Cloud-native integration deployment

Introduction

In this demonstration we will automate deploying a complex integration solution using a pipeline. This enables faster, more frequent delivery of changes into production and improves deployment confidence. We will also see how container-based platforms enable operational consistency and automation, simplifying administration of an environment.

Our scenario features an insurance quote aggregator gathering insurance quotes from multiple companies and providing a combined list to insurance sellers via an API. This involves multiple integration styles, including application integration, API management, and messaging. Our goal is automated deployment and operations.

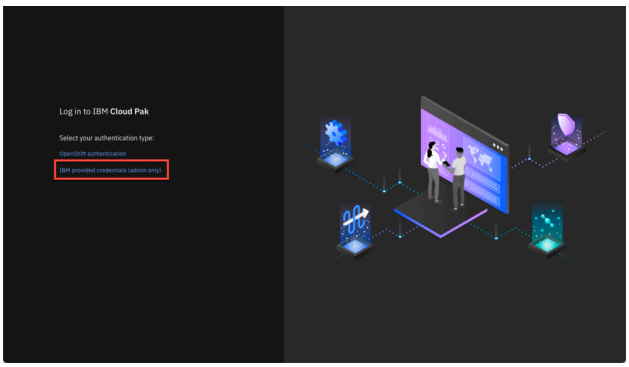
We will begin with a simple deployment of a single integration that retrieves a quote from an insurer. Later, we will explore a more complex solution, with multiple integration capabilities that we want to deploy together - including integration flows, queues, and managed APIs.

1 - Accessing the environment

1.1 - Log into Cloud Pak for Integration

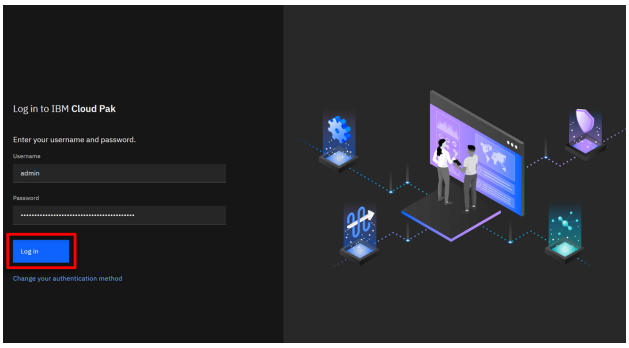
Narration
Let's see IBM Cloud Pak for Integration in action.

Action 1.1.1
- Open Cloud Pak for Integration using the URL saved from demo preparation step 2.9 and click **IBM provided credentials (admin only)**.



Narration
Let's log into Cloud Pak for Integration on IBM Cloud.

Action 1.1.2
- **Log in** using the **Username "admin"** and the 32-character **Password** that was created in demo preparation step 2.5.

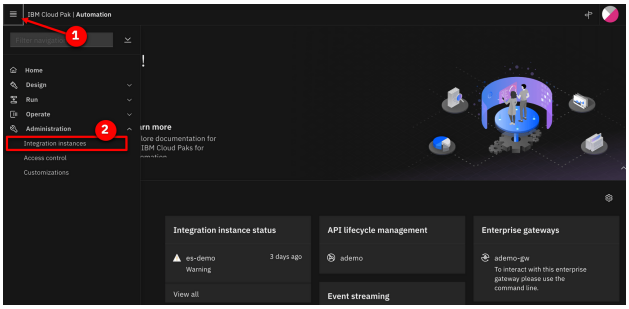


1.2 - View the Cloud Pak for Integration home screen

Narration
This is the IBM Cloud Pak for Integration home screen, which shows all the capabilities of the pak in one place. Specialized integration capabilities for API management, application integration, messaging, and more, are built on top of powerful automation services.

Let's see the integration capabilities available.

Action 1.2.1
- From the **Home Page**, open the top left menu (1) and then click **Integration instances** (2) under **Administration**.

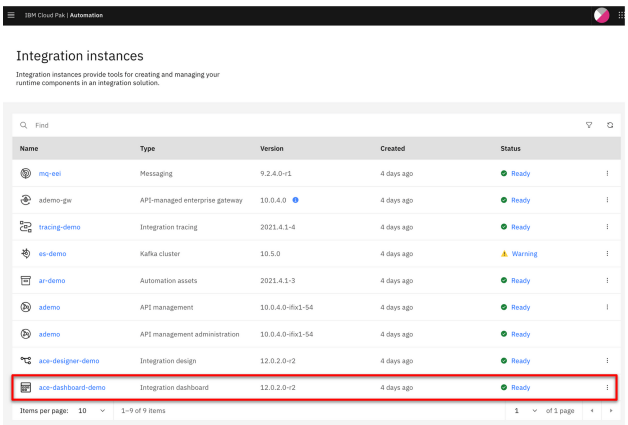


1.3 - Access integration capabilities

Narration
You are able to use a a single-user interface to access all the integration capabilities your team needs, including API management, application integration, enterprise messaging, events, and high-speed transfer. To automate customer interactions in this demo, we will use App Connect for application integration, API Connect for API management, and the Message Queue for Enterprise Messaging.

Let's open our App Connect dashboard.

Action 1.3.1
- Show the **Integration instances** page, then click the **ace-dashboard-demo** in the **dashboard** row.

