Prolifics® ACE Migration

A Prolifics How-To Guide

Migrating IBM Integration Bus to IBM App Connect on Amazon EKS Platform Using Operators

By Raveendra Nanjundappa, Prolifics Integration Solution Architect

Table of Contents

Back to TOC

Introduction

Migrating IBM Integration Bus V10 development resources to IBM App Connect Enterprise

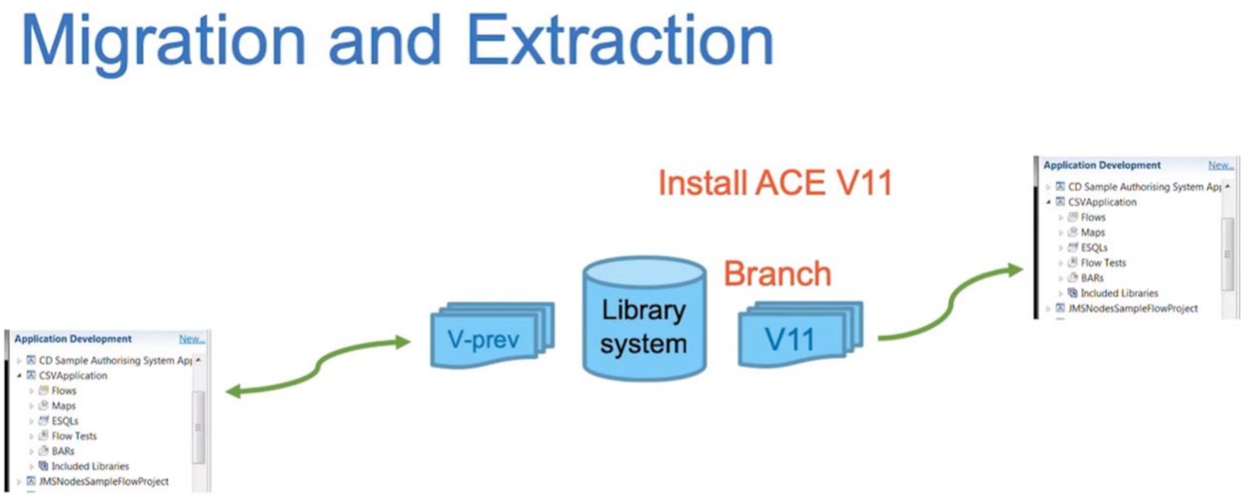
As more and more corporations are moving towards cloud deployment, IBM has taken the right step to enable its integration stack to be cloud native. One such integration component is IBM App Connect Enterprise (ACE). IBM ACE is the one-stop solution to connect any application. In this article we will get to know how to migrate IBM Integration Bus (IIB) to ACE on Kubernetes platform, specifically the AWS EKS environment.

Section 1 – Migrating from IIB to ACE

There are 2 broader ways to migrate from IIB to ACE.

Method 1: One Application at a time

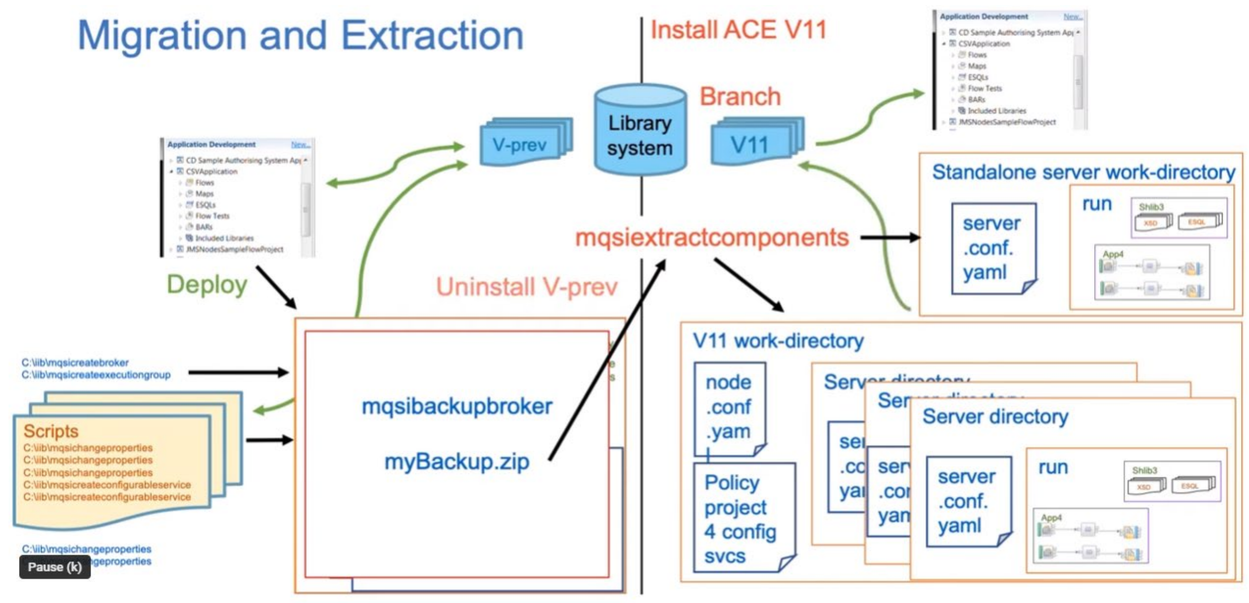
To do things one step at a time, take the IIB development resources from a branch in version control and then deploy the artifacts from the ACE toolkit into the ACE runtime.



