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
  - Environment
  - ID Prerequisites
  - What is a multi-architecture deployment anyway?
    - Multi-architecture Manifests
    - Building multi-arch images
    - Combining multi-arch images and manifests
  - Container Registries
  - DevOps ecosystem
    - Jenkins
      - Nodes
  - Topology Diagram
  - Application
  - Putting it all together
  - IBM Multicloud Manager

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

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)

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:

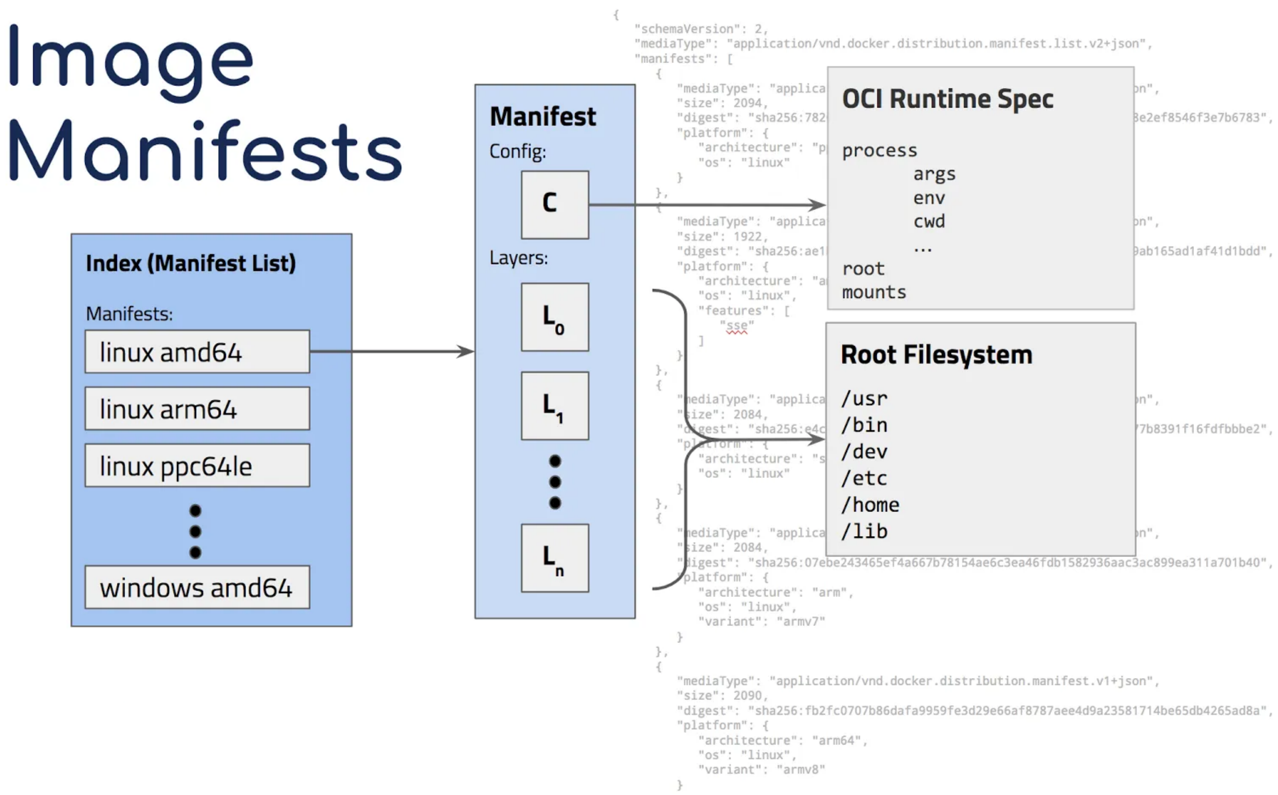
- Cost
- Throughput

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

Platforms include:

- Operating System (windows, linux etc)
- Instruction Architecture (**amd64**, **s390x**, ppcle64, arm, arm64 etc)

Multi-architecture 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 fins 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 builx builder is the most convenient mechanism but can be 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.

