

# Multi-architecture devOps using OpenShift

---

In this lab you will learn how to deploy a Jenkins pipeline to build your source code from github and deploy it to both OpenShift on Intel and OpenShift on IBM Z/LinuxONE. While there are several other steps in a devOps process, we will only focus on the deployment aspect here.

- [Multi-architecture devOps using OpenShift](#)
  - [ID Prerequisites](#)
  - [Environment](#)
  - [Topology Diagram](#)
    - [LinuxONE Community Cloud](#)
  - [What is a multi-architecture deployment anyway?](#)
    - [Dockerfile](#)
    - [Multi-architecture Manifests](#)
    - [Building multi-arch images](#)
    - [Combining multi-arch images and manifests](#)
  - [Container Registries](#)
  - [DevOps ecosystem](#)
    - [Jenkins](#)
      - [Nodes](#)
  - [Application](#)
  - [Putting it all together](#)
  - [Tips for multi-architecture builds:](#)
  - [IBM Multicloud Manager](#)

---

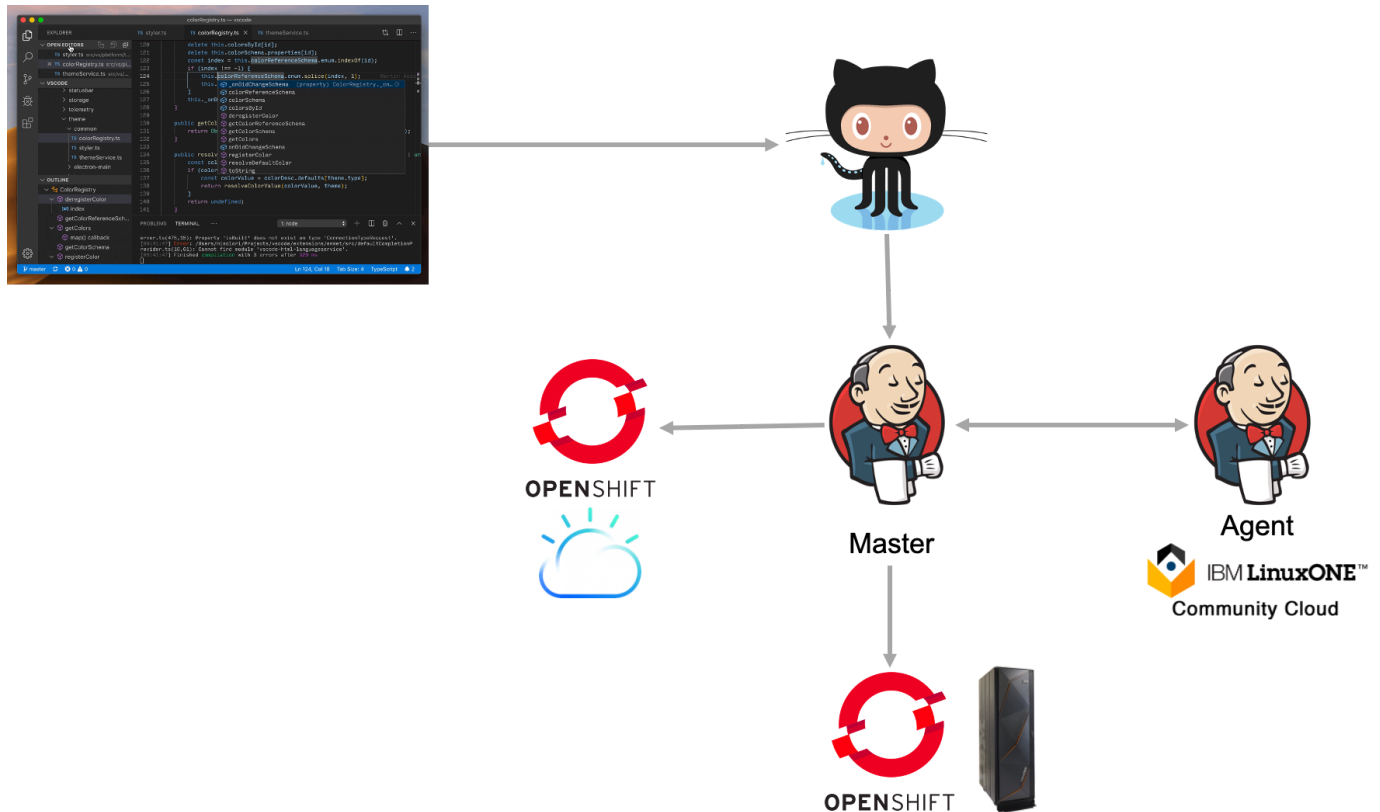
## ID Prerequisites

- [GitHub](#)
- [Docker](#)
- [IBM Cloud](#)
- IBM Washington System Center (will be distributed as part of the lab)
- [LinuxONE Community Cloud](#) (optional, but useful for self paced lab)

## Environment

- RedHat OpenShift (ROKS) on IBM Cloud
- RedHat OpenShift on IBM LinuxONE (in IBM Washington System Center)
- Jenkins (in IBM Washington System Center)
- IBM Container Registry on IBM Cloud