Method 2: Big bang approach

This method involves taking a backup of your existing IIB V10 deployment using the **mqsibackupbroker** command that will generate a zip file, which can then either be used on the same machine or moved to a different machine where you have ACE installed. With this backup zip file you can run the **mqsiextractcomponents** command. This command takes the information from the zip file and generates the artifacts that are required by ACE for placement on its filesystem into a working directory. These artifacts can be placed under a standalone integration server working directory, or, alternatively, into server directories for the use of servers which are owned by an integration node.

Back to TOC



When the artefacts are migrated from IIB to ACE, configurations are migrated from IIB to ACE equivalents. One of the most important components is IIB configurable services. These configurable services control the dynamic connectivity to different systems, for example, FTP servers, email servers, and SAP system, to name a few. There are no more configurable services in ACE and these are replaced by equivalent policies in ACE. Another key aspect is configuration of server itself. IIB server configuration is managed in BrokerRegistry, but in ACE, server configuration is controlled using yaml file (server.conf.yaml).

Section 2 – Installing IBM App Connect in a Kubernetes environment (AWS EKS)

Let's explain how to install the latest version of the IBM ACE using operators on an AWS Kubernetes cluster and deploy an integration using it.

If you are using Red Hat OpenShift platform, it significantly reduces the effort of the deployment process as it pre-integrates a large number of commonly used capabilities and performs a number of common steps automatically. However, not all enterprises have OpenShift available to them, and for that reason, IBM App Connect is designed and supported to run on any major Kubernetes platform.

This section focuses on the steps required to perform IBM App Connect deployment on AWS EKS using operators. To keep this to a reasonable length, it is assumed that the reader has some familiarity with AWS and Kubernetes already.

Why operators?

The first technology used to simplify Kubernetes deployments was Helm. You can call Helm a package manager for deploying apps onto Kubernetes platform. Helm provides templates for all

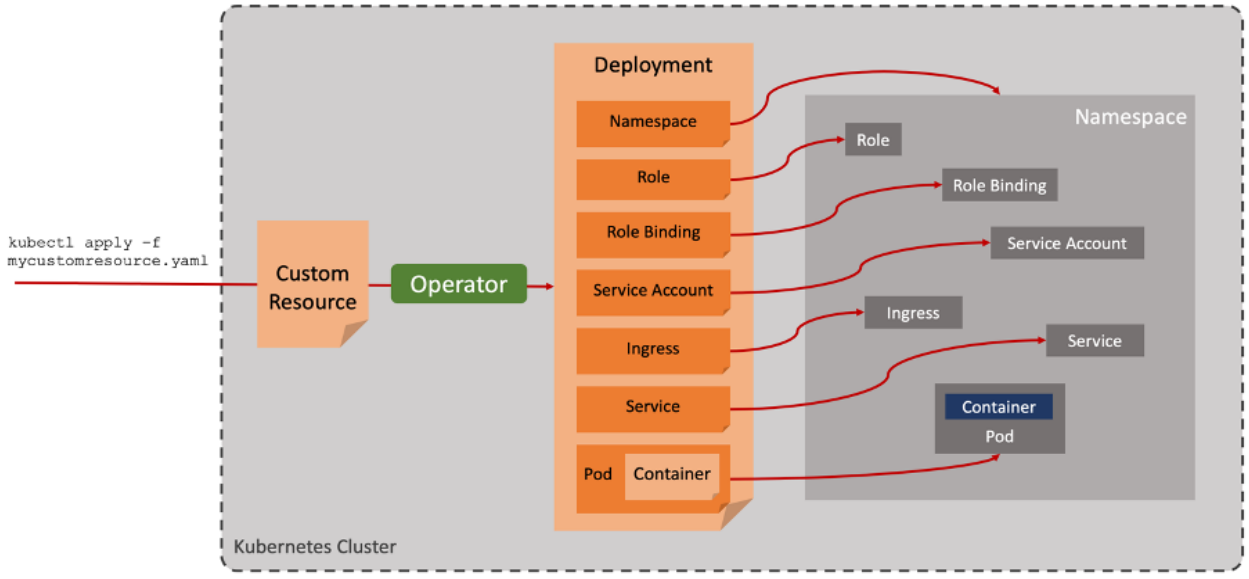
Back to TOC

the objects you need for the deployment. You could then simply provide a file containing values such as name of the application, container image it needed to be built from, replication policy and so on. This was the initial approach that was used for deployment, and you can of course still use Helm charts to deploy integration servers.

However, Helm only gets us so far. It's good for the initial deployment of something like an integration server, and it can achieve basic updates. But you will quickly learn about the shortcomings of Helm when you need to perform monitoring; continuous health checking of the environment; maintaining and upgrading of integrations and the associated runtimes once deployed; other lifecycle issues such as storage management; and more complex, multi-container installations.

It's good for the initial deployment of something like an integration server, and it can achieve basic updates, but what about setting up monitoring ans continuous health checking of the environment? How will it maintain and upgrade the integrations and the associated runtimes once deployed, and other lifecycle issues such as storage management? What about managing the more complex, multi-container installations?