Combining multi-arch images and manifests

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

```
thinklab/go-ascii-banner:x64-latest
thinklab/go-ascii-banner:arm32-latest
thinklab/go-ascii-banner:s390x-latest
```

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

```
docker manifest create thinklab/go-ascii-banner:latest \
thinklab/go-ascii-banner:x64-latest \
thinklab/go-ascii-banner:arm32-latest \
thinklab/go-ascii-banner:s390x-latest
```

Now let's review the output of:

```
docker manifest inspect thinklab/go-ascii-banner: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-ascii-banner:latest \
thinklab/go-ascii-banner: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-ascii-banner:latest \
--amend thinklab/go-ascii-banner:x64-latest \
--amend thinklab/go-ascii-banner:s390x-latest
```

Finally to push to your container registry:

```
docker manifest push thinklab/go-ascii-banner: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-ascii-banner
```

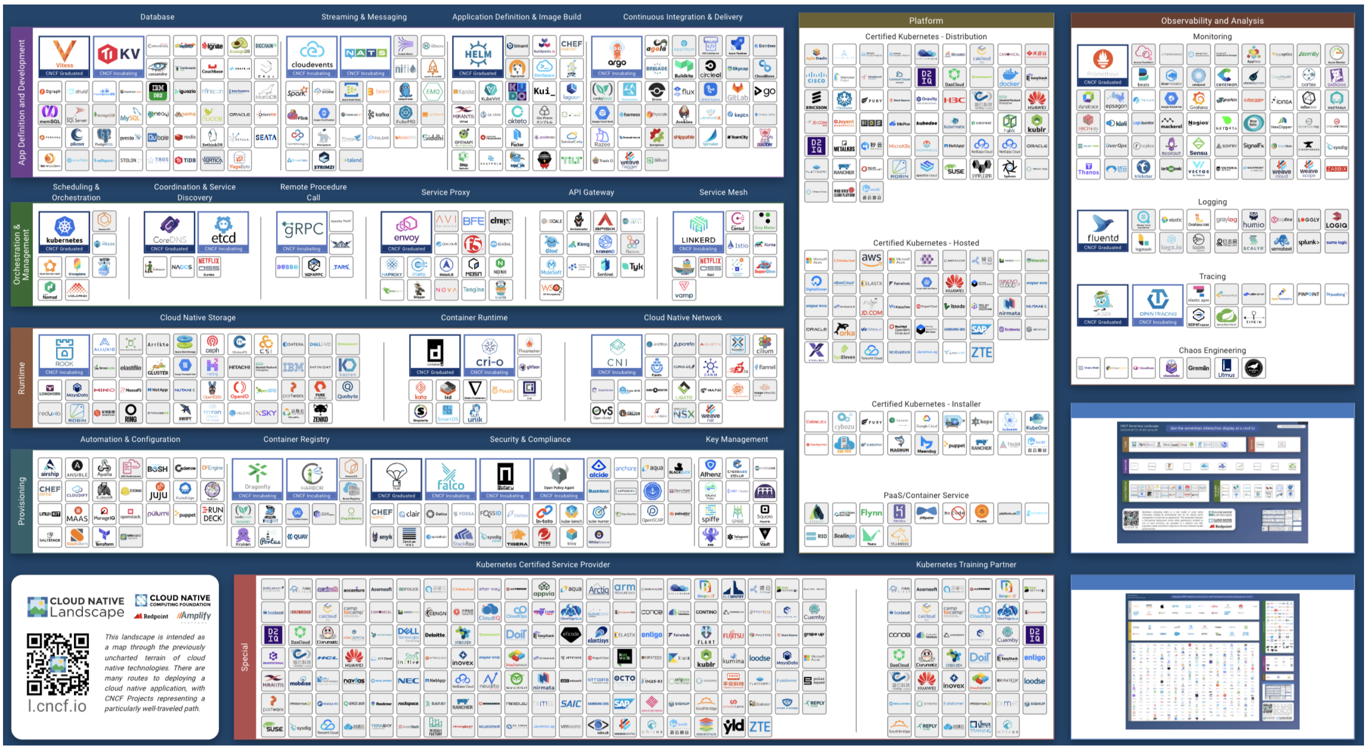
Container Registries

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.