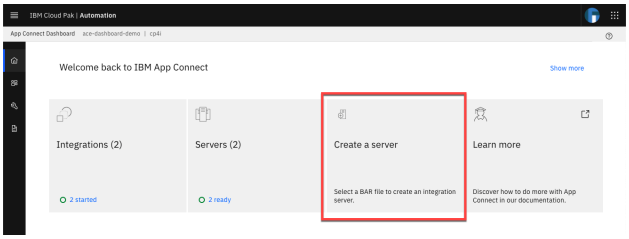


2 - Deploying your integration

2.1 - Create an integration server

Narration
We'll create an integration server for our new integration deployment. Each integration server is deployed in a separate container.

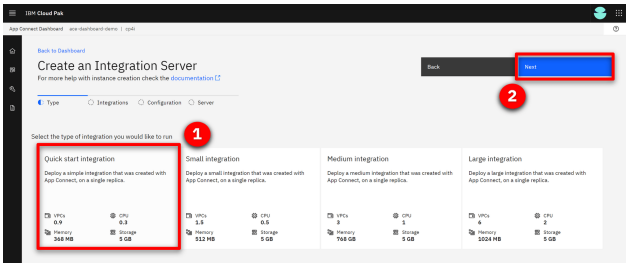
Action 2.1.1
- Click **Create a server.**



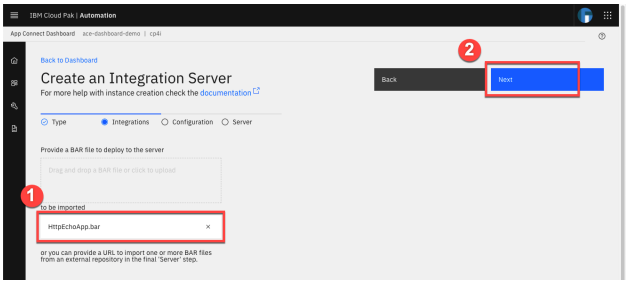
2.2 - Import the BAR file

Narration
We're now going to deploy an integration that we've already created in the App Connect toolkit. We simply drag and drop it into the console straight from the file system. If we've loaded the BAR (broker archive) file before, it will be available from an internal asset repository.

Action 2.2.1
- Select **Quick start integration** (1). Click **Next** (2).



Action 2.2.2
- Upload the **HTTPEchoApp.bar** BAR file (1) that you downloaded during demo preparation. Click **Next** (2).



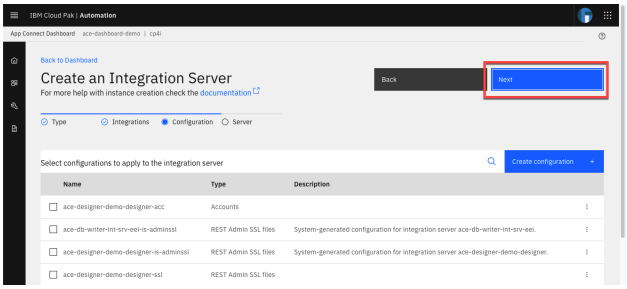
2.3 - Configure your integration server

Narration
We are then asked if we would like to apply any specific configuration information to this particular deployment, such as user credentials or certificates that are required to integrate to a particular backend system.

This might include credentials to the downstream insurance quotation engine. These are held within Kubernetes in standard mechanisms known as ConfigMaps and Secrets, with visibility limited to those with correct permissions to view, upload and change credentials.

This is particularly important in integration scenarios - such as our insurance quote aggregator example - which have credentials enabling access to multiple third-party insurers. Those systems may provide access to sensitive personal data. The insurers may be charging the insurance quote aggregator for access to their API, or conversely, providing commission on quotes. Misuse of the credentials could have all manner of undesired effects.

Action 2.3.1
- Click **Next**.



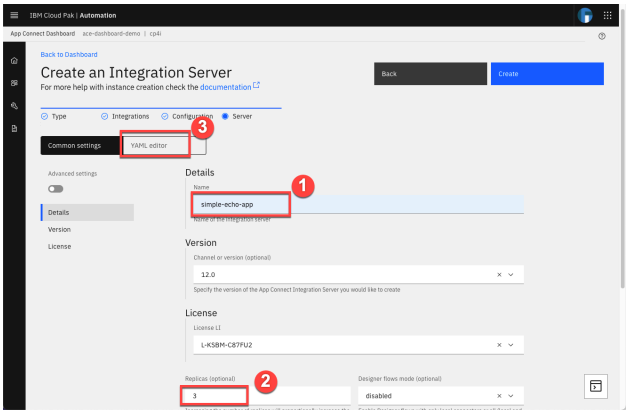
Narration
We need to fill in a form to populate a file used during deployment. The file has the details about how your integration will be deployed and operated. We can decide the version of the runtime that we want the integration to run against, and that is specific to this particular integration. Another integration might be running against a different version that it was tested against.

This is also where we decide how many replicas of the integration container we want. In a traditional installation, all integrations would inherit the same characteristics of the high availability pair of the centralised servers. In our scenario, our customers might be particularly sensitive to outages, so we might decide to minimize the availability impact of any runtime failures by increasing the number of container replicas to 3, or 5, or 7.

We could also change the amount of memory assigned to the container if we knew the data model for the insurance quotation was particularly large and therefore memory intensive.

Action 2.3.2

- Name the integration server **simple-echo-app** (1). Change the **Replicas** number to **3** (2). Open the **YAML editor** (3).

