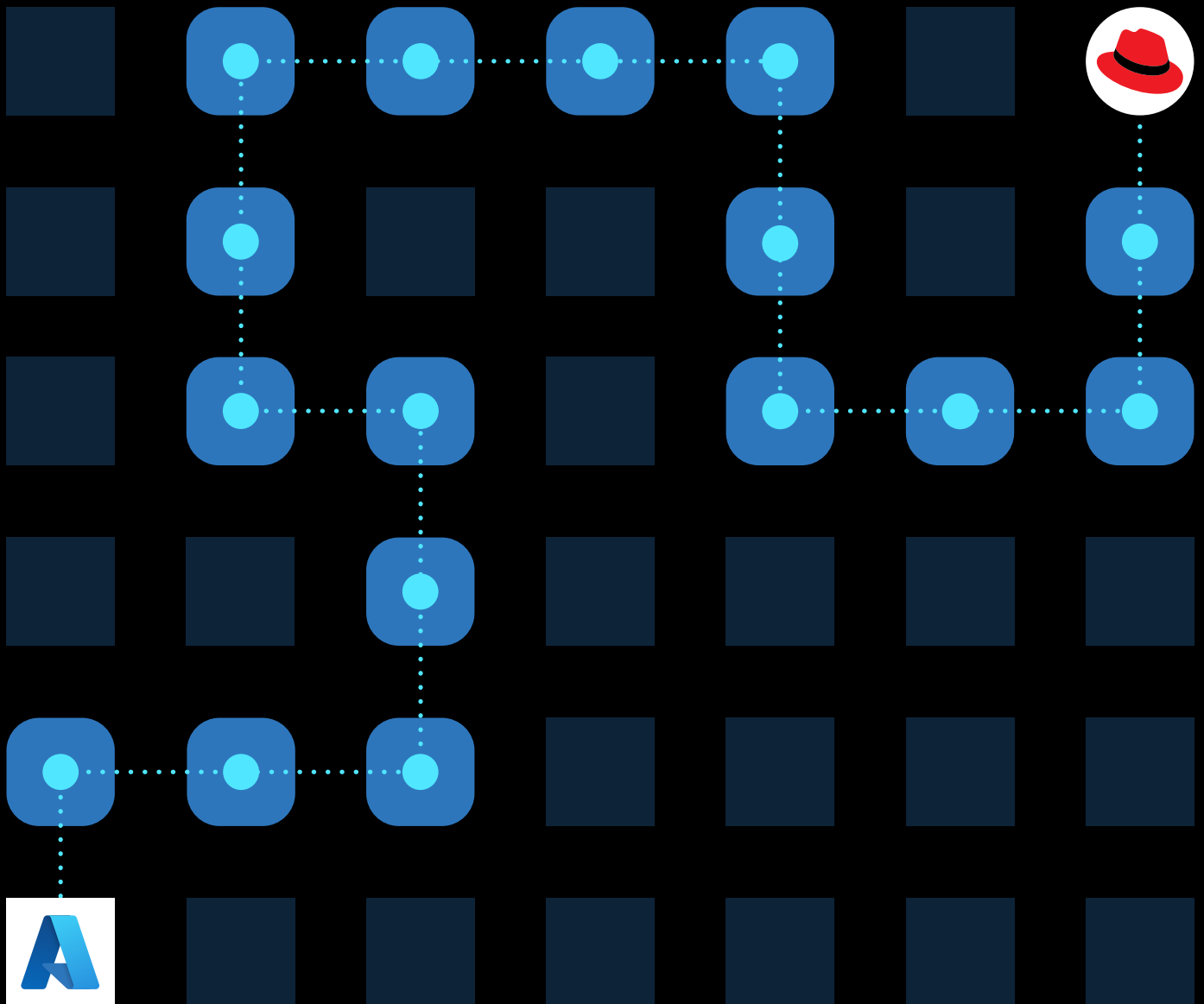


Getting started with Azure Red Hat OpenShift

From proof of concept to production



Getting started with Azure Red Hat OpenShift

3 /

Chapter 1

Talk to Microsoft and Red Hat

35 /

Chapter 5

Provisioning an Azure Red Hat OpenShift cluster

102 /

Chapter 9

Integration with other services

6 /

Chapter 2

Introduction to Red Hat OpenShift

42 /

Chapter 6

Post-provisioning – Day 2

112 /

Chapter 10

Onboarding workloads and teams

13 /

Chapter 3

Azure Red Hat OpenShift

59 /

Chapter 7

Deploy a sample application

117 /

Chapter 11

Conclusion

25 /

Chapter 4

Pre-provisioning – enterprise architecture questions

81 /

Chapter 8

Exploring the application platform

119 /

Chapter 12

Glossary

Chapter 1

Talk to Microsoft and Red Hat

If you are looking at Azure Red Hat OpenShift, we want to connect and talk with you. While this guide provides helpful direction to using the platform, Red Hat and Microsoft can provide plenty of resources that can go beyond this guide and help you take the experience further. We jointly want to make sure that Azure Red Hat OpenShift is the application platform that meets your application innovation needs.