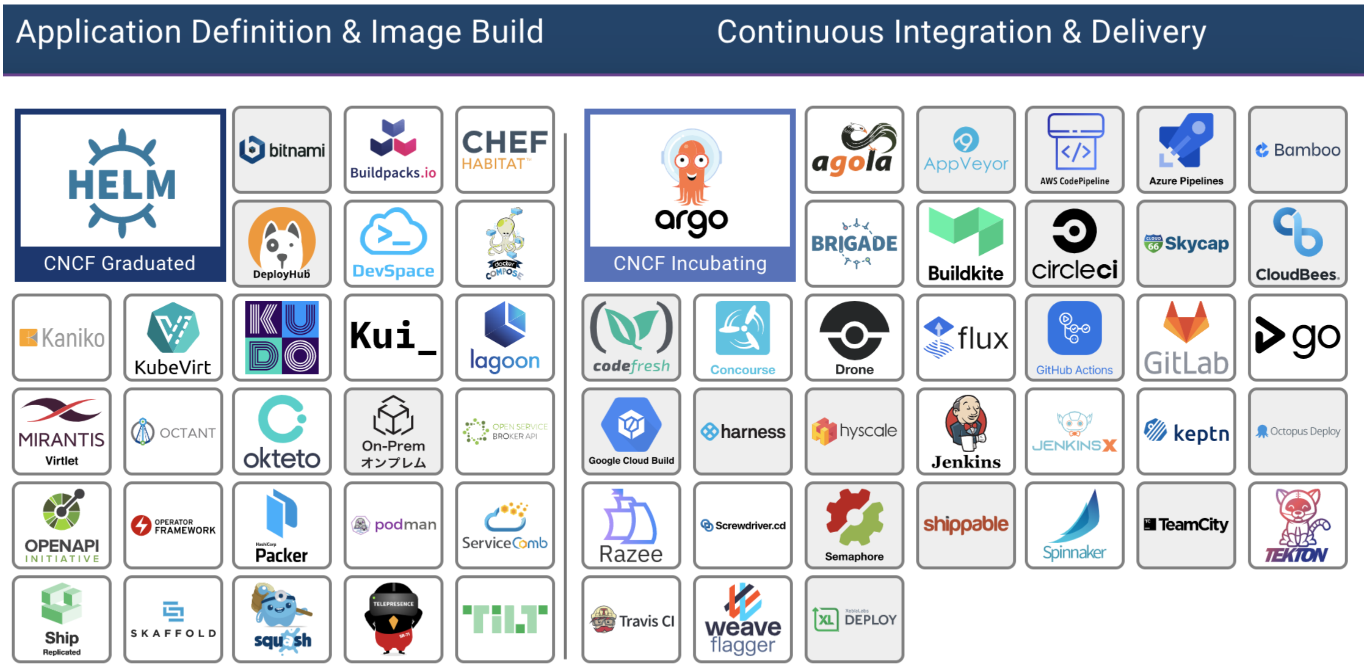
Name	Supports multi-architecture manifests	Certified for LinuxONE	Independant Product
OpenShift Container Registry	✅	✅	
Quay	✅		✅
Docker Trusted Registry	✅	✅	
jFrog Artifactory	✅		✅
Gitlab	✅		✅