This is where operators simplify things and overcome the shortcomings of Helm. This is in fact the approach used behind all the native Kubernetes objects themselves, but it is possible to extend the Kubernetes such that you can work with your own custom objects too, via the standard Kubernetes API. You provide to the API a file describing the state you would like your application to reach, and the operator works out how to turn that into required Kubernetes objects.



The operator performs all the necessary work to bring an application to life and keep it running. It can be made to do essentially anything required by the application – install it, upgrade it, monitor it and so on.

Before you install the IBM App Connect Operator, you must set up the environment with certificate management and Operator Lifecycle Manager (OLM) to manage the lifecycle of the operator.

Back to TOC

Section 3 – Here is the step-by-step process for installing IBM App Connect in a K8 environment using operator

Supported operating environment for Kubernetes - The minimum requirements for installation are as follows:

- The Kubernetes version of the cluster must be 1.23.x
- Only Linux 64-bit (x86-64) is supported
- Operator SDK 1.2.0 or later is required

Setting up your cluster

The first step is to create the EKS cluster in which you want to install IBM App Connect.

App Connect makes use of Kubernetes secrets. These secrets are stored in **etcd** and are not encrypted by default. Enable the encryption of etcd data for your cluster.

Install the following tools on your local workstation to manage the cluster, containers and other resources:

- **AWS CLI:** Download and install the command-line interface (CLI) tool for logging in to your Kubernetes environment
- **Kubernetes CLI (kubectl):** Download and install the Kubernetes CLI to run commands against your cluster
- **Operator SDK:** Download and install a supported version of Operator SDK, which is part of the open source Operator Framework that provides tools for building and managing Operators
- **Helm CLI:** Download and install the Helm CLI if you want to use Helm (rather than other available methods) to install an ingress controller. An ingress controller is required to expose the deployed App Connect instances to external

Configure AWS CLI

```
$ aws configure
AWS Access Key ID [***************J74P]:
AWS Secret Access Key [***************D12u]:
Default region name [us-east-1]:
Default output format [json]:
```

For more information refer to AWS CLI documentation.

Create/update your EKS cluster configuration

Create a kubeconfig file that stores credentials for Kubernetes cluster on EKS.

```
$ aws eks update-kubeconfig --name <cluster-name>
```

Install certificate manager

Any communication to components that needs to be secured will require certificates.

Back to TOC

The operator installation requires the Kubernetes certificate manager to generate and manage the TLS certificates that are essential for internal communication, and exposure of web user interfaces. This will also be required later by the integrations that are deployed if they expose for example HTTPS APIs.

"Cert-manager" is a Kubernetes add-on to automate the management and issuance of TLS certificates from various issuing sources. Cert-manager is included in OpenShift, but in EKS it has to be explicitly installed.

Install cert-manager using the following command:

```
kubectl apply -f https://github.com/cert-manager/cert-manager/releases/download/v1.8.0/cert-manager.yaml
```

You can verify the cert-manager installation using:

```
kubectl get pods --namespace cert-manager
```

We also need to perform a patch to the cert-manager to ensure any auto-generated secrets that store certificates are automatically removed when there are no longer any "owner references."

```
kubectl patch deployment \
  cert-manager \
  --namespace cert-manager  \
  --type='json' \
  -p='[{"op": "replace", "path": "/spec/template/spec/containers/0/args", "value": [
  "--v=2",
  "--cluster-resource-namespace=$(POD_NAMESPACE)",
  "--leader-election-namespace=kube-system",
  "--enable-certificate-owner-ref"
]}]'
```

Install the Operator Lifecycle Manager (OLM)

The Operator Lifecycle Manager (OLM) is a fundamental part of the Operator Framework. It is a Kubernetes service that looks after the operators (such as the IBM App Connect Operator). It makes operators discoverable, ensures that installed operators are kept up to date and manages their lifecycle.

Operator SDK is a pre-req for installing OLM. Ensure that this SDK is installed before proceeding with OLM installation.

Run the following command to install OLM:

```
operator-sdk olm install
```

Verify the OLM installation:

```
kubectl get crds
```

Back to TOC

Install the IBM App Connect Operator
Operator can be installed in either of these modes:

- **A cluster-wide installation into all namespaces:** In this mode IBM App Connect Operator is installed into the operator's namespace. The Operator can be installed only once in the cluster with this mode. This mode is the default option and is the recommended approach because it allows you to have a single point to control the Operator. In this mode, the Operator will be available to all namespaces on the cluster and can be used to deploy and manage App Connect instances or resources within any namespace.
- **Installation into a specific namespace on the cluster:** In this mode IBM App Connect Operator is installed into single namespace of your choice. Unlike cluster-wide installation, Operator can be installed multiple times, across multiple namespaces, in the same cluster and at different versions. Each Operator will be isolated to a single namespace and can be used to deploy and manage App Connect instances within that namespace only. Installing different versions of the Operator across namespaces is strongly discouraged, because it can result in conflicts with the CustomResourceDefinitions (CRDs) that are used to create App Connect resources.

Installation of the App Connect Operator itself involves three steps:

A. Create an OperatorGroup
B. Add App Connect to the operator catalog
C. Install the operator by creating a subscription

A. Create an OperatorGroup for the IBM App Connect Operator
The OLM runs with high levels of privilege within Kubernetes and can grant permissions to operators that it deploys. An OperatorGroup provides a mechanism for controlling what permissions are granted by OLM, and in which namespaces.