Azure Red Hat OpenShift was created for one simple reason: customers like you asked for it. More than ever before, our joint customers—enterprises as well as small organizations—are deploying Red Hat's portfolio to Microsoft Azure. Although OpenShift on Azure as a self-managed offering has been fully supported for several years, the setup, deployment, and Day-2 operations in managing the cluster require Kubernetes expertise and time to manage. That takes crucial time away from business goals.

As more and more customers roll out successful deployments with the managed offering, Azure Red Hat OpenShift, we're seeing them spend far less time on setup and Day-2 operations and more time focused on their applications.

We've created this guide, based on our hands-on experience, to provide our customers with best practices for building applications in Azure Red Hat OpenShift. This was written as a self-help guide. You can choose to read the chapters sequentially—from the introduction to the conclusion—or just search for the specific information you need.

Who this guide is for

This is meant as a guide for technical audiences—developers, operators, and platform architects—who are looking to bolster their application building and deployment capabilities by using Azure and Red Hat OpenShift for the full-service deployment of fully managed Red Hat OpenShift clusters.

What this guide covers

In this guide, we'll walk you through the key topics for understanding and adopting Azure Red Hat OpenShift, from a proof of concept within your organization to production deployment.

- Chapter 2, Introduction to Red Hat OpenShift*, begins by introducing you to Red Hat OpenShift and the reasons why so many developers, operators, and platform architects choose it as their application platform, and how they derive so much utility from it. After, you will learn why Red Hat OpenShift is a preferred platform.
- Chapter 3, Azure Red Hat OpenShift*, goes on to explain the managed service and how it is delivered to you as a customer.
- Chapter 4, Pre-provisioning – enterprise architecture questions*, covers important questions you should consider prior to deploying Azure Red Hat OpenShift. This includes a lot of best practice guidance that we learned from speaking to many customers who have done this for real.
- Chapter 5, Provision an Azure Red Hat OpenShift cluster*, points to the key resources needed to deploy an Azure Red Hat OpenShift cluster in the official documentation.
- Chapter 6, Post-provisioning - Day 2*, covers the post-provisioning tasks, often referred to as "Day 2." Where possible, this guide tries to call out how the managed service, Azure Red Hat OpenShift, makes this easier than if you had to cover these topics yourself.
- Chapter 7, Deploy Applications onto Red Hat OpenShift*, is a brief guide to deploying applications on the platform, but it should be noted that this is no different than Red Hat OpenShift running in any other environment.