- Jenkins agent on [IBM LinuxONE Community Cloud](#)

## Topology Diagram



**Note:** Using the Kubernetes Jenkins plugin or OCP native Jenkins or other cloud native devOps pipeline tooling would enable even fewer moving parts.

You can run the Jenkins master itself on one of the clusters and the agent in another OCP cluster, reducing the need for 2 separate VMs. It will be much easier to manage/scale and Jenkins kubernetes plugin can even create ephemeral agents just to build and then destroy if needed. As you will note in the [Jenkins](#) section below, the goal of this lab is to go through each step as a learning exercise and to stay away from opinionated CI/CD tooling.

TODO : More on L1CC

## What is a multi-architecture deployment anyway?

A multi-architecture deployment is a deployment that lets you consume the same image (e.g hello-world:latest) on any platform using the same deployment artifacts (pod definitions, deployments, services, routes etc). This greatly simplifies the deployment process while letting an organization optimize for metrics like:

- Cost
- Throughput
- Latency
- Security and Compliance
- Scalability
- Resiliency and reliability
- Uptime

Platforms include:

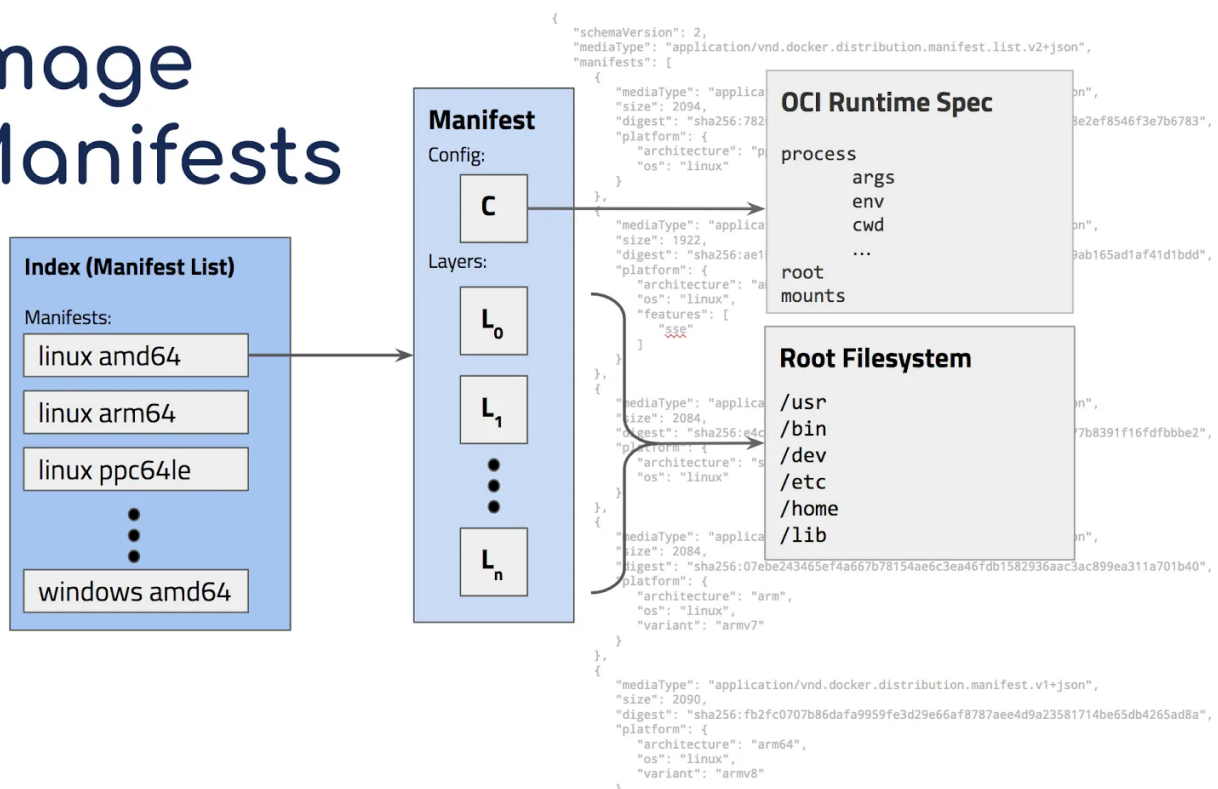
- Operating System (windows, linux etc)
- Instruction Architecture (**amd64**, **s390x**, ppc64, arm, arm64 etc)