A single OperatorGroup must be created in each namespace that will contain operators. The namespace for our operator will be **ace-demo** and we need to create an OperatorGroup within that.

Create the app-connect namespace for our operator:

```
kubectl create namespace ace-demo
```

Create an OperatorGroup definition by creating a file named appconn-operator-group.yaml with the following content:

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: app-connect-operator-group
  namespace: ace-demo
```

Back to TOC

```
spec:
  targetNamespaces:
  -ace-demo
```

It is important that the namespace above matches with the namespace into which we'll deploy our integration.

Now create the OperatorGroup:

```
kubectl apply -f appconn-operator-group.yaml
```

B. Add IBM App Connect to the operator catalog

OLM doesn't yet know the IBM App Connect operator exists, or where to get it from. We need to create an entry in the OLM catalog by creating a CatalogSource object.

Create a file named appconn-catalog-source.yaml with the following content:

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: ibm-appconnect-catalog
  namespace: olm
spec:
  displayName: "IBM App Connect Operator Catalog k8S"
  publisher: IBM
  sourceType: grpc
  image: icr.io/cpopen/appconnect-operator-catalog-k8s
  updateStrategy:
    registryPoll:
      interval: 45m
```

Note that the namespace for the CatalogSource is always "olm."

Now add it to the OLM catalog:

```
kubectl apply -f appconn-catalog-source.yaml -n olm
```

You can verify the catalog entry using:

```
kubectl get CatalogSources ibm-appconnect-catalog -n olm
```

C. Install the operator using a "subscription"

Kubernetes works "declaratively," meaning you "declare" what you want, then it works out how, and when to do it.

Installing an operator follows the same principle. You declare to the OLM that you would like it to "subscribe" to a particular operator in the catalog. It then works on downloading the current version of it and instantiating it. There are many advantages to this declarative approach. For example, we can declare that we want to follow a particular "channel" of updates to the operator, we might want every update, or just stable releases and so on. We can also declare whether or not it should update the operator automatically or not.

Back to TOC

To create a subscription to the IBM App Connect Operator, create a file named appconn-sub.yaml with the following contents:

As you already know, you can install the Operator into a single namespace or cluster wide.

- To install into a single namespace, replace *namespaceName* with the namespace that you created earlier (ace-demo)
- To install cluster wide, replace *namespaceName* with operators

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: ibm-appconnect
  namespace: namespaceName
spec:
  channel: v9.2
  name: ibm-appconnect
  source: ibm-appconnect-catalog
  sourceNamespace: olm
```

Notice that we have subscribed to "v9.2" channel. This means we're subscribing to a specific version of the operator, which was correct at the time of this writing.

To install the operator, add the subscription:

```
kubectl apply -f appconn-sub.yaml
```

To verify that the operator has installed:

```
kubectl get csv
```

Note:

- Deploy License Service to track license consumption of IBM App Connect. For compliance with the licensing requirements, ensure that License Service is deployed on your cluster to monitor and measure license usage of App Connect.
- Obtain an entitlement key and create your IBM App Connect authoring and runtime environments, and other resources. An entitlement key (supplied as a Kubernetes pull secret) will be required to pull the software images from the IBM Entitled Registry

Back to TOC

Section 4 – Creating an Ingress for external access to IBM App Connect instances

At this point, the cluster is unable to receive requests from the outside world. To expose the deployed instances of App Connect Dashboard, App Connect Designer, integration servers, integration runtimes, and switch servers to external traffic, you need to define ingress rules that route paths to the internal services and ports. Ingress is a Kubernetes service that exposes the services in the cluster to the public or private network.

Points to remember:

- An ingress controller must be installed and running in your Kubernetes cluster.
- If you have deployed a switch server, you need to customize the ingress controller to enable SSL pass-through for switch server interactions. A switch server enables you to run hybrid integrations that interact with callable flows or with applications in a private network, so this customization provides secure connectivity between the switch server and an external integration server that acts as a switch client.
- For each App Connect Dashboard, App Connect Designer, integration server, integration runtime, or switch server instance, you need to also create an ingress resource. This ingress resource contains rules that define an externally reachable URL that you can use to access the running service in the cluster.

Creating an ingress route for an App Connect Dashboard instance

Create a file named appconn-ingress-dashboard.yaml with the following contents:

```
kind: Ingress
apiVersion: networking.k8s.io/v1
metadata:
  name: <dashboardIngressName>
  namespace: <namespaceName>
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/backend-protocol: HTTPS
spec:
  tls:
    - hosts:
      - <dashboardHostPrefix>.mydomain.com
  rules:
    - host: <dashboardHostPrefix>.mydomain.com
      http:
        paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: <dashboardCRname>-dash
              port:
                number: 8300
```

Run the following command to create the ingress:

```
kubectl apply -f appconn-ingress-dashboard.yaml
```

Back to TOC

Creating an ingress route for a switch server
To expose a switch server to external traffic, you must create an ingress route immediately after you create the switch server. This is because during its initialization, the switch server will need to provide a TLS host name (defined in an ingress resource) in order to request a certificate for this host. To prevent certificate-related errors from the ingress controller, the host name in the generated certificate and the TLS host name that is defined in your ingress resource must match.

Create a file named appconn-ingress-switch.yaml with the following contents:

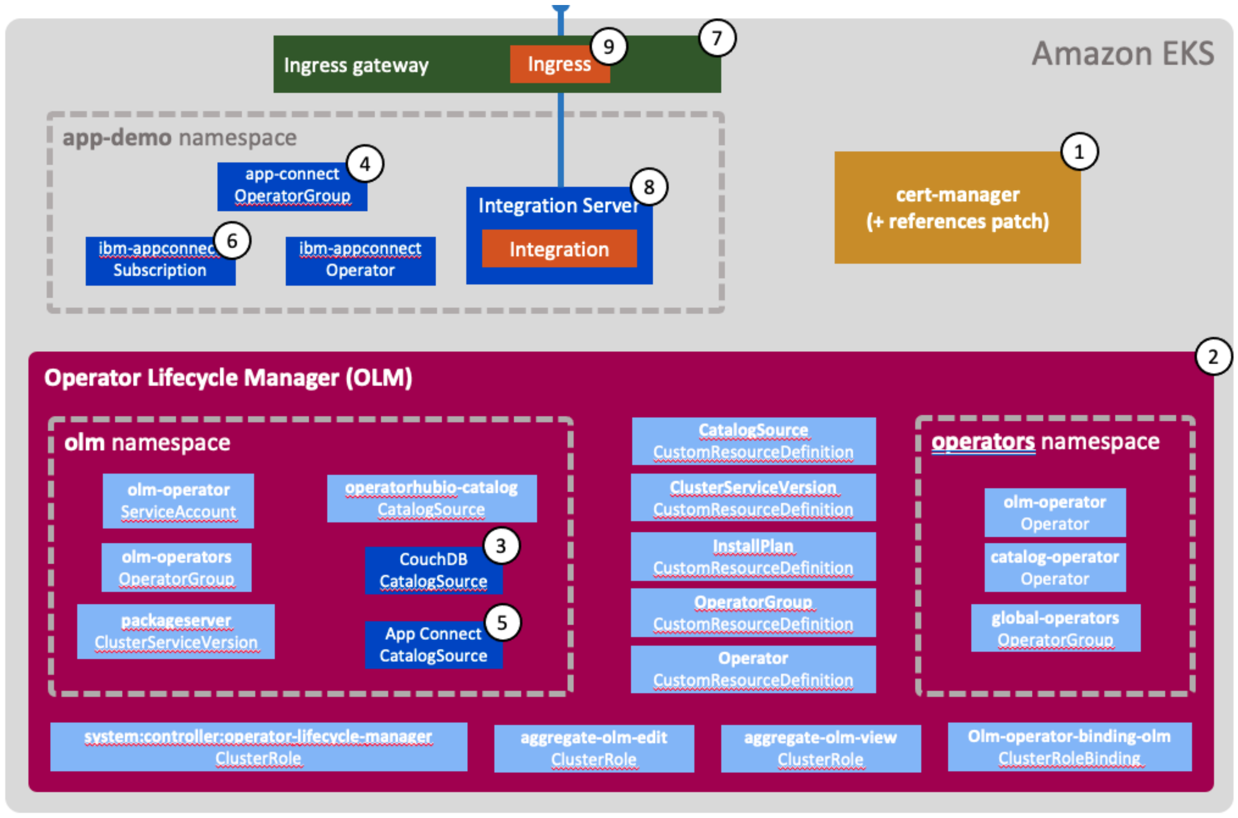
```
kind: Ingress
apiVersion: networking.k8s.io/v1
metadata:
  name: <switchServerIngressName>
  namespace: <namespaceName>
  labels:
    appconnect.ibm.com/switch: <switchServerCRName>
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/ssl-passthrough: "true"
spec:
  tls:
    - hosts:
      - <switchServerHostPrefix>.mydomain.com
  rules:
    - host: <switchServerHostPrefix>.mydomain.com
      http:
        paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: <switchServerCRName>-switch
              port:
                number: 4443
```

Run the following command to create the ingress:

```
kubectl apply -f appconn-ingress-switch.yaml
```

Back to TOC

The following diagram provides a summary of the activities and components we have deployed in the EKS cluster.


Amazon EKS
Ingress gateway
Ingress
9
7
app-demo namespace
app-connect OperatorGroup
4
Integration Server
8
Integration
ibm-appconnect Subscription
6
ibm-appconnect Operator
cert-manager (+ references patch)
1
Operator Lifecycle Manager (OLM)
2
olm namespace
olm-operator ServiceAccount
operatorhubio-catalog CatalogSource
olm-operators OperatorGroup
CouchDB CatalogSource
3
packageserver ClusterServiceVersion
App Connect CatalogSource
5
CatalogSource CustomResourceDefinition
ClusterServiceVersion CustomResourceDefinition
InstallPlan CustomResourceDefinition
OperatorGroup CustomResourceDefinition
Operator CustomResourceDefinition
operators namespace
olm-operator Operator
catalog-operator Operator
global-operators OperatorGroup
system:controller:operator-lifecycle-manager ClusterRole
aggregate-olm-edit ClusterRole
aggregate-olm-view ClusterRole
Olm-operator-binding-olm ClusterRoleBinding

This completes the installation of required App Connect components and we can move on to deploying an actual integration.

Section 5 – Deploy your integration

Create the barauth configuration object

Create a text file named github-barauth.yaml with the below contents:

```
apiVersion: appconnect.ibm.com/v1beta1
kind: Configuration
metadata:
  name: github-barauth
  namespace: ace-demo
spec:
  data:
eyJhdXRoVHlwZSI6IkJBU0lDX0FVVEgiLCJjcmVkZW50aWFscyI6eyJ1c2VybmFtZSI6IiIsInBhc3N3b3JkIj
oiIn19Cgo=
  description: authentication for github
  type: barauth
```

Back to TOC

spec.data section is the base64 encoded string of authentication credentials since we are pulling the bar file from github.

Apply the barauth configuration definition to Kubernetes:

```
kubectl apply -f github-barauth.yaml
```

Create an Integration Server for the simple integration
Deploy an App Connect integration server certified container with a github link to the BAR file by creating a file named http-echo-service.yaml with the following contents:

```
apiVersion: appconnect.ibm.com/v1beta1
kind: IntegrationServer
metadata:
  name: http-echo-service
  namespace: ace-demo
spec:
  adminServerSecure: false
  barURL: >-
    https://github.com/amarIBM/hello-world/raw/master/HttpEchoApp.bar
  configurations:
    - github-barauth
  createDashboardUsers: true
  designerFlowsOperationMode: disabled
  enableMetrics: true
  license:
    accept: true
    license: L-KSBM-C37J2R
    use: AppConnectEnterpriseProduction
  pod:
    containers:
      runtime:
        resources:
          limits:
            cpu: 300m
            memory: 350Mi
          requests:
            cpu: 300m
            memory: 300Mi
  replicas: 1
  router:
    timeout: 120s
  service:
    endpointType: http
  version: '12.0’
```

Create the integration server:

```
kubectl apply -f http-echo-service.yaml
```

Back to TOC

Create ingress object

Our simple integration service receives requests over HTTP, and these may come from the public internet. To enable a container to receive requests from outside of the EKS cluster, you will need to create an ingress route.

Create a definition file named is-ingress.yaml with the following contents, replace **<external-hostname>** with the correct value:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: http-echo-service-ingress
  namespace: ace-demo
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - host: <external-hostname>
    http:
      paths:
      - path: /
        pathType: ImplementationSpecific
        backend:
          service:
             name: http-echo-service-is
             port:
               number: 7800
```

Deploy the ingress object:

```
kubectl apply -f is-ingress.yaml
```

Test the integration

Verify the status of Integration Server pod and ensure that it is running:

```
kubectl get pods
```

Invoke the service using curl command using the URL you configured in the ingress definition:

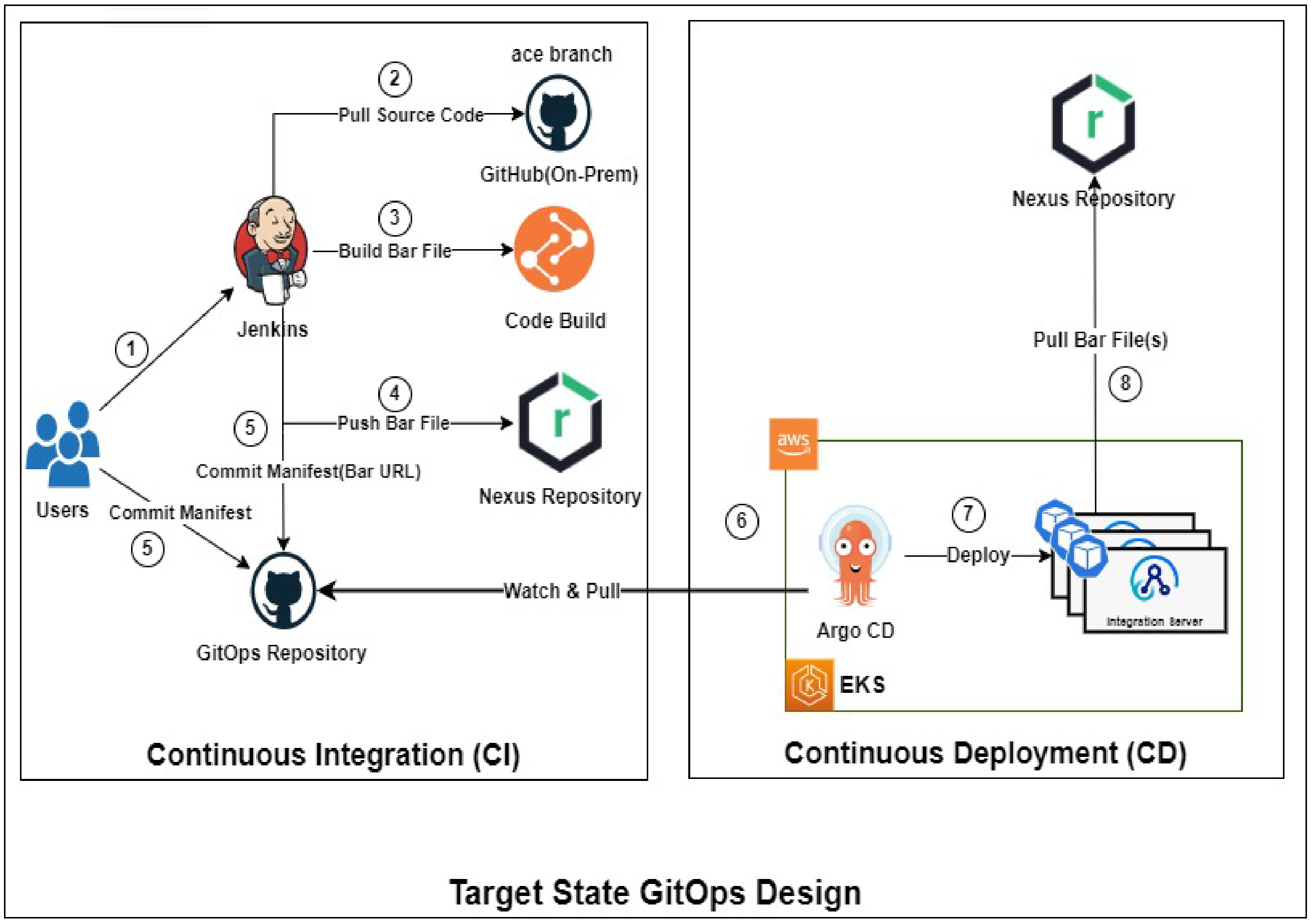
```
curl -X POST http://http-echo-service-http-ace-demo. yyy.yyy.yyy.yyy.yourdomain.com/Echo
```