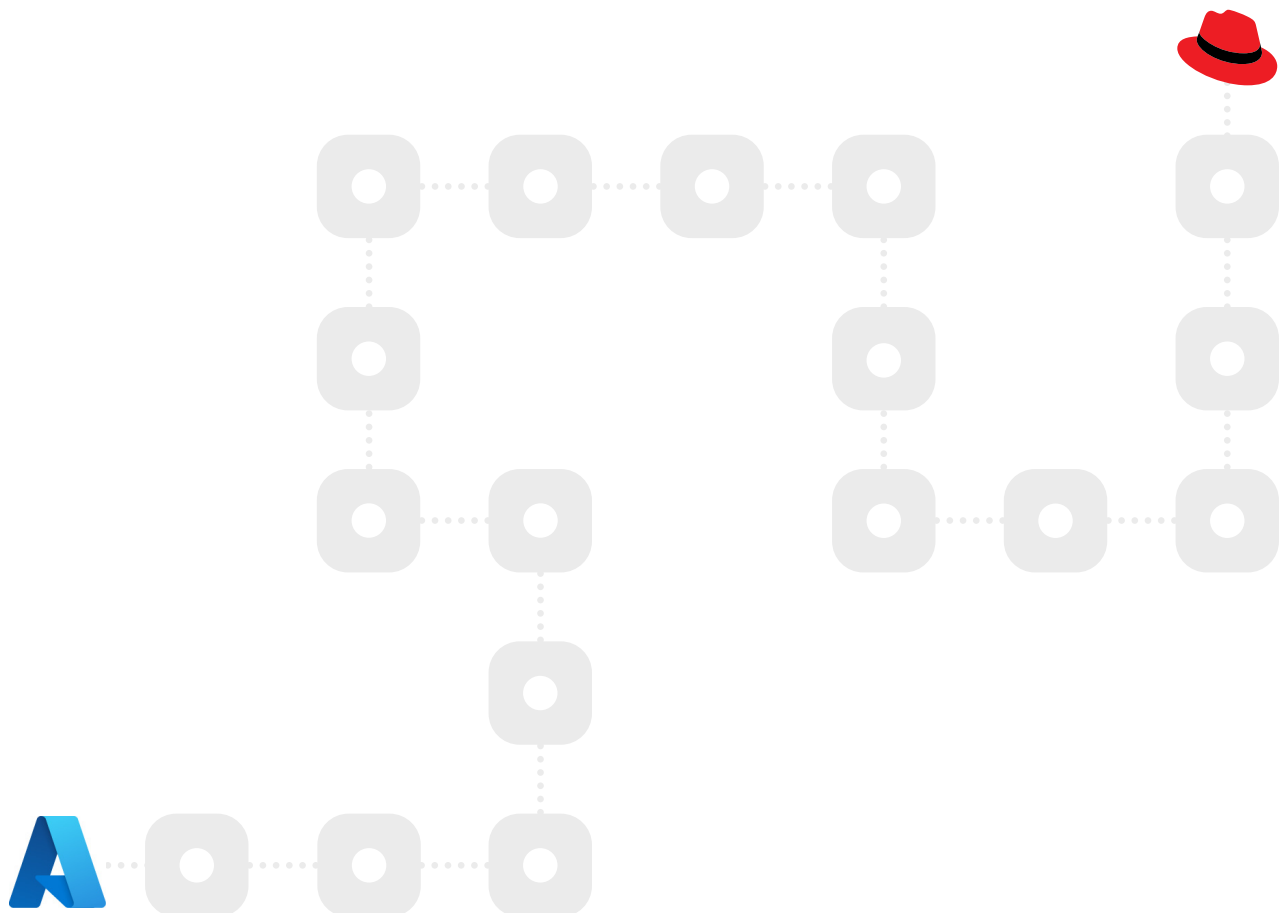
Chapter 8, Exploring the Application Platform, provides a guided overview of some key functionality of the application platform – Container Registry, Pipelines, serverless, and similar.

Chapter 9, Integration with other Services, covers platform integration, using Azure Service Operator, Azure DevOps, and similar services.

Chapter 10, Onboarding workloads and teams, finishes off by providing some best practices and recommendations on how to onboard application teams and gain some traction in adopting the service within your organization.

Chapter 11, Conclusion, contains the conclusion.

Chapter 12, Glossary, serves as a glossary.



Chapter 2

Introduction to Red Hat OpenShift

This chapter provides a brief introduction to what Red Hat OpenShift is, how it is used, and the benefits of the service typically seen by Red Hat customers. This serves as a useful primer if you are just getting introduced to OpenShift for the first time, or as a quick refresher if you're looking to learn a bit more about the platform.

Red Hat OpenShift overview

[Red Hat OpenShift](#) is an enterprise-ready application platform with full-stack automated operations for managing hybrid cloud and multicloud deployments. It is optimized to improve developer productivity and promote innovation. With automated operations and streamlined lifecycle management, Red Hat OpenShift empowers development teams to build and deploy new applications and helps operations teams provision, manage, and scale Kubernetes platforms.

Development teams have access to validated images and solutions from hundreds of partners with security scanning and cryptographic signing throughout the delivery process. They can access on-demand images and get native access to a wide range of third-party cloud services, all through a single platform.

Operations teams are given visibility into deployments wherever they are, and across teams, with built-in logging and monitoring. Red Hat Kubernetes operators embed unique application logic that enables the service to be functional, not simply configured but also tuned for performance and updated and patched from the OS with a single touch. In short, Red Hat OpenShift is a one-stop shop that enables organizations to unleash the power of their IT and development teams.