## Dockerfile

Writing a simple Dockerfile is easy, but we thought including some best practices here will help, esp since it will help speed up the multi-arch build process.

## Multi-architecture Manifests

# Image Manifests



To enable multi-architecture, docker added support for manifests which let you link which platform to image (but exposing the end result as the same image). e.g "docker run hello-world" will first look at the version (**latest** is implied if no version tag is specified) then will check the local operating system and architecture (e.g linux, s390x) and query that combination in the registry. Once it finds that combination, it'll pull *only that specific container* locally. Multi-arch images are similar to "fat binaries" at the container registry level but single, os and architecture specific images at the docker daemon level.

By default the Docker daemon will look at its current operating system and architecture but it is possible to force download of a specific platform/architecture using the **--platform** command which is available in docker API **1.32+** and need **experimental features** turned on in Docker daemon. The full specification of multi-architecture manifests can be found [here](#). More information on **docker pull** be found in the official docs [here](#).

## Building multi-arch images

Images are just binaries and as such, require to be built on the appropriate platform (build architecture = destination architecture). There are 2 ways of building multi-arch images:

- docker **buildx** builder
- docker default builder

The buildx builder is the most convenient mechanism but can be very slow for non-native architectures as it is emulating the target architectures ISA in qemu. Docker's default builder is the most popular and is used in production by almost every organization building multi-arch images, but needs to run on the destination architecture while building.

## Combining multi-arch images and manifests

The first step is to build the containers on each architecture and store them in a single location. You could push them separately once the manifest is pushed to the repo.

We build these images with these specific tags as it'll make mapping architectures to containers easier.

```
thinklab/go-hello-world:x64-latest  
thinklab/go-hello-world:arm32-latest  
thinklab/go-hello-world:s390x-latest
```

Next, we create a manifest which contains each of these images:

```
docker manifest create thinklab/go-hello-world:latest \  
thinklab/go-hello-world:x64-latest \  
thinklab/go-hello-world:arm32-latest \  
thinklab/go-hello-world:s390x-latest
```

Now let's review the output of:

```
docker manifest inspect thinklab/go-hello-world:latest
```

```
{
  "schemaVersion": 2,
  "mediaType":
"application/vnd.docker.distribution.manifest.list.v2+json",
  "manifests": [
    {
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "size": 3254,
      "digest": "sha256:....",
      "platform": {
        "architecture": "s390x",
        "os": "linux"
      }
    },
    {
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "size": 3038,
      "digest": "sha256:....",
      "platform": {
        "architecture": "arm",
        "os": "linux"
      }
    },
    {
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "size": 2824,
      "digest": "sha256:....",
      "platform": {
        "architecture": "amd64",
        "os": "linux"
      }
    }
  ]
}
```

The manifest command automatically picked up and annotated the architecture and os for each image.