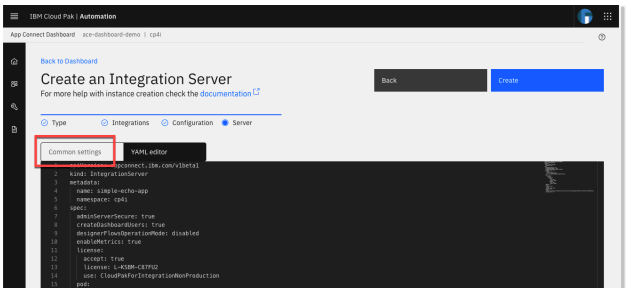


Narration

This form is just a graphical way of editing the YAML formatted deployment properties file. You can see it's updating that text file, and setting the number of replicas to "3". This file is known as the "custom resource definition" for the integration. It's used to instantiate this integration using command line tools, or by calling one of the OpenShift APIs, or from a pipeline - as we will do later in this demonstration.

Action 2.3.3

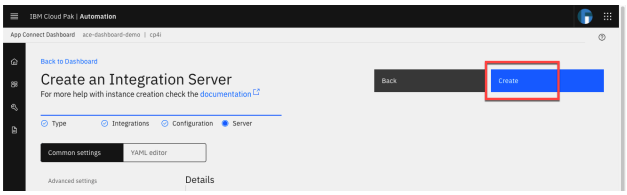
- Open the **Common settings** tab.



2.4 - Explore server creation

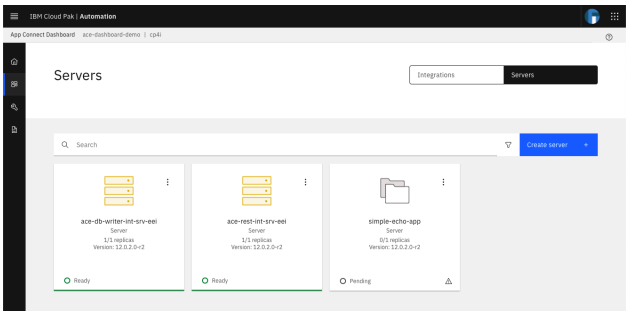
Narration
Let’s create that server.

Action 2.4.1
- Click **Create**.



Narration
We can see a new server appear, but it hasn’t started up yet. Behind the scenes, Kubernetes has received all the instructions it needs to start up – what image to download, how much CPU and memory to provide, and other parameters. It’s also been asked to create three replicas of the server and load balance between them.

Action 2.4.2
- Explore the server creation dashboard.

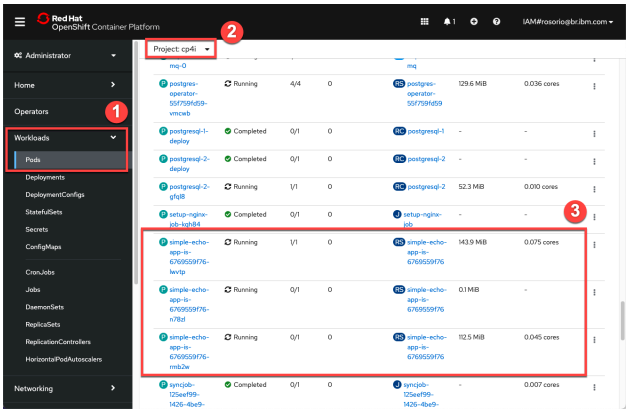


Narration
Let’s take a quick look at what Kubernetes is doing based on our instructions. There is a set of three pods starting up, which house the containers that integrate with our insurer. Those could have been created from a normal Kubernetes command line, using the same custom resource definition file, or (as we’ll see later,) by a pipeline that calls the Kubernetes APIs.

It’s easy to transition from a manual deployment to creating a scripted, automated deployment using this interface.

Action 2.4.3

- Open the OpenShift Web Console by clicking your cluster URL and then clicking OpenShift web console. On the left menu, select **Workloads**, then **Pods** (1). Select **cp4i** project (2). Show the simple-echo-app pods creation (3).

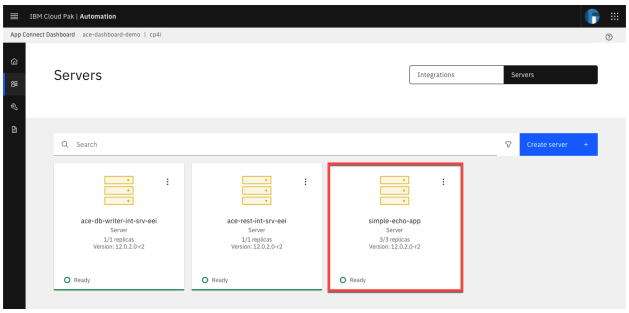


Narration

The App Connect Dashboard displays the integration up and running with three replicas. Kubernetes manages high availability implicitly and will ensure there are always three. If one of the replicas fail, Kubernetes will reinstate a new one in its place.

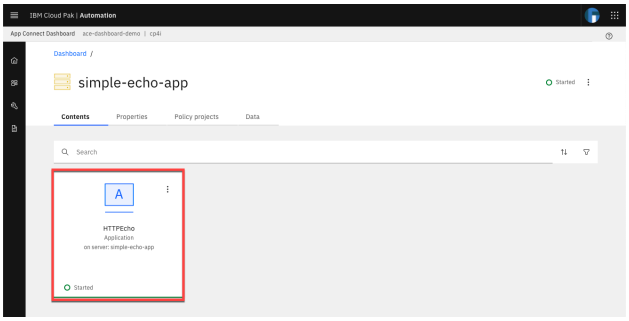
Action 2.4.4

- Return to the App Connect Dashboard page. Click **simple-echo-app server**.