You should receive a response, letting you know that your request made it into the container and returned back.

Example use case of deploying integrations using pipeline

Coming to deployment of integration, you can follow different methods depending on devops process adopted in your enterprise. Below is one of the ways you can automate the deployment of App Connect workload. This method uses the combination of Jenkins and Argo CD along with nexus repo. You can visualize the process from the diagram below:

Back to TOC



1. Developers will be using Jenkins as a Continue Integration (CI) to build the code after committing changes into the GiHub code repository

Jenkins:

2. **Pull** the code from GitHub repository
3. **Build** the bar file(s) using **ANT** script
4. **Push** the bar file(s) into **Nexus** Repository with version
5. The Workforce/Administrator/Operations team will **commit the manifest/configuration resources** such as Custom Resource Definition (CRD), server configuration, etc., into **GitOps Repository** with the version

Argo CD:

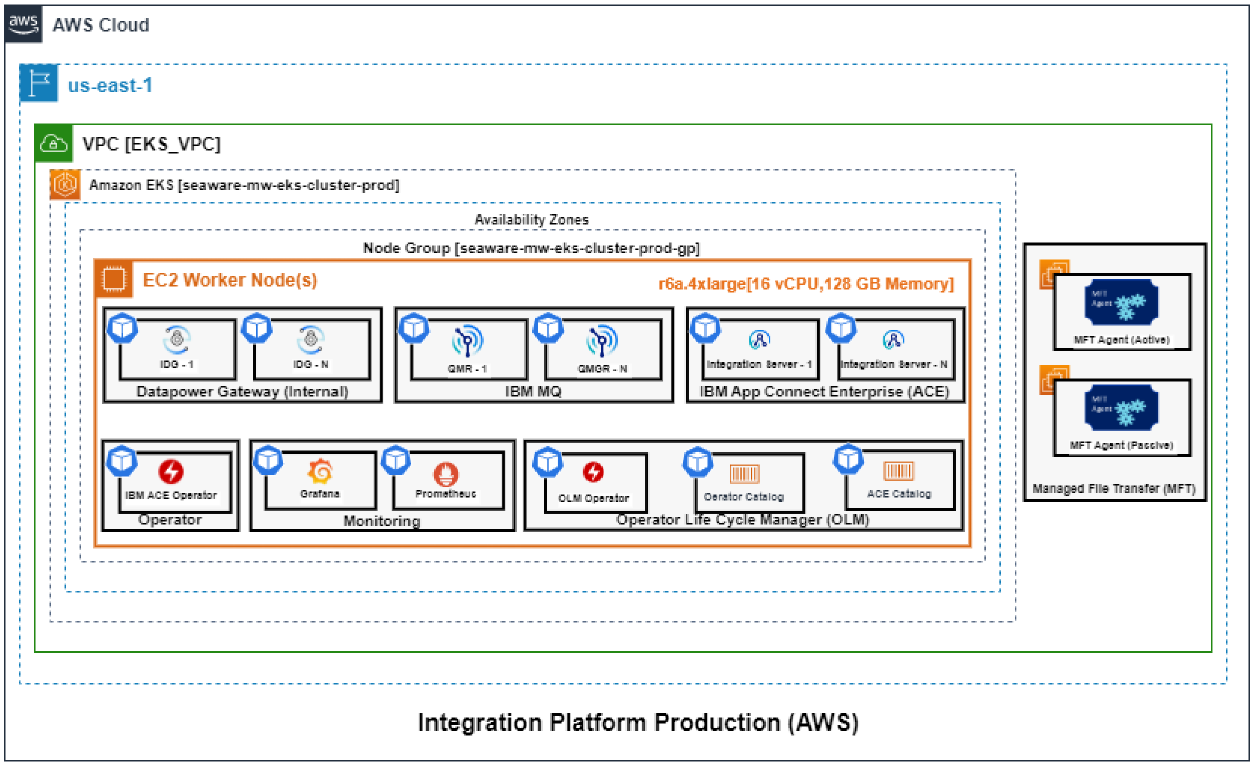
6. Continuously sync and **pull** the changes.
7. **Deploy** the changes into AWS EKS cluster.
8. **Pull** the bar file(s) from Nexus Repository during Kubernetes Pod Container Initialization

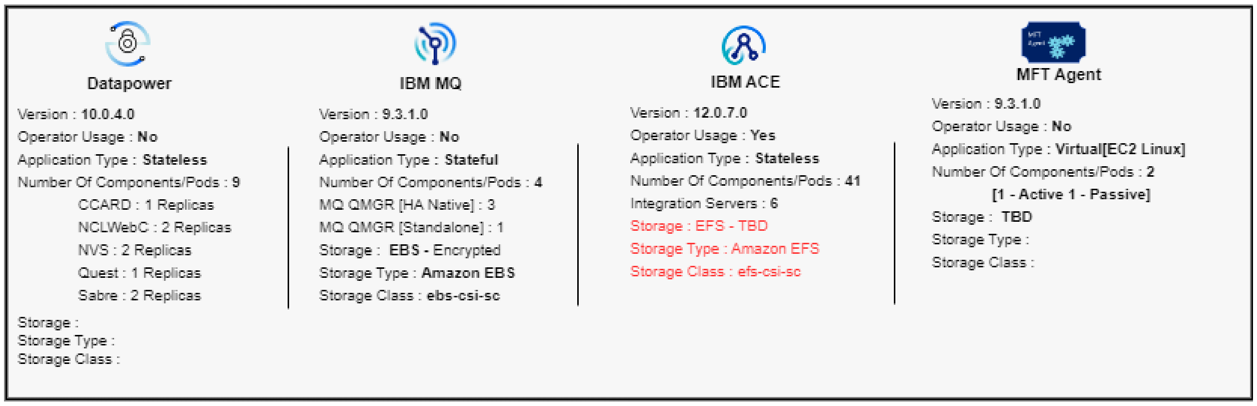
Back to TOC

Section 6 - Conclusion

That's it – you have deployed a simple flow from an IBM Integration Bus environment into IBM App Connect container on Amazon EKS.

Here is the holistic view of IBM integration stack deployed onto AWS EKS platform and how they inter-operate.


AWS Cloud
us-east-1
VPC [EKS_VPC]
Amazon EKS [seaware-mw-eks-cluster-prod]
Availability Zones
Node Group [seaware-mw-eks-cluster-prod-gp]
EC2 Worker Node(s)
r6a.4xlarge[16 vCPU,128 GB Memory]
IDG - 1
IDG - N
Datapower Gateway (Internal)
QMR - 1
QMGR - N
IBM MQ
Integration Server - 1
Integration Server - N
IBM App Connect Enterprise (ACE)
IBM ACE Operator
Operator
Grafana
Prometheus
Monitoring
OLM Operator
Oerator Catalog
ACE Catalog
Operator Life Cycle Manager (OLM)
MFT Agent (Active)
MFT Agent (Passive)
Managed File Transfer (MFT)
Integration Platform Production (AWS)



Datapower

Version : **10.0.4.0**
Operator Usage : **No**
Application Type : **Stateless**
Number Of Components/Pods : 9
CCARD : 1 Replicas
NCLWebC : 2 Replicas
NVS : 2 Replicas
Quest : 1 Replicas
Sabre : 2 Replicas
Storage :
Storage Type :
Storage Class :

IBM MQ

Version : **9.3.1.0**
Operator Usage : **No**
Application Type : **Stateful**
Number Of Components/Pods : **4**
MQ QMGR [HA Native] : 3
MQ QMGR [Standalone] : 1
Storage : **EBS** - Encrypted
Storage Type : **Amazon EBS**
Storage Class : **ebs-csi-sc**

IBM ACE

Version : **12.0.7.0**
Operator Usage : **Yes**
Application Type : **Stateless**
Number Of Components/Pods : **41**
Integration Servers : 6
Storage : EFS - TBD
Storage Type : Amazon EFS
Storage Class : efs-csi-sc

MFT Agent

Version : **9.3.1.0**
Operator Usage : **No**
Application Type : **Virtual[EC2 Linux]**
Number Of Components/Pods : **2**
[1 - Active 1 - Passive]
Storage : **TBD**
Storage Type :
Storage Class :

The steps would largely be the same for any non-OpenShift Kubernetes environment such as the Azure Kubernetes service (AKS), or indeed a self-managed non-OpenShift Kubernetes cluster.

Back to TOC

About the author

Raveendra Nanjundappa, Prolifics Integration Solution Architect, has more than 10 years of experience in the IT industry executing various roles – Solution/Integration Architect, Technology Specialist, Senior Consultant, and Technical/Project Lead with specialization in middleware and Enterprise integration/SOA. His expertise is in architecting, designing and building middleware messaging and SOA/Integration solutions using IBM WebSphere product suite. This includes Websphere MQ, IBM Integration Bus/ Websphere Message Broker, IBM Datapower, Websphere Service Registry and Repository (WSRR), Websphere Extreme Scale (WXS), and more. Nanjundappa has experience of working in various industries, such as banking, manufacturing, retail and health care.