OpenShift cloud services – cost savings and business benefits

Thousands of customers trust Red Hat OpenShift to change the way they deliver applications, improve their relationships with customers, and gain competitive advantages to be leaders in their industries. IDC's research, outlined in [The Business Value of Red Hat OpenShift, March 2021](#), demonstrates the strong value that interviewed organizations have achieved with the Red Hat OpenShift platform by delivering higher-quality and more timely applications and features to their businesses, while also optimizing their development and IT-related costs and staff time requirements.

Key metrics are as follows:

- 636% return on investment over 5 years
- 10-month payback time
- 54% lower 5-year cost of operations
- 3x more new features per year
- 20% higher application developer productivity
- 71% less unplanned downtime
- 21% more efficient IT infrastructure teams

Source: IDC white paper, sponsored by Red Hat, *The Business Value of Red Hat OpenShift*, doc # US47539121, March 2021

Building on top of this value of Red Hat OpenShift, the Red Hat OpenShift cloud services, which includes Azure Red Hat OpenShift, deliver additional business benefits. In the Forrester study titled [The Total Economic Impact™ of Red Hat OpenShift Cloud Services - Cost Savings and Business Benefits](#), several key financial benefits were highlighted.

Key financial benefits of OpenShift cloud services are as follows:

- 468% Return On Investment (ROI)
- \$4.08M Net Present Value (NPV)
- 6-month payback time

Beyond these financial benefits, the key findings of quantified benefits of the Forrester report include:

- **Improved development velocity:** Using Red Hat OpenShift cloud services allows organizations to shorten their development cycle by up to 70%.
- **20% of developer time is recaptured from infrastructure maintenance work:** Interviewees noted that Red Hat OpenShift cloud services eliminated the need for developers to maintain the application development infrastructure, allowing them to fully focus on building the product or solution. Over 3 years, this developer time recapture is worth more than \$2.3 million.
- **Improved operational efficiency by 50%:** Since Red Hat OpenShift cloud services are a managed service, interviewees noted that using the solution meant they can reassign 50% of DevOps employees who were previously responsible for managing the infrastructure to other work that is more productive. Over 3 years, this increased operational efficiency is worth more than \$1.3 million.*

The report also found the following unquantified benefits:

- **Developer satisfaction and retention:** Interviewees highlighted that developers benefited from Red Hat OpenShift cloud services, allowing them to break down updates into smaller pieces, reducing the pressure of extensive testing in a very limited timeline and reducing the need to respond to fire drills once in production.
- **Security and reduced risk:** Interviewees shared how Red Hat OpenShift cloud services automated certain features and security updates, eliminating the need for manual maintenance while still ensuring that their environment is secure.
- **Reliability:** Interviewees noted that using Red Hat OpenShift cloud services made their application platform more reliable over the long run, as there are fewer outages or system failures, even with an expanding environment.
- **Portability and business continuity:** Interviewees also noted that Red Hat OpenShift cloud services ensured business continuity and assisted with their disaster recovery strategy due to its portability, scalability, and flexibility.

Source: Forrester, *The Total Economic Impact of Red Hat OpenShift Cloud Services*, December 2021

*Further reading: [Enterprises accelerate agility with cloud services](#)

"Red Hat OpenShift or bare Kubernetes?" – The cost of building your own Kubernetes application platform

Red Hat OpenShift is often referred to as "enterprise Kubernetes," but it can be hard to understand what that really means at first glance. Customers frequently ask, "OpenShift or bare Kubernetes?" However, it's important to understand that **OpenShift already uses Kubernetes**. In the overall OpenShift architecture, Kubernetes provides both the foundation on which the OpenShift platform is built and much of the tooling for running OpenShift.

Kubernetes is an incredibly important open-source project—one of the key projects of the Cloud Native Computing Foundation and an essential technology in running containers.

However, the real question that potential users of OpenShift might have is, "**Can I just run my applications with Kubernetes alone?**" Many organizations start by deploying Kubernetes and find that they can get a container, even an enterprise application, running in just a few days. However, as the Day-2 operations begin, security requirements arise, and more applications are deployed, some organizations find themselves falling into the trap of building their own **platform as a service (PaaS)** with Kubernetes technology. They add an open-source ingress controller, write a few scripts to connect to their **continuous integration/continuous deployment (CI/CD)** pipelines, and then try to deploy a more complex application... and that is when the problems start. If deploying Kubernetes were the tip of the iceberg, then the complexity of Day-2 management would be the hidden, vast, ship-sinking bulk of that same iceberg under the water.