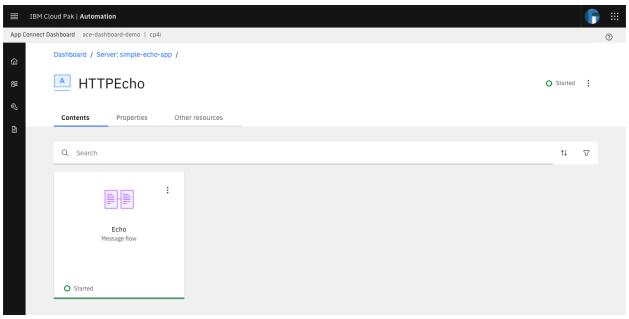
Action 2.4.5

- Open the **HTTPEcho** Application.



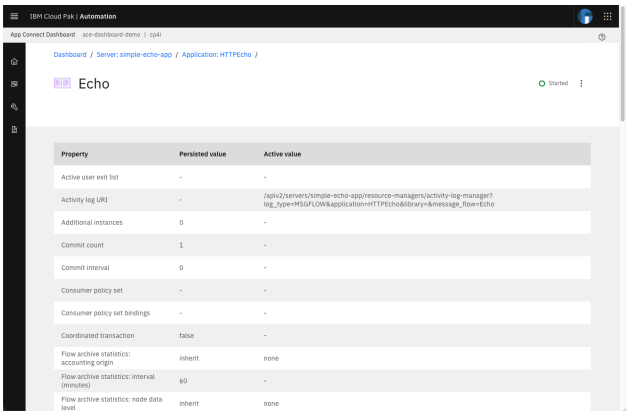
Action 2.4.6

- Open the **Echo** Message flow.



Action 2.4.7

- Return Explore the **Echo properties** page.



3 - Exploring the pipeline

3.1 - Explore the pipeline

Narration
The insurance quote aggregator now wants quotations from three different companies. This will require integration capabilities such as integration flows, queues, and managed APIs. Here are all the requirements to consider:

- Each insurance company has its own way of exposing their quotation system, so we have to build integration flows in App Connect to help us request those quotes and pull back the answers in a form we can use.

- The API needs to be responsive, handling as many quotations as possible, as quickly as possible.

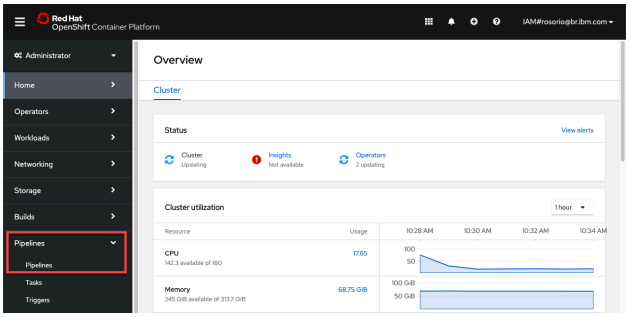
- These integrations and queuing capabilities need to scale independently, but deploy as a single solution. Deploying in a fine-grained way using containers means independent updating and scaling, things independently. They can deploy the complete solution in a consistent way through one CI/CD pipeline.

- The insurance companies want to make integration data available via an API so insurance partners can build this into their systems. If they can expose those APIs through an API management system, it will be easier for partners to onboard themselves to use the APIs, and potentially monetize them.

We're going to use the Kubernetes native Tekton pipeline capability that comes as part of OpenShift to deploy our integration solution. Here on my machine I forked and cloned a github project with the pipeline. Let's execute a script to create the pipeline in my OpenShift environment.

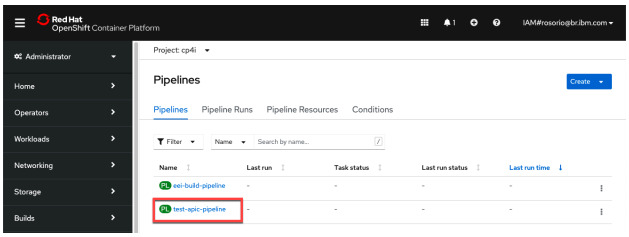
Now, let's open the OpenShift Console to check the pipeline that we created in the cp4i project.

Action 3.1.1
- In the OpenShift Web Console, open **Pipelines**, then **Pipelines**.



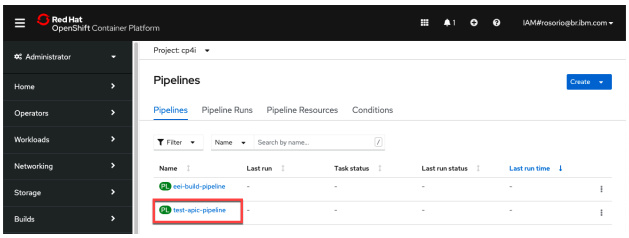
Action 3.1.2

- Filter by **cp4i project**.



Action 3.1.3

- Click to open **test-apic-pipeline**.

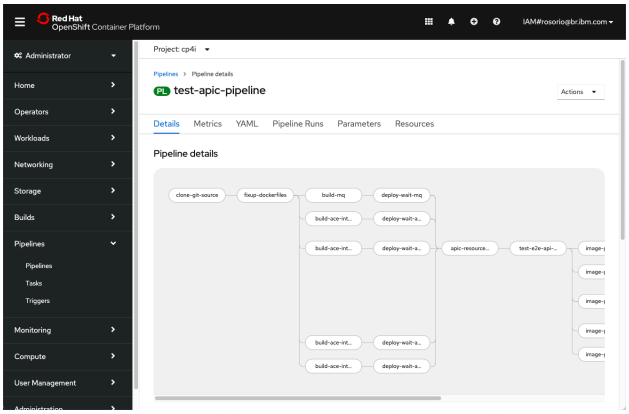


Narration

The details on this interface show that we specified a pipeline with multiple tasks that build multiple integration images and a queue manager, and also configure some API exposure. The pipeline then deploys these to a test environment, and upon successful completion of tests, deploys into the main environment.

Action 3.1.4

- Explore **test-apic-pipeline**.



4 - Initiating the pipeline from Git

4.1 - Configure the web hook

Narration
Source code is stored in a source code repository – in this case Git. We're going to set up a webhook on Git which will trigger our Tekton pipeline whenever we make a commit to the code.

Note that it is not just the integration code that could trigger a build. It could also be a change to an MQ configuration, or a change to the definition of an API.

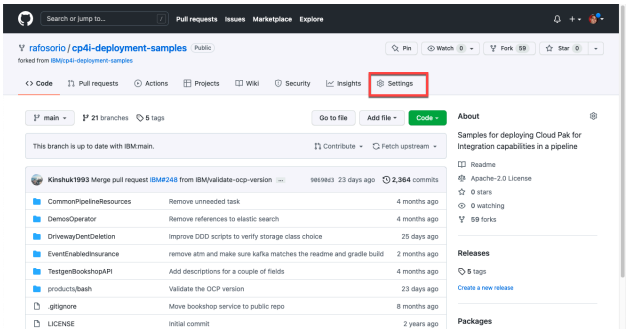
Furthermore, there are other things that we could, indeed should, store in Git. The custom resource definitions define the environment, so on some level they are "infrastructure as code". A change to those could trigger a pipeline too. This is the starting point for what is called GitOps, where operators never actually connect directly to the platform, but instead make configuration changes in the implicitly audited source code repository, following the exact same process as developers.

Our insurance quote aggregator now has even greater confidence in the consistency of what is in production as the code repository contains absolutely everything required to re-create it. If something happens in production, the code repository shows exactly what changes were recently made, whether infrastructure of integration code. Our aggregator could also easily build an exact replica of the whole solution to safely diagnose the problem.

Perhaps one of the most attractive features of this approach is that there is a complete configuration to roll back to a previous version. Imagine how much more comfortable you would be deploying a new business feature, if you knew you could get back to your previous state, rapidly, and with precision.

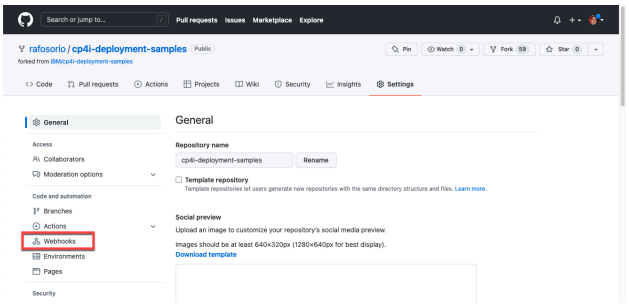
Here in the webhook page, we just need to input our webhook URL and define the content type as JSON. Our webhook will "fire" immediately we create it.

Action 4.1.1
- Open the GitHub project you forked during Demo preparation and click **Settings**.



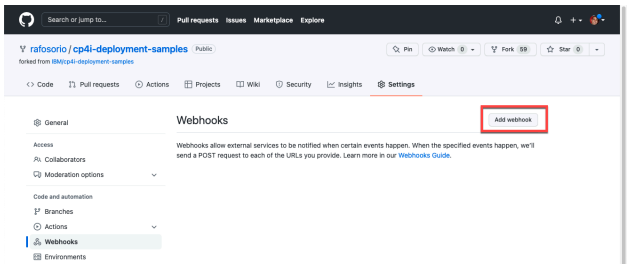
Action 4.1.2

- Click **Webhooks**.



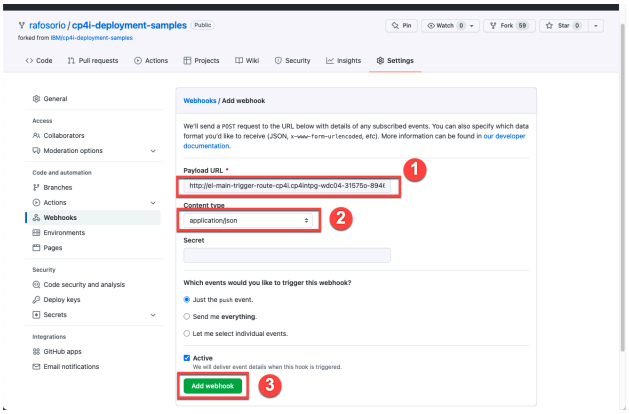
Action 4.1.3

- Click **Add webhook**.



Action 4.1.4

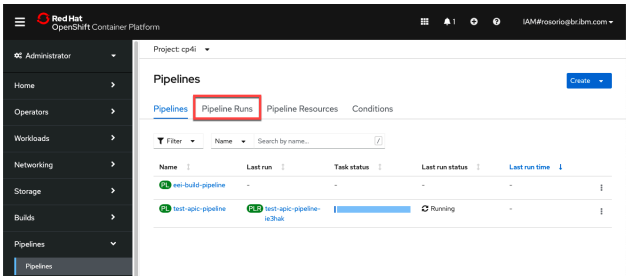
- Use the **Webhook route** URL from demo preparation step 8.3 in Payload URL (1). Change the content type to **application/json** (2). Click **Add webhook** (3).