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 .
    }
    ..
}
```

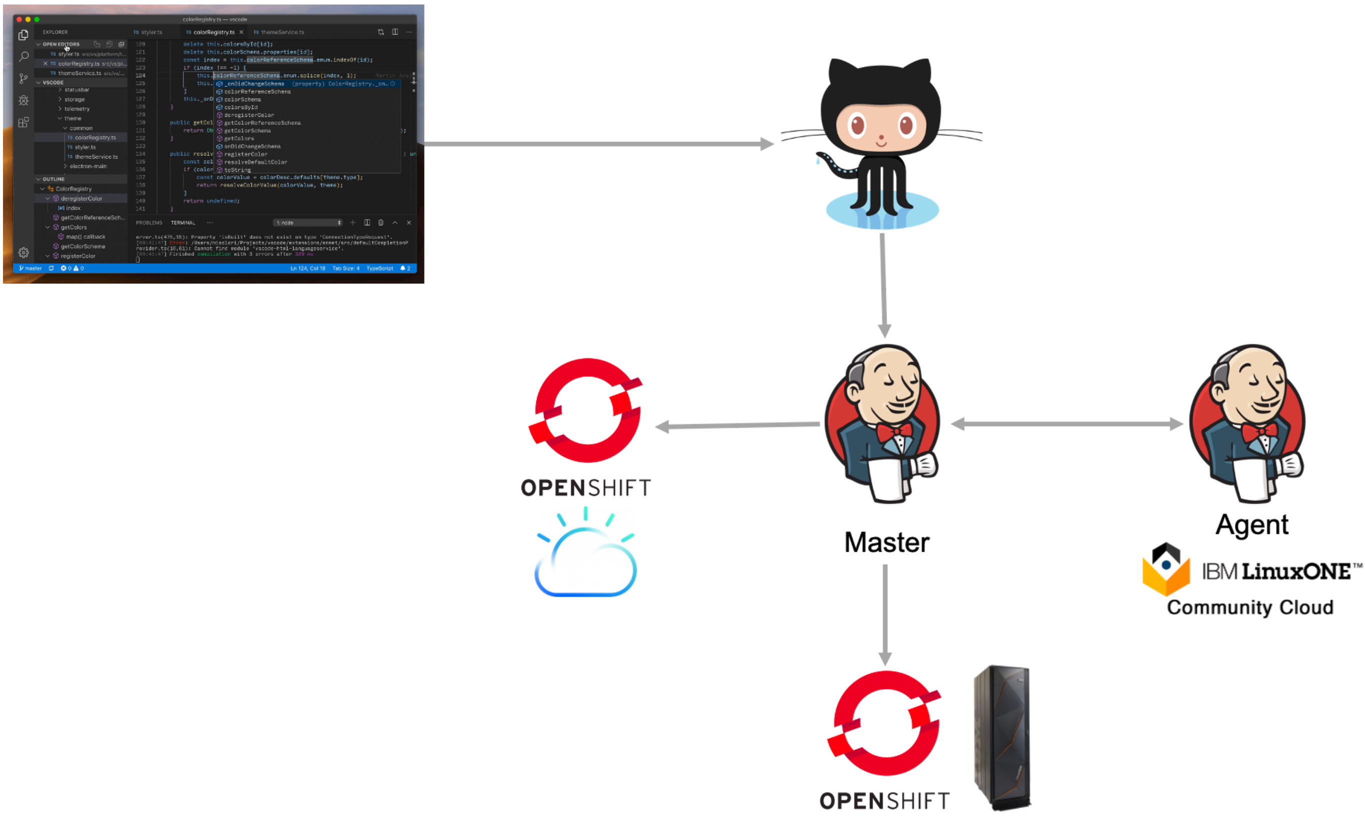
Mixed node pipelines are also possible:

```
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.

Topology Diagram

The lab follows this topology:



> **Note:** Using the kubernetes Jenkins plugin or OCP native Jenkins or other cloud native devOps pipline 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.

Application

We picked a simple app, that provides some interactivity in terms of its output across code changes. We will be using a Go app that prints ASCII art from text.

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` in the code to anything you prefer and commit and push your code to github

```
myFigure := figure.NewFigure("Hello World", "", true)
```

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')`.

Thats it ! You now now build your multi-architecture deployment pipeline on OpenShift !

IBM Multicloud Manager

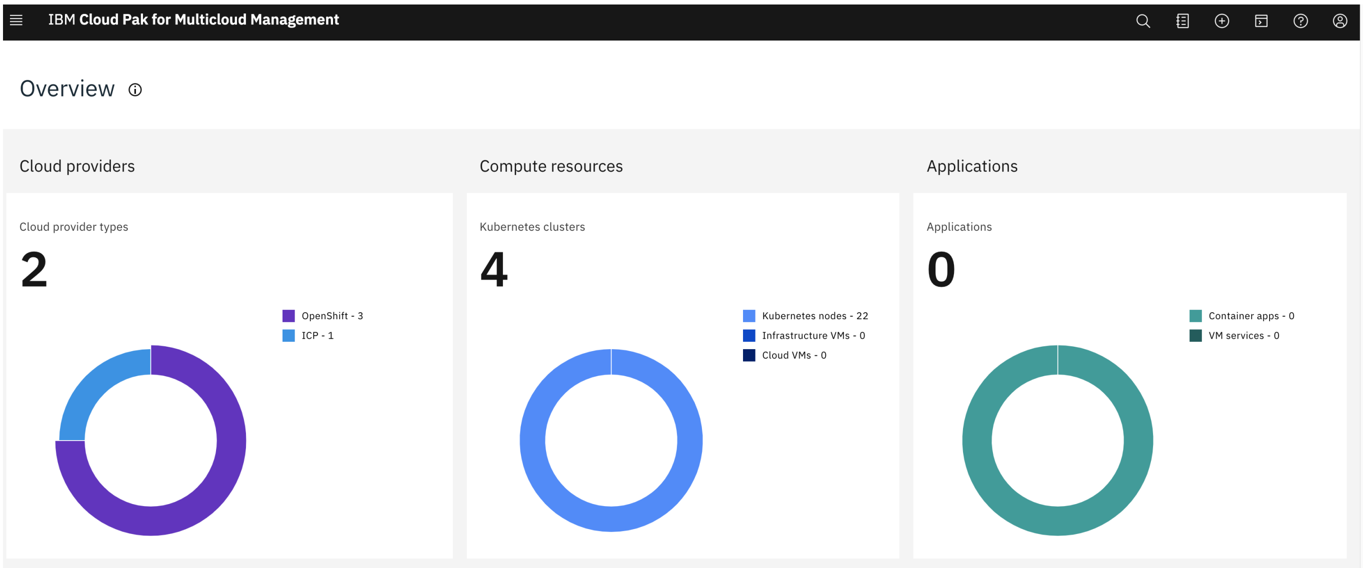
Using MCM you could add both OCP on Intel and Z clusters, setup a `podPlacementPolicy` and deploy your app to MCM and let MCM decide the best place to deploy it.

```yaml
apiVersion: mcm.ibm.com/v1alpha1
 kind: PlacementPolicy
 metadata:
   name: placement1
   namespace: mcm
 spec:
   clusterLabels:
     matchLabels:
       cloud: ibm-linuxone
```