You could also manually annotate with:

```
docker manifest annotate thinklab/go-hello-world:latest \
thinklab/go-hello-world:s390x-latest --arch s390x --os linux
```

If we want to update the images referenced in the manifest, we could rebuild and tag appropriately, then run:

```
docker manifest create thinklab/go-hello-world:latest \  
--amend thinklab/go-hello-world:x64-latest \  
--amend thinklab/go-hello-world:s390x-latest
```

Finally to push to your container registry:

```
docker manifest push thinklab/go-hello-world:latest
```

Now you can do a docker pull on either Intel, ARM or IBM Z (s390x) and it will automatically pull the right container. This also applies to Kubernetes.

```
docker pull thinklab/go-hello-world
```

## Container Registries

When doing multi-arch, not all container registries are created equal. There are a few factors that might be important to help make a selection.

Name	Supports multi-architecture manifests	Certified for LinuxONE	Independent Product
OpenShift Container Registry	✓	✓	
Quay	✓		✓
Docker Trusted Registry	✓	✓	
jFrog Artifactory	✓		✓
Gitlab	✓		✓

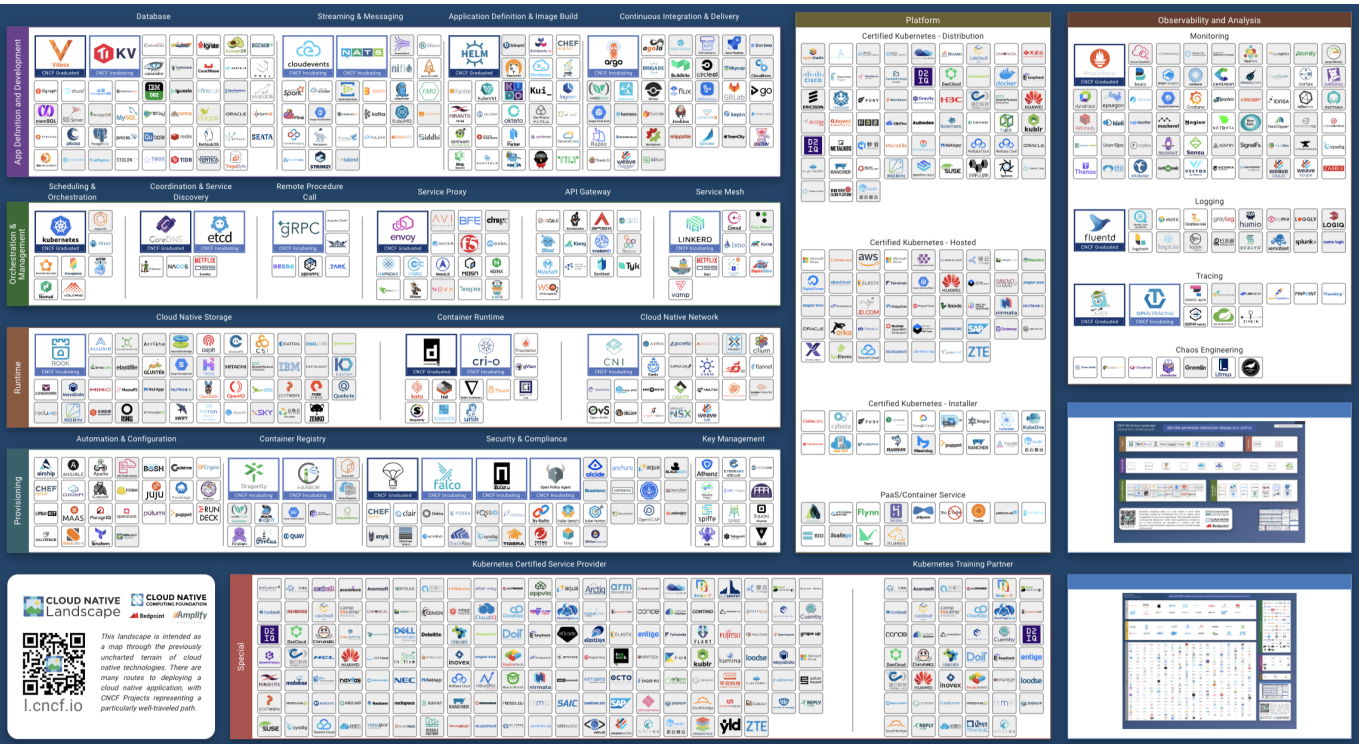
*Supports multi-architecture manifests:* Not all registries support multi-arch manifests, the ones in the list do as they are relatively mature projects, but there are several in the wild that do not.

*Certified for LinuxONE:* A registry might be able to store images tagged for the s390x architecture but the registry itself might not run on a LinuxONE server. This is important as several organizations prefer to run their mission critical workloads and databases on LinuxONE and since a container registry stores images used in production, LinuxONE support is favorable.

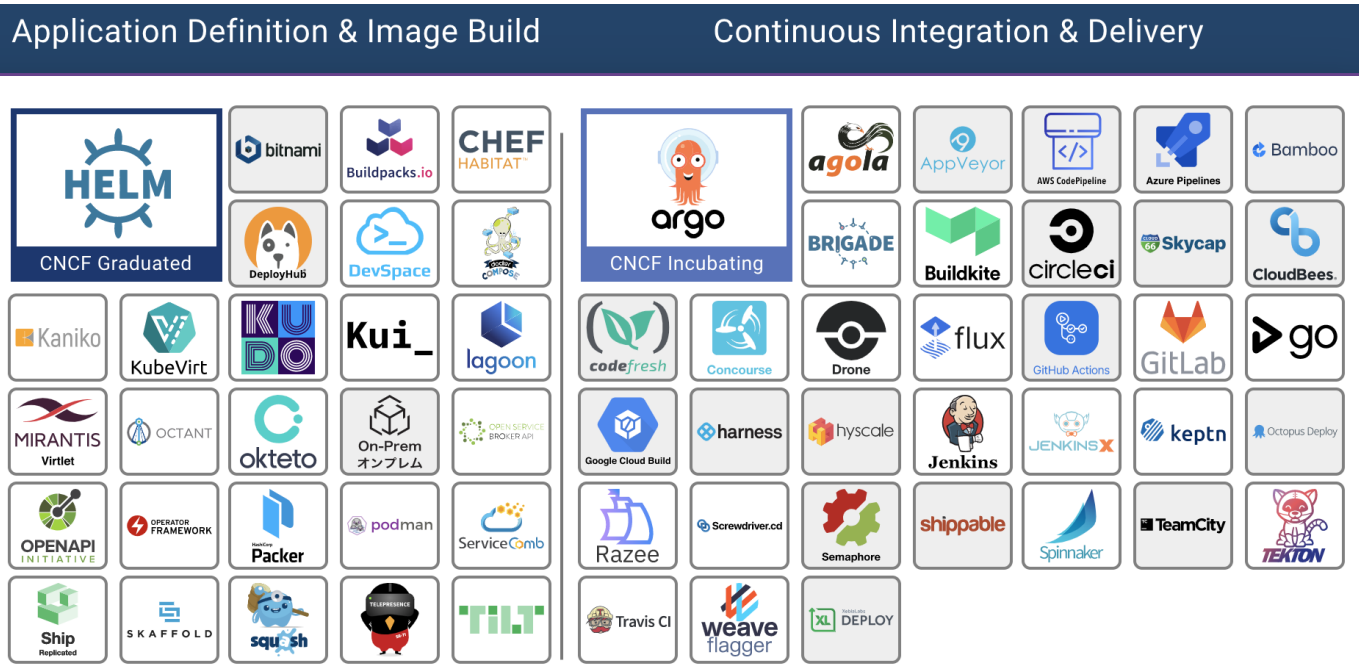
*Independent Product:* A registry might come bundled with a larger set of products. Independent Product implies it can run on its own, without an overarching product.

## DevOps ecosystem

This is a map of the CNCF ecosystem around containers:



Drill down to the build and delivery section:



As there are many options, each doing things in its own opinionated way, we will use Jenkins for this lab which will give you a foundation to build upon and consume the other tools.

## Jenkins



We will not cover Jenkins setup as it is a "household name" in the world of modern delivery pipelines. More information can be found at Jenkins [official](#)

*Useful plugins to install:*

- SSH
- Kubernetes
- OpenShift Jenkins Pipeline
- OpenShift Login

**Note:** While using Jenkins plugins will make this *much* easier, we will do ***deployments the hard way*** as a learning exercise in this lab. We will just be using Jenkins as a glorified remote bash scripts runner, so every step is clear.

Other tools such as Tekton, JenkinsX, Razee etc make this much easier as they were built for kubernetes CI/CD. Cloud providers offer their own build tooling and now even GitHub offers native CI/CD with GitHub Actions. You will most likely use these other tools in production environments.

## Nodes

If you don't specify a node, Jenkins will run the stage on any node. If you only use one architecture the default behavior might be ok but for multi-arch builds, you need to build on the deployment architecture if you use the native docker builder (**docker build**). Note, the **s390x** here is a label you must explicitly give your Jenkins nodes. You can call the node what ever you want but having the architecture in its name will help for this lab.

*Jenkins has no innate architectural recognition capability.*

```
node ('s390x') {
    stage('Source') {
        // Get some code from our Git repository
        git 'https://github.com/IBM/node-s2i-openshift.git'
    }
    stage('Build') {
        // Build the container
        docker build -t node .
    }
    ..
}
```



Mixed node pipelines are also possible:

```
node ('s390x') {  
    ..  
}  
node {  
    ..  
}  
node ('prod') {  
    ..  
}
```

Here, the stages in the middle `node {}` can run on any node, the stages on the `node ('prod')` will run on any nodes with prod label and of course, `node ('s390x')` will run on our node labeled s390x for this lab.

## Application

We will be using a simple hello-world Go web-server, that provides some interactivity in terms of its output across code changes. We will be using a [Go app](#) that prints `Hello, World!` on `http://localhost:8080`. This will also be a good test for Routing in OpenShift.

## Putting it all together

1. Fork the code into your GitHub repo
2. Modify the Jenkinsfile and replace `["GIT REPO HERE"]` to use your repo
3. Copy the contents of the Jenkinsfile to your Jenkins job
4. Run the job
5. See your output at `https://ip:port` for your ROCKS cluster and `https://ip:port` for your OCP on Z cluster (links to both will be shown as part of the job)
6. clone the repo you forked in step 1.
7. Change the `Hello World` on Line 14 in the code to anything you prefer and commit and push your code to github

```
func HelloServer(w http.ResponseWriter, r *http.Request) {  
    fmt.Fprintf(w, "Hello, World!") ← Modify this  
}
```

8. Watch the Jenkins dashboard for activity and follow the links at end to get connectivity information to your code

As this job uses Github hooks, it'll automatically build after step 7.

Note, how our [Jenkinsfile](#) has a mix of `node ('s390x')` and `node ('amd64')`.

## Tips for multi-architecture builds:

- Use multi-architecture base images. The official images on RedHat Container Registry and dockerhub of popular run-times are multi-arch enabled, signed and supported by their vendors (e.g IBM for the popular WebSphere Liberty runtime)
- Use multi-stage builds. You will store as many copies of binaries as architectures, so image size can creep up quickly
- Optimize for prod by stripping debug symbols, using UPX etc
- Use native architecture build environments instead of `buildx` for speed

**Thats it ! You can now build your multi-architecture deployment pipeline on OpenShift across any public and private cloud environment ! 🎉**