4.2 - Check the pipeline run

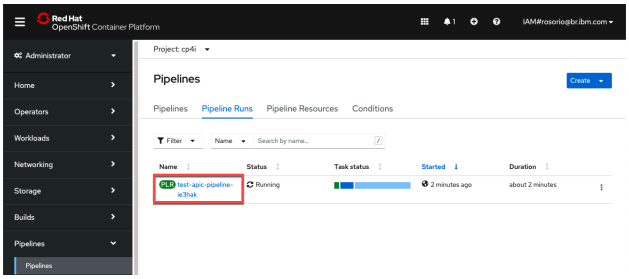
Narration
Let’s check our pipeline. Tekton is well suited to deploy cloud native solutions, and is itself a cloud native application. It runs on the Kubernetes platform in containers, and directly leverages the rapid deployment and scalability of containers to run pipelines.

Action 4.2.1
- In the OpenShift web console, click **Pipelines**, then the **Pipeline Runs** tab.



Narration
We can see our pipeline has been started, and we have a high-level view of its progress.

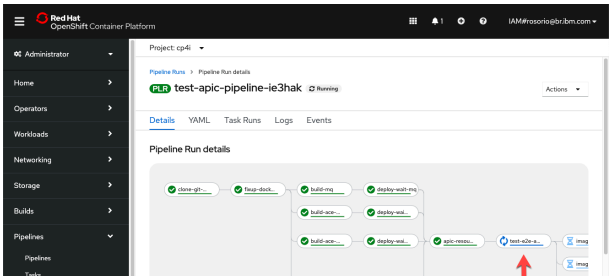
Action 4.2.2
- Open the pipeline diagram.



Narration
You can see a nice visualization of the pipeline doing its work all the way through to the testing tasks in the diagram.

The pipeline diagram shows our progress through the build, test, and then deployment to two different environments. We can see it has been initiated, and the builds of the various images are taking place. The arrow indicates the current build progress.

Action 4.2.3
- Display the pipeline diagram page.



5 - Confirming the deployment

5.1 - Check the App Connect deployment

Narration
Kubernetes is building a brand new topology on the fly. It is unique to this integration solution, ensuring queues, integrations and managed APIs are all able to connect to one another. Furthermore, it is then creating replicas for availability and performance, and implementing the load balancing across them.

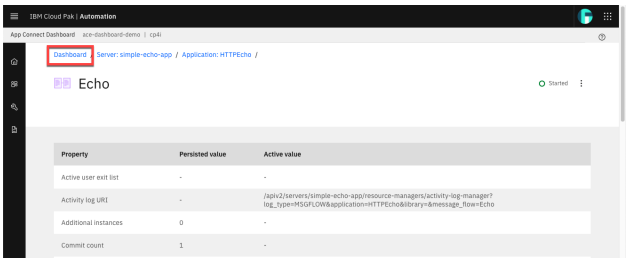
For our insurance quote aggregator this means that this integration solution is completely independent of all their other integrations. They can re-build the whole thing at will, scale it up or down, or refresh the underlying runtimes with new fix paks with no fear that they might disrupt some other part of their business.

Recall that part of the pipeline introduced a set of tests that had to be passed before the solution could be deployed to a second environment. Let's consider what those tests might be doing.

For our integration brokerage scenario, for example, we can imagine just how complex the testing could be. We might be setting up stubs to make calls against in the background. We might be bringing performance test capabilities to push load into the system. Then we need to tear all of that down before we then go onto the next step of finally evaluating the test results and then deciding whether to push into the target environment. This is one of the real benefits of the elastic nature of container environments. Test environments can be created on demand, and torn down once their purpose has expired.

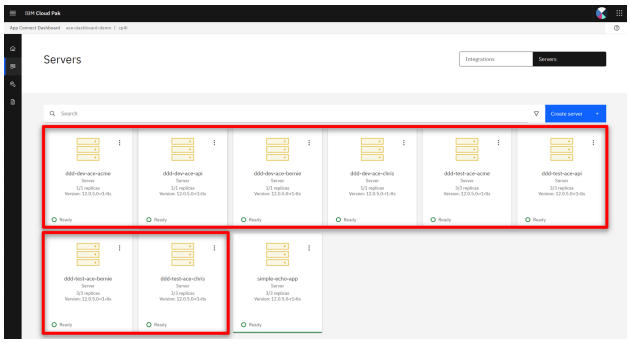
After only a short while we will see that the integration servers for our solutions are started, but what about the rest of our insurance integration solution?

Action 5.1.1
- Return to the App Connect Dashboard server window from earlier in the demo.



Action 5.1.2

- Show the **ddd-dev-ace servers**. If you don't see them, refresh the page.



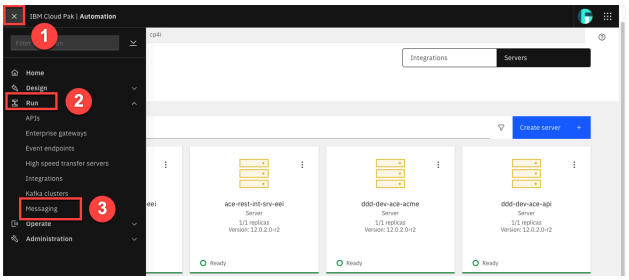
5.2 - Check the queue deployment in MQ Explorer

Narration

Navigating to the messaging capability, we can see that new queue managers have been created with the appropriate queues. These will allow the aggregating integration to initiate all the quotation requests to the insurers in parallel.

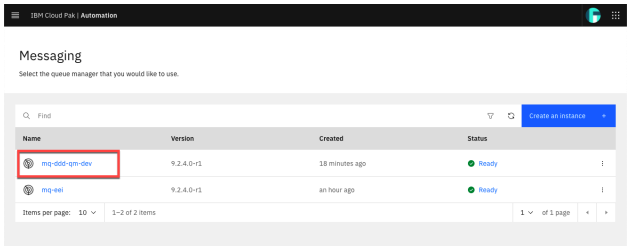
Action 5.2.1

- Open **Menu** (1). Select **Run** (2), then **Messaging** (3).



Action 5.2.2

- Show the **mq-dd-qm-dev** server.



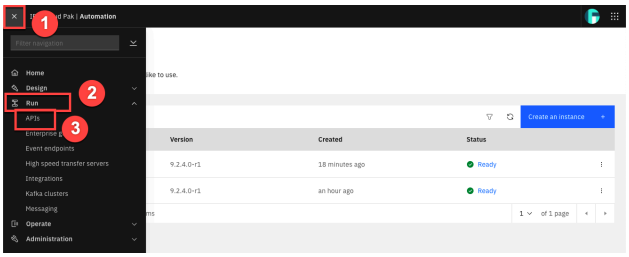
5.3 - Check the API deployment

Narration
Navigating to API management Portal, we can see the APIs have been deployed into the appropriate products.

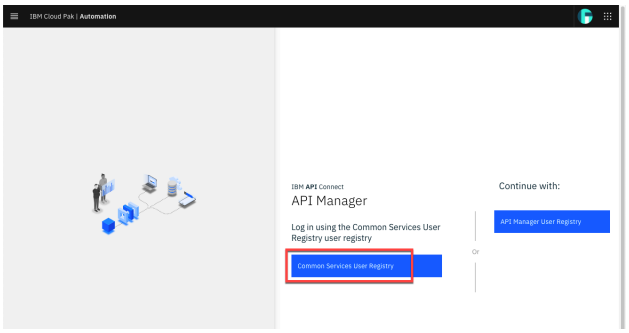
Potential partners who want to use the insurance quote aggregators' new aggregation API will be able to come to the portal and self-subscribe to use it.

It's worth noting that as we navigated between different underlying components we remained in the same user interface, logged in as the same user, and we saw a consistent look and feel to the way each of the components was administered. Furthermore, the navigation and administration capabilities are all governable by a common role-based access control mechanism inherited from the underlying OpenShift platform.

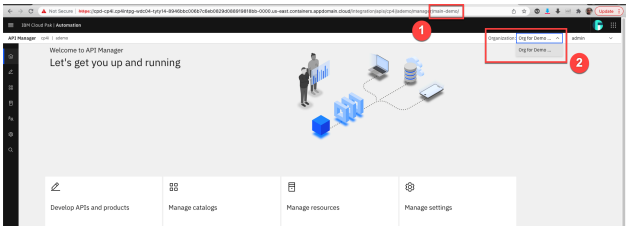
Action 5.3.1
- Open **Menu** (1). Select **Run** (2), then **API** (3).



Action 5.3.2
- Select **Common Services User Registry**.

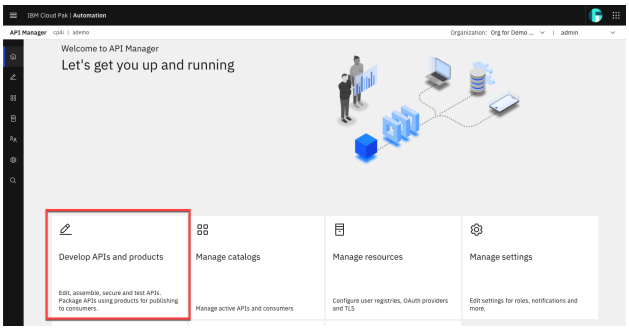


Action 5.3.3
- Check that you are using **main-demo** organization (1). If not, click **Organization** combobox and select the other organization available (2).



Action 5.3.4

- Click **Develop APIs and products**.



6 - Demonstrating availability and scalability

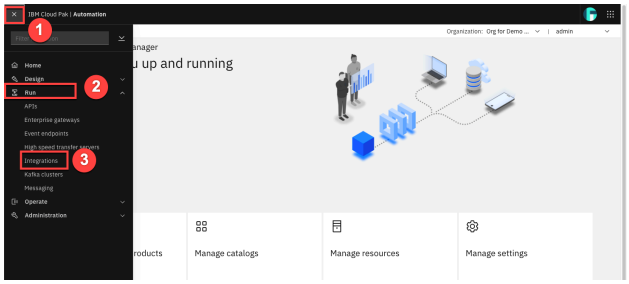
6.1 - Scale up the replicas

Narration
Let's imagine that our new insurance quote aggregator API is much more popular than we expected. We may want to scale up the number of replicas of the integration server to handle the load.

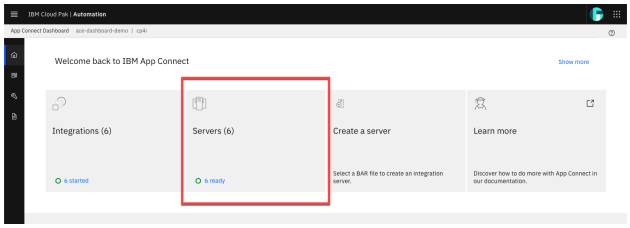
For simplicity, we will edit the custom resource definition directly through the UI so we can quickly see the scaling taking place. However, note that in a real scenario we would probably now do this in the source code repository and let the GitOps pipeline bring the change into production.

We will change the number of replicas, and say move it from three to four. That provides "future state" requirements and the underlying Kubernetes infrastructure will now do all the operational work needed to get it to that new state. No infrastructure expansion project, no manual infrastructure changes, just a change of a number in a file.

Action 6.1.1
- Open **Menu** (1). Select **Run** (2), then **Integrations** (3).

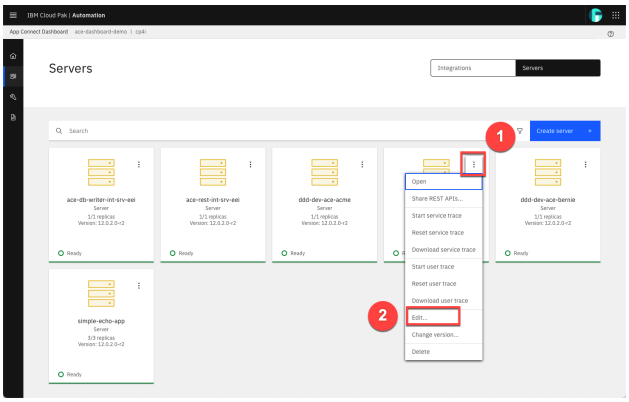


Action 6.1.2
- Open **Servers**.



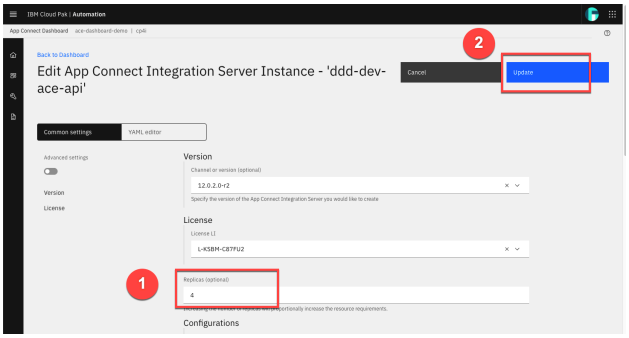
Action 6.1.3

- Click the **ddd-dev-ace-api** (1) context menu and select **Edit** (2).



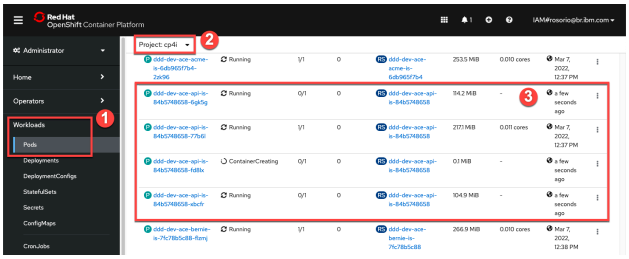
Action 6.1.4

- Type **4** in the **Replicas** field (1). Click **Update** (2).



Action 6.1.5

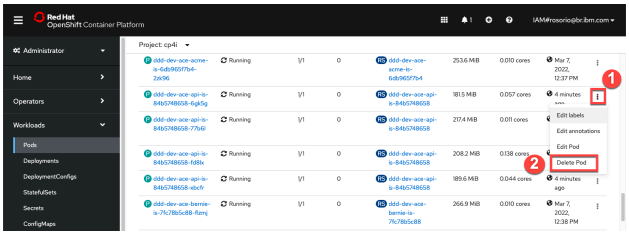
- On the OpenShift Web Console page, open the **Workloads > Pods** page (1), filter by **cp4i** project (2) and show the four pods of the ddd-dev-ace-api (3).



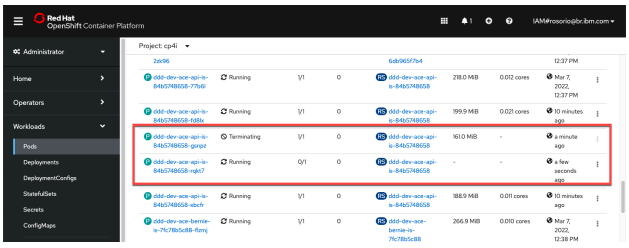
6.2 - Show high availability

Narration
Finally, we're going to emulate a failure by deleting one of the pods that looks after the integration containers and watching it come back up again. We will see Kubernetes and OpenShift doing its job, making sure the state that we've requested matches with what is actually deployed and running. This promise of operational reliability is not something you have to build in, it is just fundamental to the way that Kubernetes works.

Action 6.2.1
- On OpenShift Web Console Pods page, click the context menu of a ddd-dev-ace-api pod and select **Delete Pod** (2).



Action 6.2.2
- Show the re-creation of the pod.



Summary

In this demonstration an insurance broker automated deployment and operations for an API providing an aggregated list of insurance quotes to customers. The API worked with application integration, API management, messaging, and multiple integration styles. With Kubernetes handling much of the daily infrastructural and operational tasks, the broker was able to focus on defining and implementing other value add features for customers.