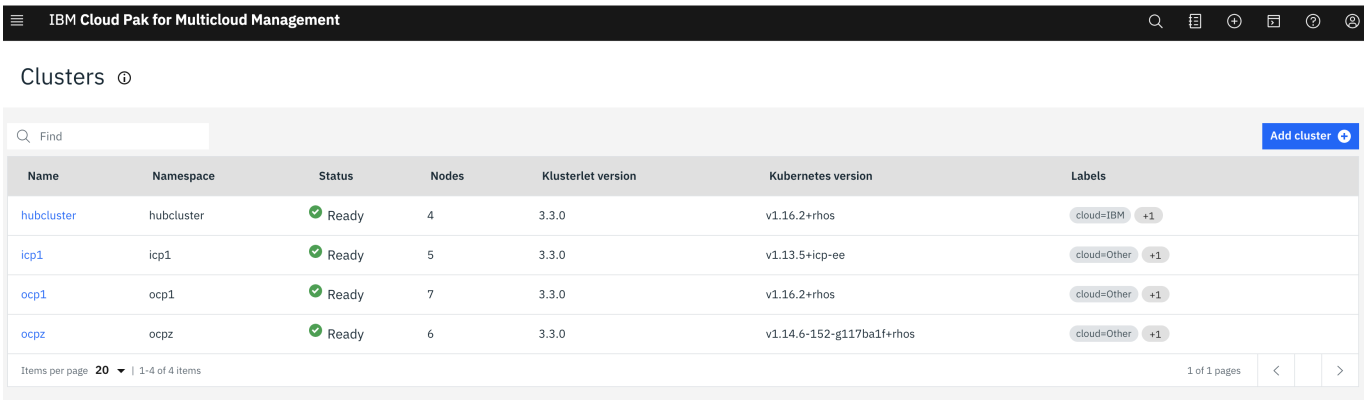
More details on placement policies can be found in the official IBM Multicloud Manager documentation

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

This shows the summary in a GUI:

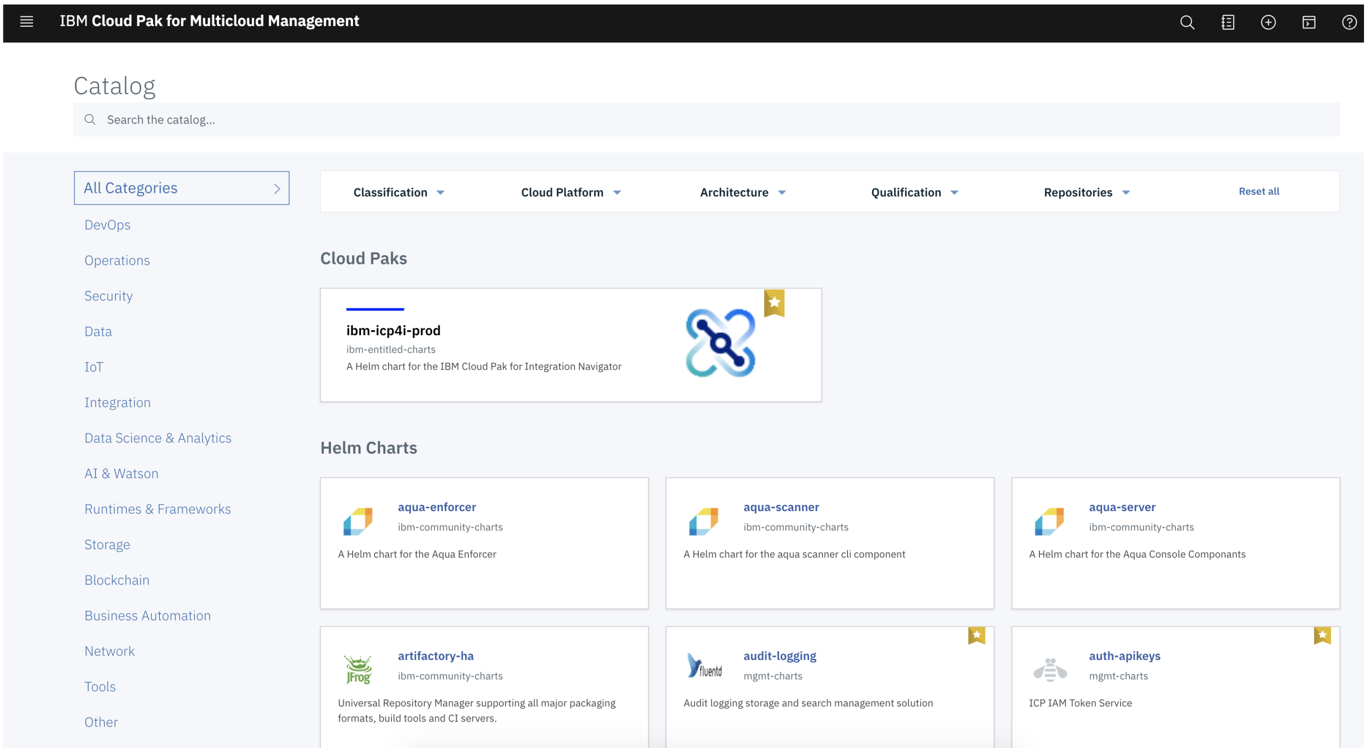


Cluster(s) health in a single view:

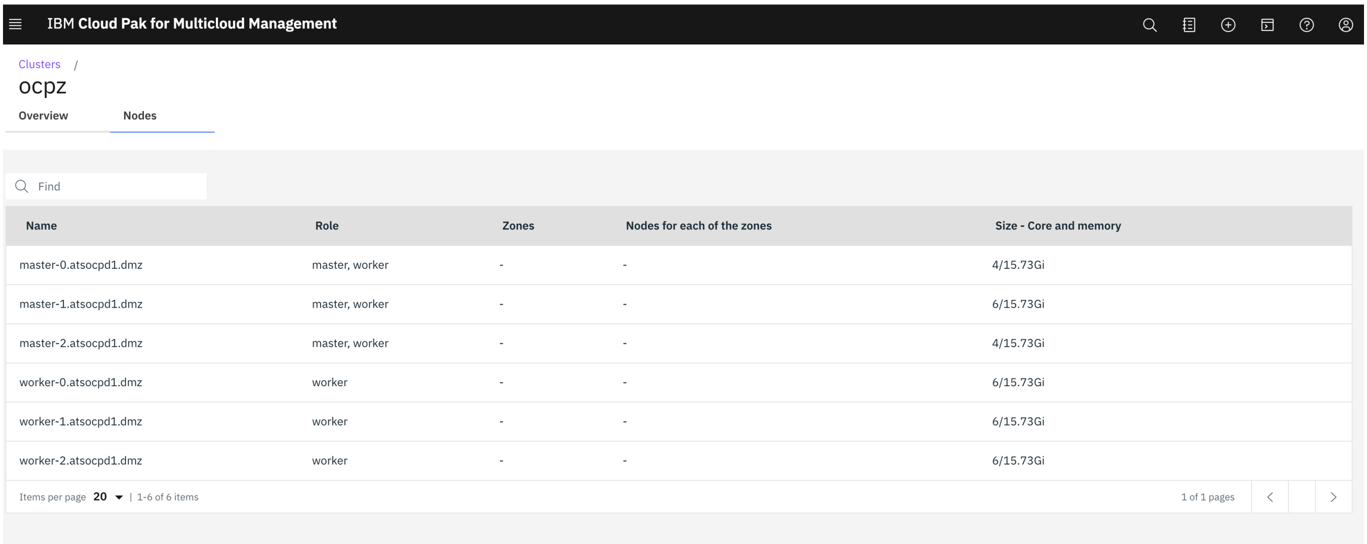


Name	Namespace	Status	Nodes	Klusterlet version	Kubernetes version	Labels
hubcluster	hubcluster	Ready	4	3.3.0	v1.16.2+rhos	cloud=IBM +1
icp1	icp1	Ready	5	3.3.0	v1.13.5+icp-ee	cloud=Other +1
ocp1	ocp1	Ready	7	3.3.0	v1.16.2+rhos	cloud=Other +1
ocpz	ocpz	Ready	6	3.3.0	v1.14.6-152-g117ba1f+rhos	cloud=Other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

Cross-cluster catalog:

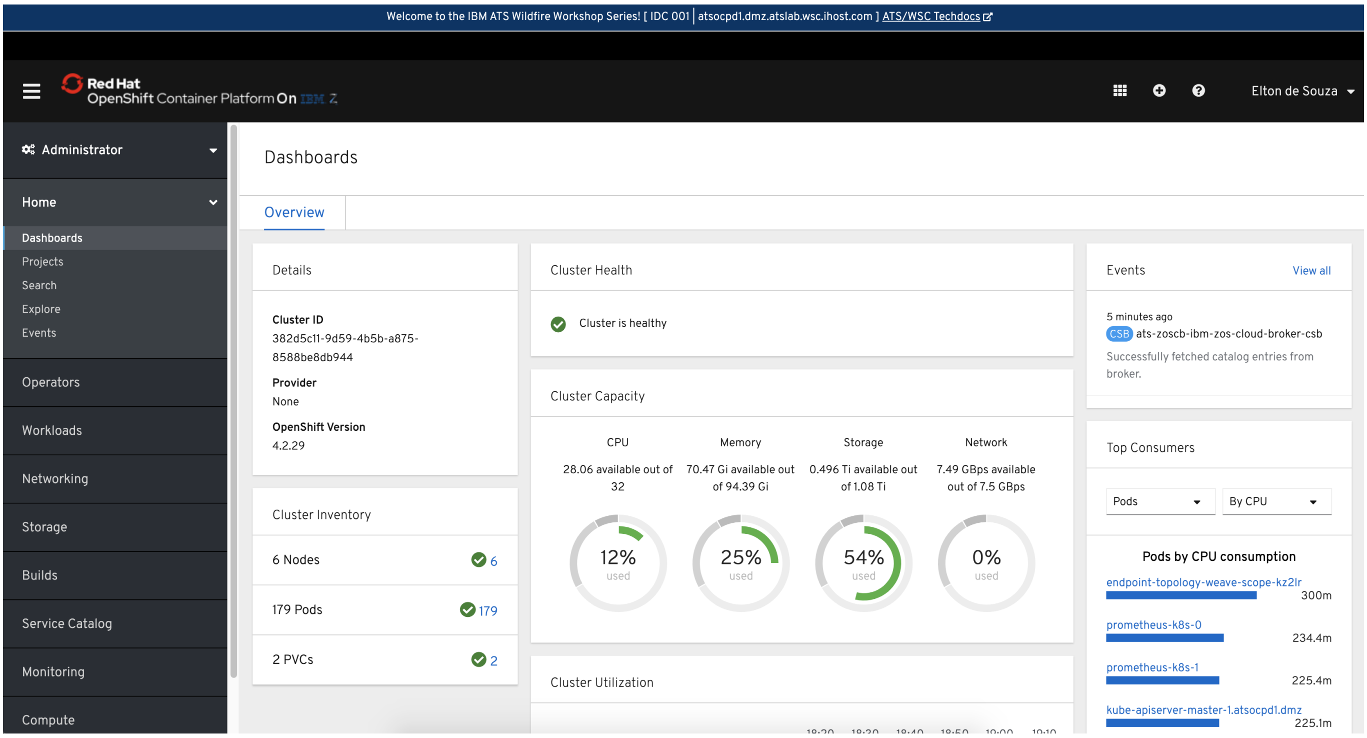


Details about our specific OCP on Z cluster



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

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: