KUBERNETES FOR EPICS IOCS

G. Knap, T. Cobb, Y, Moazzam, U. Pedersen, C. Reynolds,
Diamond Light Source, Oxfordshire, UK

Abstract

EPICS [1] IOCs at Diamond Light Source (DLS) [2] are built, deployed, and managed by a set of in-house tools that were implemented 15 years ago. This paper will detail a proof of concept to demonstrate replacing these legacy tools and processes with modern industry standards.

IOCs are packaged in containers with their unique dependencies included.

Container orchestration for all beamlines in the facility is provided through a central Kubernetes cluster. The cluster has remote nodes dedicated to each beamline that host IOCs on the beamline networks.

All source, images and individual IOC configurations are held in repositories. Build and deployment to the production registries is handled by continuous integration.

Development containers provide a portable development environment for maintaining and testing IOC code.

INTRODUCTION

The approach presented here has 5 main themes:

1. **Containers**: package each IOC with its dependencies and execute it in a lightweight virtual environment. [3]
2. **Kubernetes**: centrally orchestrates all IOCs at the facility [4].
3. **Helm Charts**: deploy IOCs into Kubernetes and provide version management [5].
4. **Repositories**: Source, container and Helm repositories hold all of the assets required to define a beamline's IOCs.
5. **Continuous Integration**: source repositories automatically build containers, Helm charts and deliver them to package repositories.

An initial proof of concept (POC) has been implemented at DLS on the test beamline BL45P. All the source code for the proof of concept, plus documentation and tutorials can be found in the GitHub organization epics-containers [6].

SCOPE

The POC initially targets Linux IOCs. This includes IOCs that communicate with their associated devices over the network, as well as those that connect to local devices through USB, PCIe etc. It does not include provision for Operator Interfaces (OPIs) as these vary greatly between facilities. Future plans include:

1. Support OPIs by having a 2nd container for each IOC instance that serves OPI files over HTTP.
2. Supporting RTEMS hard IOCs: using a containerised developer environment shared with soft IOCs.
3. Support Windows IOC development through a similar approach to RTEMS.

CONTAINERS

A class of IOCs that connect to a particular class of device will all share identical binaries and library dependencies; they will differ only in their start-up script and EPICS database. Thus containerized IOCs may be represented as follows:

1. **Generic IOC**: A container image for all IOCs that will connect to a class of device.
2. **IOC Instance**: a Generic IOC image plus unique instance configuration. Typically the configuration is a single start-up script only.

This approach means that the number of container images is kept reasonably low and they are easier to manage.

Image Layering

Container images are typically built by layering on top of existing images.

For the POC, an image hierarchy is used to improve maintainability as shown in Fig. 1 below.

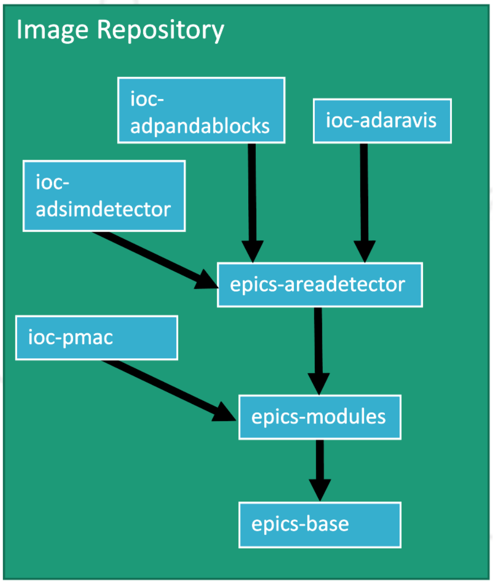

Figure 1: Image hierarchy for the generic IOCs in the current proof of concept.

EPICS base [7] and essential tools are compiled inside one image; the most commonly used support modules (primarily Asyn [8]) and the AreaDetector [9] framework also have their own images. Generic IOC images are then leaves in the hierarchy and are based upon the appropriate dependencies.

Images also have internal layering and every layer is shared between all instances of IOCs, both in image repositories and at runtime. Fig. 2 shows and example of this internal layering.

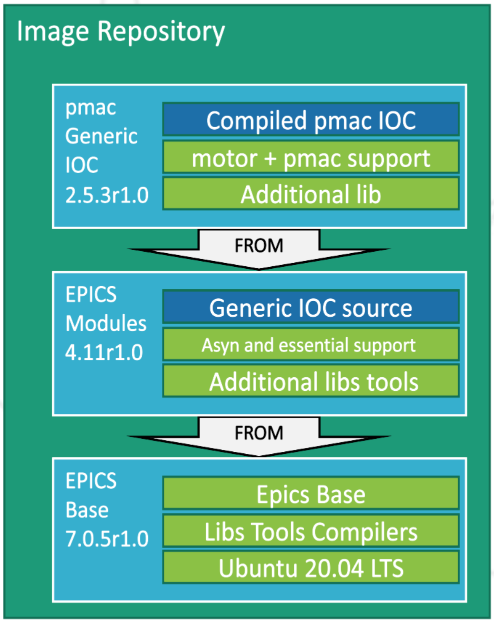

Figure 2: Example of the layered nature of a container image for the pmac motor controller generic IOC.

Developer Container

To save on resources at runtime, all images are built using a staged Dockerfile [10]. There are two targets: a developer target which includes all of the compilers and tools used to build the runtime assets; and a runtime target containing the minimum assets to run the generic IOC.

The developer targets are used to provide an environment for developers to compile and debug the IOC and its dependent support modules. This provides a portable development environment and means there is no need to replicate the tool chain directly on developer workstations.

KUBERNETES

A central Kubernetes cluster is used to orchestrate the IOC instances for all beamlines in the facility. It provides the following functionality that is typically handled by separate tools:

- Auto start IOCs when servers come up
- Manually Start and Stop IOCs
- Monitor IOC status and versions
- Deploy versions of IOCs to the beamline
- Roll back to a previous IOC version
- Allocate the server which runs an IOC
- View the current log
- View historical logs (via graylog at DLS)
- Connect to an IOC and interact with its shell
- Debug an IOC (by starting a bash shell inside its container)

Cluster Topology

A single multi-tenant central cluster runs all of the worker nodes. This provides centralized management of all beamlines and other services. The High Availability (HA) control plane has 3 virtual servers distributed across 3 physical servers.

Each beamline has its own physical servers that are configured as remote worker nodes to the central cluster. This means that:

1. IOC instances are close to the hardware that they communicate with, avoiding network bottlenecks. This is important for high bandwidth IOCs such as area detectors.
2. IOC instances may be given affinity to a specific server and communicate directly with hardware connected to that server (e.g. a USB device)
3. IOC instances reside on the same subnet as the beamline's Channel Access (CA) clients and any network attached devices. This is a requirement for CA and some network attached devices (see below).

Each beamline has its own Kubernetes namespace and user id in which all the IOC instances will run. This provides isolation between the beamlines.

Container Networking

Containers use namespaces to isolate their use of system resources. This is an important feature for building reliable, scalable and secure applications. However, EPICS IOCs rely on network protocols that may not suit network isolation because they do not easily pass through Network Address Translation (NAT). For this reason the POC foregoes virtual networks and uses the native networking of the host server.

Channel Access (CA) and pvAccess (PVA) are the primary protocols for communication between IOCs and clients. Both protocols require a broadcast in order to establish initial communication. The broadcast does not work via NAT to a virtual LAN.

Other protocols between IOCs and devices had issues with virtual networks. e.g. GigE Vision Stream Protocol (GVSP) establishes a connection by passing an IP address and port number in the application layer and therefore does not pass through a NAT.

Workarounds to the protocol issues were investigated on a case by case basis but it became clear that the only reliable solution was to use native networking within a single subnet. This is a slight concession to security, but is no worse than traditional IOC deployment. All other namespaces are applied to IOC containers and they are isolated from the host in all respects except network.

HELM

The POC supplies a Helm Chart Library that describes all of the Kubernetes resources required to deploy an IOC instance to a beamline. Each Beamline has a source repository that specifies a Helm Chart for each of its IOC instances. The beamline source need only refer to the

Library and supply a few parameters to define the unique properties of the IOC instance – most notably its start-up script. The Library has templates for the following resources:

1. A Deployment [11] which makes sure 1 instance of the IOC Pod is always running.
2. The Pod [12] is described within the Deployment YAML. It includes a reference to the Generic IOC image it uses to launch the container.
3. A ConfigMap [13] which is mounted as a folder containing the unique configuration for the IOC instance. This typically contains only a start-up script.
4. A Persistent Volume Claim [14] which is mounted to provide persistent storage across IOC restarts and upgrades. This is used to hold autosave data.
5. Note that there is no Kubernetes Service [15] associated with the IOC since it uses native networking and is addressed via the host's IP address directly.

Helm command line functions are used to deploy IOC instances to a cluster and manage multiple versions of IOCs within the cluster.

Helm charts may be stored in a registry. For the POC the registry is provided by the epics-containers GitHub organization. New versions of IOC Helm charts are released to the registry and then deployed from it. Fig. 3 demonstrates this release process.

REPOSITORIES

All of the assets required to manage a set of IOCs for a beamline are held in repositories. Thus all version control is done via these repositories and no shared file-systems are required. The classes of repository are as follows:

1. **Beamline Source**: (1 per beamline) holds the source for Helm charts for each IOC instance. Also hosts the continuous integration (CI) steps to generate Helm charts and publish them to the Helm repository.
2. **Container Image Source**: (1 per Generic IOC) holds the Dockerfile that describes the contents of a Generic IOC. Also hosts the CI to generate an image from the Dockerfile and publish to the image repository.
3. **Helm Repository**: (1 per IOC Instance) holds the published IOC Instance Helm Charts ready for deployment to Kubernetes.
4. **Image Repository**: (1 per Generic IOC) holds the Generic IOC container images and their dependencies.

CONTINUOUS INTEGRATION

All published assets in this process are generated via CI (see Repositories above). Every published asset has its own version number and its own source repository. When releasing an asset the developer will version tag the source repository. This causes the CI to build the source, publish the result and tag it with the same version number.

The POC uses GitHub Actions [16] to implement its CI and publishes assets to GitHub Packages [17]. However, during development GitLab CI [18] and Google Container Registry [19] were also tested.

CONCLUSION

The POC demonstrates that it is possible to deploy and manage IOCs for a beamline using only standard open source tools such as Kubernetes, Helm and GitHub. The test beamline BL45P has successfully deployed its IOCs using this approach and required no custom software or scripts to do so.

DLS will continue to develop this approach as a possible site wide solution. The epics-containers GitHub organization will be continuously updated to track progress. The organization is also available for contribution from EPICS developers at other sites.

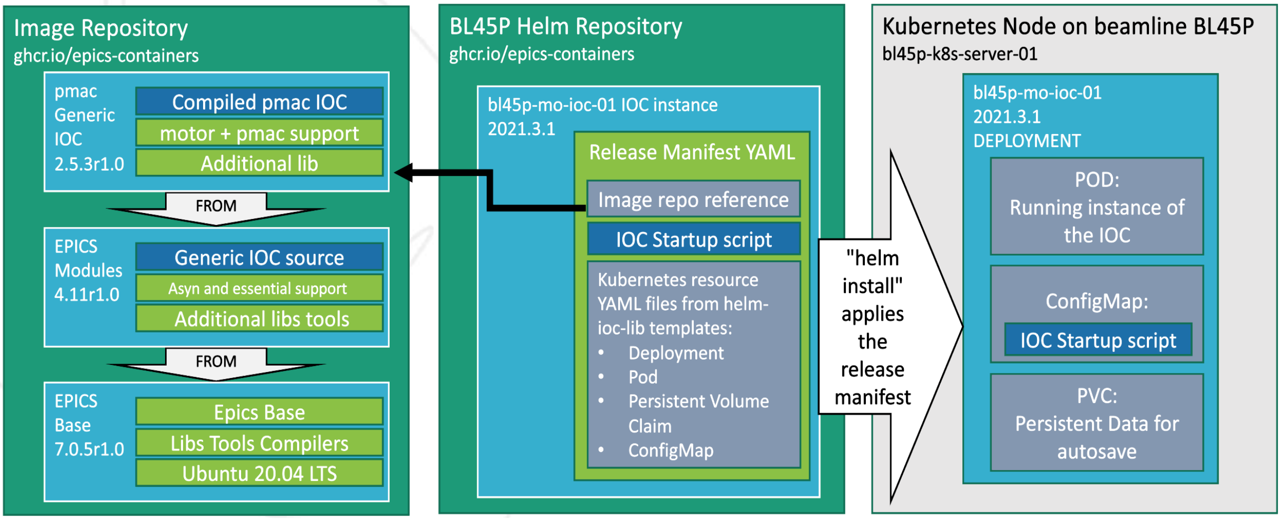

Figure 3: Example of a motion IOC (bl45p-mo-ioc-01) showing how the generic IOC image is built up, packaged with deployment details into a Helm chart, and then deployed to a Kubernetes node.

REFERENCES

[1] EPICS, https://epics-controls.org/

[2] R. P. Walker, "Commissioning and Status of the Diamond Storage Ring", in *Proc. 4th Asian Particle Accelerator Conf. (APAC'07)*, Indore, India, Jan.-Feb. 2007, paper TUYMA03, pp. 66-70.

[3] Open Container Initiative, https://opencontainers.org/

[4] Kubernetes, https://kubernetes.io/

[5] Helm Charts, https://helm.sh/docs/topics/charts/

[6] EPICS Containers, https://epics-containers.github.io

[7] EPICS Base, https://epics-controls.org/resources-and-support/base/

[8] EPICS Asyn, https://epics-controls.org/resources-and-support/documents/howto-documents/device-support-asyn-driver/

[9] EPICS areaDetector, http://cars9.uchicago.edu/software/epics/areaDetector.html.

[10] Staged Docker Builds, https://docs.docker.com/develop/develop-images/multistage-build/

[11] Kubernetes Deployment, https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[12] Kubernetes Pod, https://kubernetes.io/docs/concepts/workloads/pods/

[13] Kubernetes Config Map, https://kubernetes.io/docs/concepts/configuration/configmap

[14] Kubernetes PVC, https://kubernetes.io/docs/concepts/storage/persistent-volumes/

[15] Kubernetes Service, https://kubernetes.io/docs/concepts/services-networking/service/

[16] GitHub Actions, https://docs.github.com/en/actions

[17] GitHub Packages, https://docs.github.com/en/packages

[18] GitLab CI, https://docs.gitlab.com/ee/ci/

[19] Google Container Registry, https://cloud.google.com/container-registry