While it's possible to start solving these challenges and problems, it often takes an operations team of several people, and several weeks and months of effort to build and maintain this "custom PaaS built on Kubernetes." This leads to inefficiency in the organization, complexity in supporting this from a perspective of security and certification, as well as having to develop everything from scratch when onboarding developer teams. If an organization were to list out all the tasks of building and maintaining this custom Kubernetes platform—that is, Kubernetes—with all the extra components needed to run containers successfully, they would be grouped up as follows:

- **Cluster management:** This includes installing OSes, patching the OS, installing Kubernetes, configuring CNI networking, authentication integration, Ingress and Egress setup, persistent storage setup, hardening nodes, security patching, and configuring the underlying cloud/multicloud.
- **Application services:** These include log aggregation, health checks, performance monitoring, security patching, container registry, and setting up the application staging process.
- **Developer integration:** This includes CI/CD integration, developer tooling/IDE integration, framework integration, middleware compatibility, providing application performance dashboards, and RBAC.

While there are many more actions and technologies that could be added to the list—most of those activities are essential for any organization to seriously use containers—the complexity, time, and effort in just setting all of that up is insignificant to the ongoing maintenance of those individual pieces. Each integration needs to be thoroughly tested, and each component and activity will have a different release cycle, security policy, and patches.

What do you get with Red Hat OpenShift as opposed to bare Kubernetes?

When an organization comes to running containers in production, built on just bare Kubernetes, the components mentioned in the previous section are normally installed and integrated together to create a kind of application platform of their own design.

Azure Red Hat OpenShift combines all of the components mentioned in the previous section into a single platform, bringing ease of operations to IT teams while giving application teams what they need to execute their tasks. All of these topics will be covered in greater detail later in the guide, but with this in mind, let's take a look at some of the key differences between the two:

- **Ease of deployment:** Deploying an application in Kubernetes can be time consuming. It involves pulling your GitHub code onto a machine, spinning up a container, hosting it in a registry such as Docker Hub, and finally, understanding your CI/CD pipeline, which can be very complicated. OpenShift, on the other hand, automates the heavy lifting and the back-end work, only requiring you to create a project and upload your code.
- **Security:** Today, we see that most Kubernetes projects are worked on in teams of multiple developers and operators. Even though Kubernetes now supports controls such as RBAC and IAM, it still requires manual setup and configuration, which takes time. Red Hat and OpenShift have done a great job of identifying security best practices after years of experience, which are available to customers out of the box. You simply add new users and OpenShift will handle things such as name-spacing and creating different security policies.
- **Flexibility:** In using Azure Red Hat OpenShift, you're able to take advantage of well-known best practices of deployment, management, and updating. All the heavy lifting within the back end is taken care of for you without the need for much finger pushing, enabling you to deliver your apps quicker. As well as being nice for teams that like being told how to get things done and benefit from a streamlined approach, the Kubernetes platform allows you to manually customize your CI/CD DevOps pipeline, which offers more room for flexibility and creativity when developing your processes.

- **Day-to-day operations:** Clusters comprise of a group of multiple VMs and inevitably, your operations teams will need to spin up new VMs that need to be added to a cluster. The configuration process through Kubernetes can be time-consuming and complex, requiring scripts to be developed to set up things such as self-registration or cloud automation. With Azure Red Hat OpenShift, cluster provisioning, scaling, and upgrade operations are automated and managed by the platform.
- **Management:** While you can take advantage of the Kubernetes standard tools and dashboards that come with any distribution, most developers need a more complete and robust platform. Azure Red Hat OpenShift offers a great web console that builds on the Kubernetes APIs and capabilities for operations teams to manage their workloads.

In *Chapter 8, Exploring the application platform*, there is a guided description of many of the important value-added capabilities of Azure Red Hat OpenShift.

Summary

Azure Red Hat OpenShift is being selected by many joint Red Hat and Microsoft customers as their preferred application platform to adopt containerized applications.

In the next chapter, we'll explore Red Hat OpenShift as a cloud service, diving into the architecture, integration, and management.

Chapter 3

Azure Red Hat OpenShift

With **Azure Red Hat OpenShift**, you can deploy fully managed Red Hat OpenShift clusters without worrying about building and managing the infrastructure to run it.

Azure Red Hat OpenShift is a cloud-native, on-demand application platform that is jointly engineered, operated, and supported by Red Hat and Microsoft. A specialized **Site Reliability Engineering (SRE)** team automates, scales, and secures OpenShift clusters and works side by side to provide an integrated support experience. With Azure Red Hat OpenShift, there are no virtual machines to operate, and no patching is required. Control plane nodes and worker nodes are patched, updated, and monitored on your behalf by Red Hat and Microsoft. Your Azure Red Hat OpenShift clusters are deployed into your Azure subscription and are included on your Azure bill.

Deliver applications faster with Azure Red Hat OpenShift:

- Empower developers to innovate through built-in CI/CD pipelines, and then easily connect your applications to hundreds of Azure services such as MySQL, PostgreSQL, Redis, and Azure Cosmos DB.
- Replace complexity and remove barriers to productivity with automated provisioning, configuration, and operations.
- Scale on your terms, where you can start a highly available cluster with three worker nodes in a few minutes and scale as your application demand changes with a choice of standard, high-memory, or high-CPU worker nodes.
- Enterprise-grade operations, security, and compliance with an integrated support experience.

Next, we'll dig into the technical details behind how Red Hat OpenShift is built and operated.

Architecture

Azure Red Hat OpenShift uses Azure infrastructure services—virtual machines, network security groups, storage accounts, and other Azure services—as the foundation of the Red Hat OpenShift installation. The Azure architecture is completely deployed into your Azure subscription as well, making it easy to integrate with other services that are already running within your account.

OpenShift itself is built on top of Red Hat Enterprise Linux CoreOS, which hosts the microservices-based architecture of smaller, decoupled units that work together. On each virtual machine, the kubelet service is running, which is the foundation of the Kubernetes cluster. The database behind this architecture, which is running on the control plane, is **etcd**, a reliable, clustered key-value store.