# IBM Multicloud Manager

TODO : More on IBM MCM

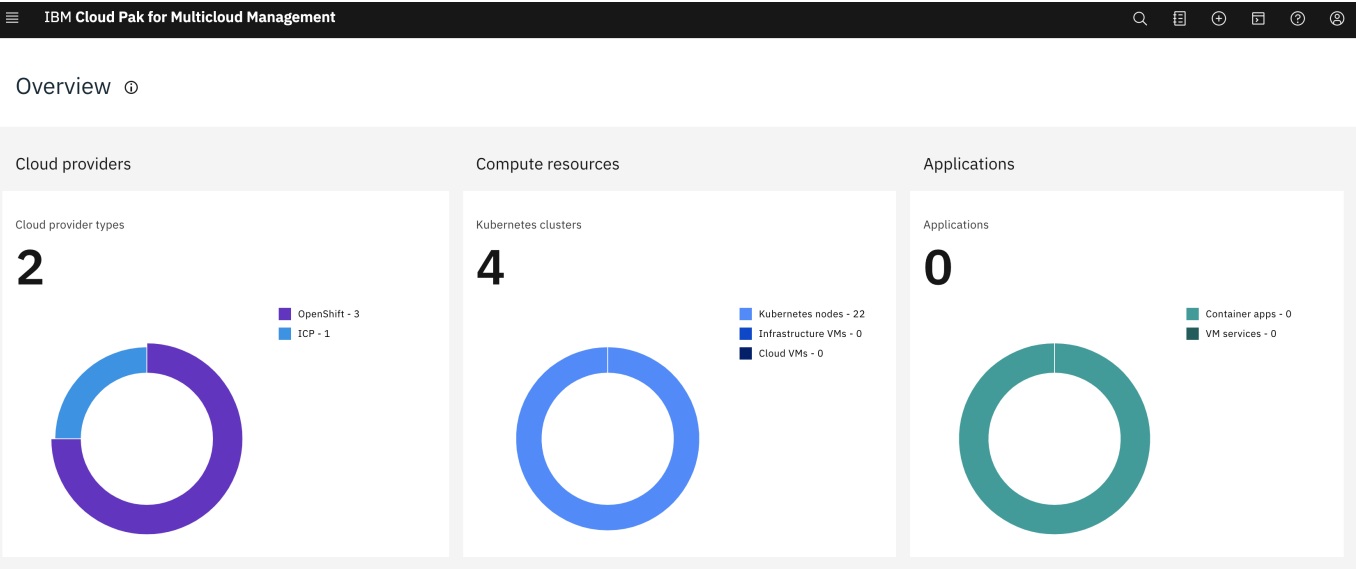
Using MCM you could add both OCP on Intel and Z clusters, setup a `podPlacementPolicy` and deploy your app to MCM and let MCM decide the best place to deploy it.

```
apiVersion: mcm.ibm.com/v1alpha1
kind: PlacementPolicy
metadata:
  name: prod-placement-policy
  namespace: mcm
spec:
  clusterLabels:
  matchLabels:
    cloud: ibm-linuxone
```

More details on placement policies can be found in the official IBM Multicloud Manager [documentation](#)

Our MCM cluster is setup with several kubernetes based PaaS clusters.

This shows the summary in a GUI:



Cluster(s) health in a single view:

The screenshot shows the 'Clusters' page of the IBM Cloud Pak for Multicloud Management GUI. It displays a table with 7 columns: Name, Namespace, Status, Nodes, Kusterlet version, Kubernetes version, and Labels. There are 4 rows of cluster data. The table also includes a search bar, a 'Find' button, and a 'Add cluster' button.

Name	Namespace	Status	Nodes	Kusterlet version	Kubernetes version	Labels
hubcluster	hubcluster	Ready	4	3.3.0	v1.16.2+rhos	cloud=IBM +1
icp1	icp1	Ready	5	3.3.0	v1.13.5+icp-ee	cloud=Other +1
ocp1	ocp1	Ready	7	3.3.0	v1.16.2+rhos	cloud=Other +1
ocpz	ocpz	Ready	6	3.3.0	v1.14.6-152-g117ba1f+rhos	cloud=Other +1

Cross-cluster catalog:

IBM Cloud Pak for Multicloud Management

Catalog

Q Search the catalog...

All Categories >

DevOps

Operations

Security

Data

IoT

Integration

Data Science & Analytics

AI & Watson

Runtimes & Frameworks

Storage

Blockchain

Business Automation

Network

Tools

Other

Classification Cloud Platform Architecture Qualification Repositories Reset all

Cloud Paks

ibm-icp4i-prod

ibm-entitled-charts

A Helm chart for the IBM Cloud Pak for Integration Navigator

Helm Charts

aqua-enforcer

ibm-community-charts

A Helm chart for the Aqua Enforcer

aqua-scanner

ibm-community-charts

A Helm chart for the aqua scanner cli component

aqua-server

ibm-community-charts

A Helm chart for the Aqua Console Components

artifactory-ha

ibm-community-charts

Universal Repository Manager supporting all major packaging formats, build tools and CI servers.

audit-logging

mgmt-charts

Audit logging storage and search management solution

auth-apikeys

mgmt-charts

ICP IAM Token Service

Details about our specific OCP on Z cluster

IBM Cloud Pak for Multicloud Management

Clusters /

ocpz

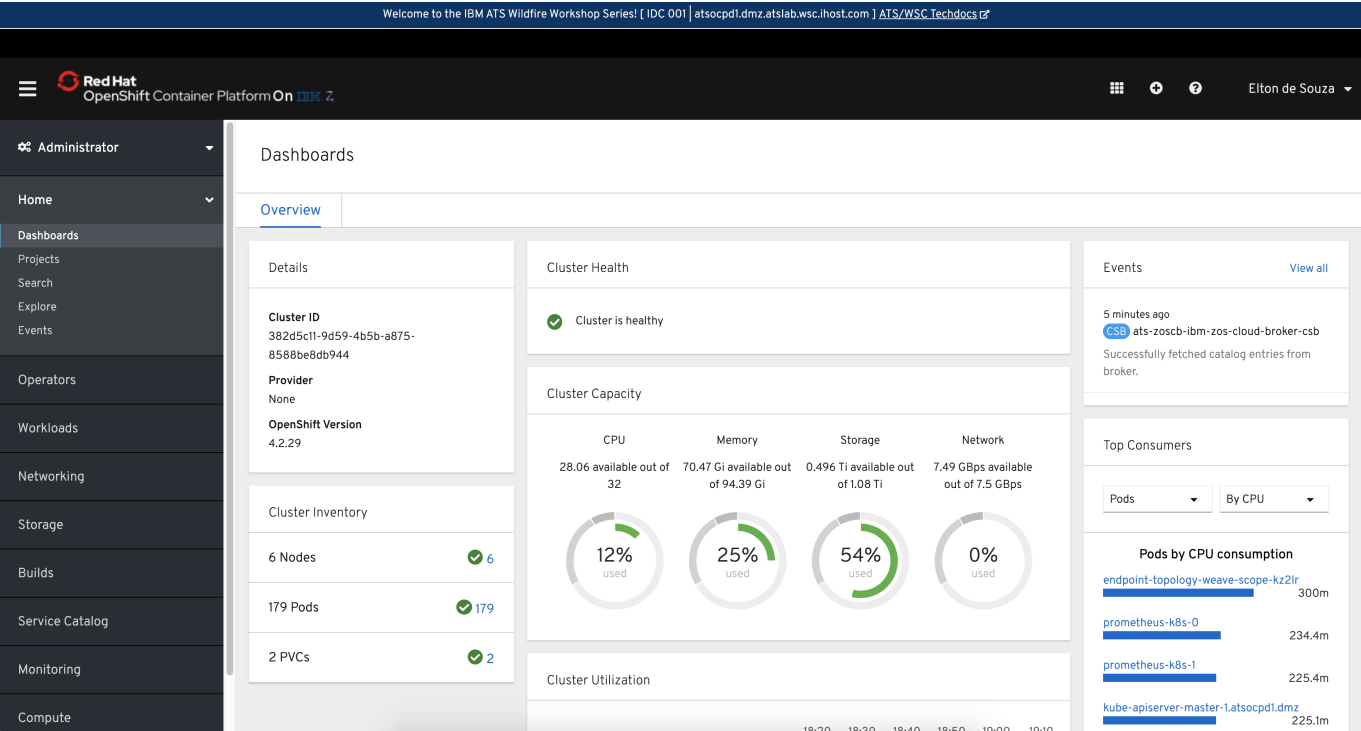
Overview Nodes

Q Find

Name	Role	Zones	Nodes for each of the zones	Size - Core and memory
master-0.atsocpd1.dmz	master, worker	-	-	4/15.73Gi
master-1.atsocpd1.dmz	master, worker	-	-	6/15.73Gi
master-2.atsocpd1.dmz	master, worker	-	-	4/15.73Gi
worker-0.atsocpd1.dmz	worker	-	-	6/15.73Gi
worker-1.atsocpd1.dmz	worker	-	-	6/15.73Gi
worker-2.atsocpd1.dmz	worker	-	-	6/15.73Gi

Items per page 20 | 1-6 of 6 items1 of 1 pages<>

Clicking on the endpoint will take you to the OCP on Z Cluster:



In this scenario, I deployed WebSphere Liberty on OpenShift though IBM Multicloud Manager:

