGURATION**

EXECUTION OPTIONS  
NOTIFICATIONS  
COMMENTS  
PRODUCES ARTIFACTS

**Deploy (Manifest) Configuration**

**Basic Settings**

Account:

Application:

**Manifest Configuration**

Manifest Source: ☐ Text ☒ Artifact

Expected Artifact:

## 8.6 Add manual judgement stage

Many teams want someone to approve manually before a deployment is pushed to production. If yours is such a team, you can add a Manual Judgement stage.

Click Add stage and select Manual Judgement from the drop down:

sample-microservice
Permalink
Create
Configure
Pipeline Actions

Configuration
Bake dev
Bake prod
Deploy dev
Manual Judgment
Add stage
Copy an existing stage

Manual Judgment
Stage type: Manual Judgment
Waits for user approval before continuing

Type: Manual Judgment
Stage Name: Manual Judgment
Depends On: Deploy dev
Remove stage
Edit stage as JSON

MANUAL JUDGMENT CONFIGURATION
EXECUTION OPTIONS
COMMENTS

Manual Judgment Configuration

Instructions
Provide any instructional text that would assist making a manual judgment (can contain HTML)

Propagate Authentication
Judgment Inputs
Send Notifications

Option
Add judgment input

## 8.7 Add deploy to prod stage

This is our final stage where, if everything goes well, we push the deployment to the production environment. Create it by clicking Add stage and selecting the expected artifact sample-microservice-prod, type: embedded/base64. It should have a the dependency Manual Judgement.



The screenshot shows the Jenkins Pipeline Configuration interface for a pipeline named 'sample-microservice'. At the top, there's a header with the pipeline name, a 'Permalink' icon, and buttons for 'Create', 'Configure', and 'Pipeline Actions'. Below this is a visual representation of the pipeline stages: 'Configuration' (grey), 'Bake dev' (grey), 'Bake prod' (grey), 'Deploy dev' (grey), 'Manual Judgment' (green), and 'Deploy prod' (green). The 'Deploy prod' stage is currently selected. Below the pipeline diagram, there are buttons for 'Add stage' and 'Copy an existing stage'. The 'Deploy prod' stage configuration is shown below, including its 'Type' (Deploy (Manifest)), 'Stage Name' (Deploy prod), and 'Depends On' (Manual Judgment). To the right of the configuration are buttons for 'Remove stage' and 'Edit stage as JSON'. On the left side, there's a sidebar with links for 'DEPLOY (MANIFEST) CONFIGURATION', 'EXECUTION OPTIONS', 'NOTIFICATIONS', 'COMMENTS', and 'PRODUCES ARTIFACTS'. The main configuration area is titled 'Deploy (Manifest) Configuration' and contains 'Basic Settings' (Account: default, Application: awsblog) and 'Manifest Configuration' (Manifest Source: Artifact, Expected Artifact: name: sample-microservice-prod, type: embedded/base64).

## 9. Testing

Create two namespaces in your Kubernetes cluster:

```
kubectl create namespace sample-microservice-dev
kubectl create namespace sample-microservice-prod
```

Now you can test the entire pipeline by making a modification to `main.go` and pushing the commit to GitHub. You will notice the following, in sequence:

1. Jenkins build getting triggered.
2. New Docker image being published to Amazon ECR.

### 3. Spinnaker pipeline being triggered.

You can watch progress on the pipelines screen. At the manual judgement stage it will look like:

The screenshot shows the Spinnaker Pipelines interface. At the top, there are tabs for PIPELINES, INFRASTRUCTURE, TASKS, and CONFIG. The PIPELINES tab is active, showing a list of pipelines. The 'sample-microservice' pipeline is selected, and its execution is shown. The pipeline is in the 'RUNNING' state. A 'Manual Judgment' dialog box is open, showing 'Stop' and 'Continue' buttons. The pipeline progress bar shows stages: Bake dev, Deploy dev, Manual Judgment, and Deploy prod. The 'Bake dev' stage is currently active, and its details are shown in a modal window.

**Manual Judgment**

Stop Continue

**DOCKER REGISTRY**  
sample-microservice:v9  
2 minutes ago  
sample-microservice-0.1.0.tgz Status: RUNNING  
values/dev.yaml  
values/prod.yaml  
107995894928.dkr.ecr.us-west-...  
Version v9  
Details

Duration: 02:26

Bake dev Deploy dev Manual Judgment Deploy prod

Bake prod

**STAGE DETAILS: BAKE DEV**  
Duration: 00:03

Step	Started	Duration	Status
Bake dev	2019-06-02 05:40:07 PDT	00:03	SUCCEEDED

**BAKE DEV**

Task Status	Artifact Status
Task	Duration
Create Bake	00:03
Bind Produced Artifacts	00:00

Source | Permalink

Click on Continue and the pipeline will continue to push the deployment to prod environment.

Congratulations! You are up and running with your Spinnaker pipeline. You can refer to the official [Spinnaker Guide](#) to learn more.

## 10. Teardown

Once you are done testing you can clean up by following these steps:

## 10.1 Delete Helm chart

```
helm delete spinnaker --purge
```

This will delete all resources associated with Spinnaker Helm deployment.

## 10.2 Delete ingress

```
kubectl -n spinnaker delete ingress spinnaker
```

This will delete the Spinnaker ingress.

## Conclusion

In this post, we showed you how to install Spinnaker and create a continuous delivery pipeline. We also shared various Spinnaker concepts and the different kinds of stages that can be used for building a pipeline. While this pipeline is very simple, Spinnaker supports many other things, such as rollbacks and canary deployments. It can integrate with CI tools like Jenkins and [Travis CI](#). It can also integrate with [Prometheus](#) and [SignalFx](#) for canary analysis. To learn more about Spinnaker's capabilities, see the awesome [Spinnaker documentation](#).

Happy deploying!