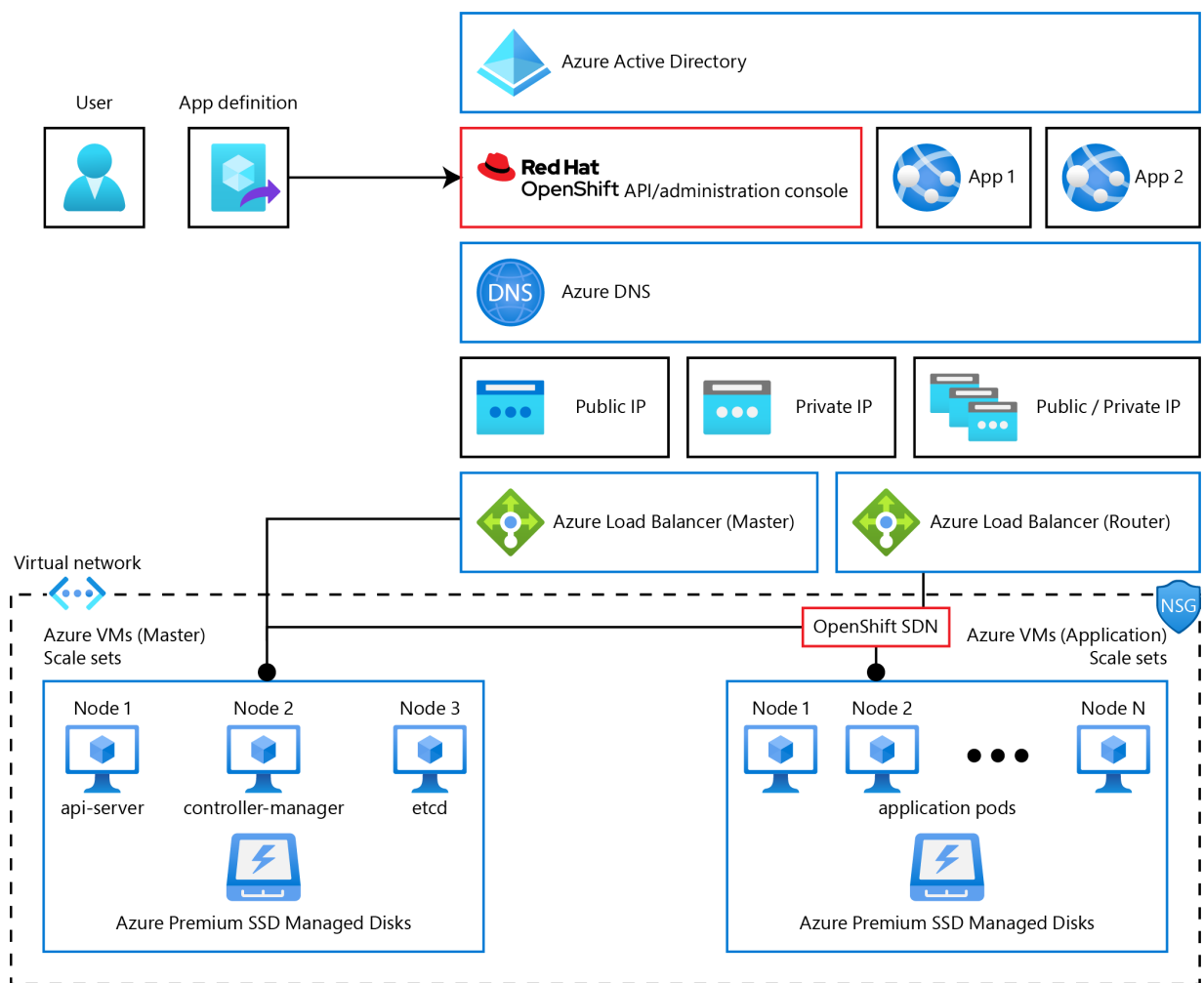


Figure 3.1: Azure Red Hat OpenShift architecture

The following two sections, *Compute – control and worker nodes*, and *Networking*, go into more detail about what is included in this diagram.

Compute – control and worker nodes

Red Hat OpenShift's architecture runs on virtual machines (nodes) that have one of three specific roles—control, infrastructure, or application. However, Azure Red Hat OpenShift version 4 does not support infrastructure nodes at the time of writing, so this book will just cover control and worker nodes.

The **control** nodes (historically called **master** nodes) are virtual machines that contain essential components for the Kubernetes cluster. These include the API server, controller manager server, scheduler, and etcd. They manage the Kubernetes cluster and schedule pods to run on the worker nodes.

- **The Kubernetes API server** validates and configures the data for pods, services, and replication controllers. It also provides a focal point for the shared state of the cluster.
- **The Kubernetes controller manager** watches etcd for changes to objects, such as replication, namespace, and service account controller objects, and then uses the API to enforce the specified state. Several such processes create a cluster with one active leader at a time.
- **The Kubernetes scheduler** watches for newly created pods without an assigned node and selects the best node to host the pod.
- **etcd** stores the persistent master state while other components watch etcd for changes to bring themselves into the specified state.

The **application** nodes are where your applications will actually run.

Each node in a Kubernetes cluster runs a service called kubelet, which maintains the **container runtime interface (cri-o)**, a service proxy, and some other essential services that run on each node. All the nodes are connected with OpenShift's software-defined networking technology. In Azure Red Hat OpenShift, this is an **Open Virtual Network (OVN)**, which runs on top of an Azure virtual network.

Azure Red Hat OpenShift creates nodes that run on Azure virtual machines that are connected to Azure disks for storage. You may notice that they have quite large disks—1 TB. That is because, in Azure, the IOPS guarantee for storage performance is linked to the size of the disk. 1 TB in size guarantees sufficient bandwidth for the underlying disk used by the etcd database.

Networking

Azure Red Hat OpenShift requires a single Azure virtual network, with two subnets configured—one for the control plane nodes and one for the worker nodes. The minimum size of both networks is /27 (32 addresses). However, be careful not to set the size to be too small if you intend to scale the cluster later.

The two Azure Load Balancers (labeled **Master** and **Router** in the diagram) direct traffic as follows:

- Master/control plane load balancer: Sends ingress traffic for users of the OpenShift API. Also provides outbound connectivity for the control plane nodes
- Router/application load balancer: Sends ingress traffic to the applications running in OpenShift, via an OpenShift "router" or ingress controller. Also provides outbound connectivity for the worker nodes

An excellent article about the networking configuration, requirements, and limitations can be found in the [networking documentation](#).

Integration with other Azure services

As a native service on Azure, Azure Red Hat OpenShift can be deployed alongside, and integrated with, many of the services you are used to using on Azure. The following is a high-level overview of just some of the Azure infrastructure services where common integrations exist.

Figure 3.2 shows many of the common Azure service integration points for OpenShift on Azure:

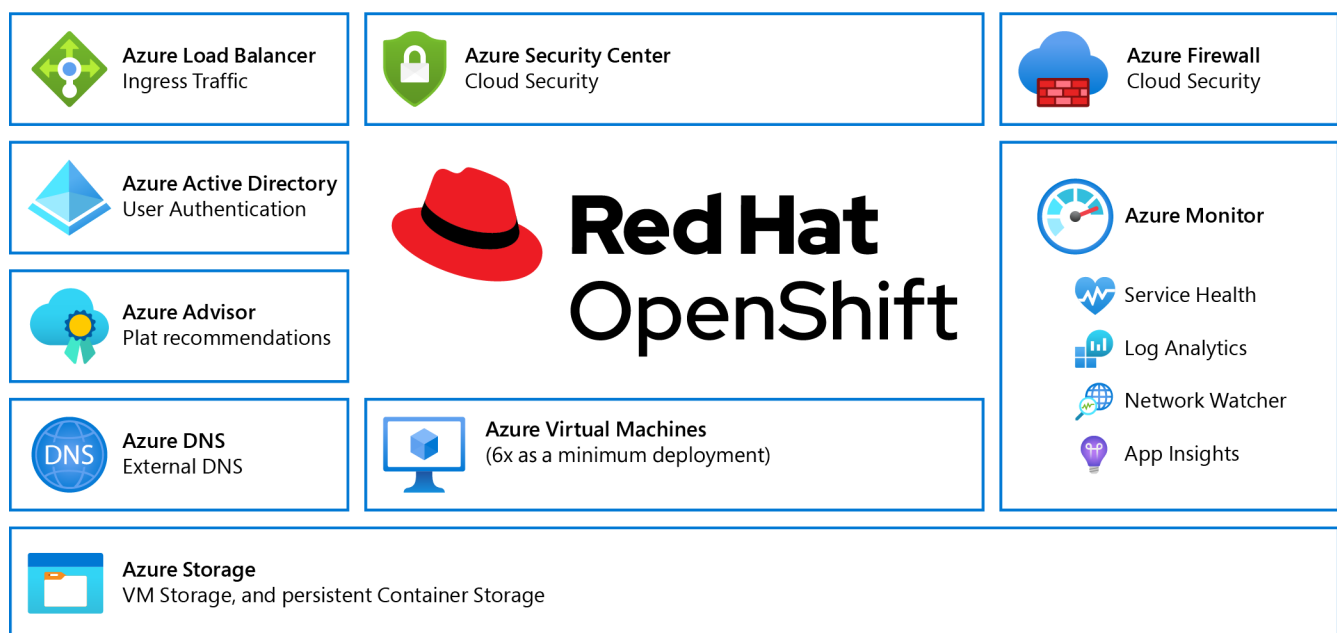


Figure 3.2: Common Azure service integration points

Additionally, applications running on OpenShift can integrate very closely with Azure services by using [Azure Service Operator](#). This is covered in more detail later in *Chapter 9, Integration with other services*.

Management

A simple description to understand what is managed as part of the Azure Red Hat OpenShift service, is to think of everything from the data center, up to the cluster operators as being "managed". Cluster operators are services that run on the control plane nodes and look after monitoring, updates, and the health of the cluster. This means that Microsoft and Red Hat will jointly monitor, maintain, and manage those components to keep the cluster online. Everything above the cluster operators remains the responsibility of the customer.

As a consumer of Azure Red Hat OpenShift, you are given full **cluster-admin** access to the cluster, which means that you also share some responsibility in not breaking parts of your cluster. It's important to understand the [Support Policy](#) to understand what you can, and cannot do with the cluster.

A detailed description of precisely what Microsoft and Red Hat will do as part of the cloud service is described in [Azure Red Hat OpenShift Responsibility Matrix](#).

Authentication and authorization

Azure Active Directory is a common way to provide authentication to Azure Red Hat OpenShift clusters. However, it is not mandatory, and other authentication mechanisms, such as Login with GitHub, or a simple "password file," can be used as alternatives.

When Azure Active Directory is being used, Azure Red Hat OpenShift and Kubernetes APIs will forward authentication requests. Users will present their credentials and will be authorized based on their roles.

The authentication layer identifies the user associated with requests to the Azure Red Hat OpenShift API. The authorization layer then uses information about the requesting user to determine if the request should be allowed.

Configuration instructions for Azure Active Directory are described in the *Authentication - Azure Active Directory* section. This process is functionally very similar if another authentication provider is being used instead of Azure Active Directory.

Authorization is handled in the Azure Red Hat OpenShift policy engine, which defines actions such as "create pod" or "list services" and groups them into roles in a policy document. Roles are bound to users or groups by the user or group identifier. When a user or service account attempts an action, the policy engine checks for one or more of the roles assigned to the user (for example, customer administrator or administrator of the current project) before allowing it to continue.

The relationships between cluster roles, local roles, cluster role bindings, local role bindings, users, groups, and service accounts are set out in *Figure 3.3*:

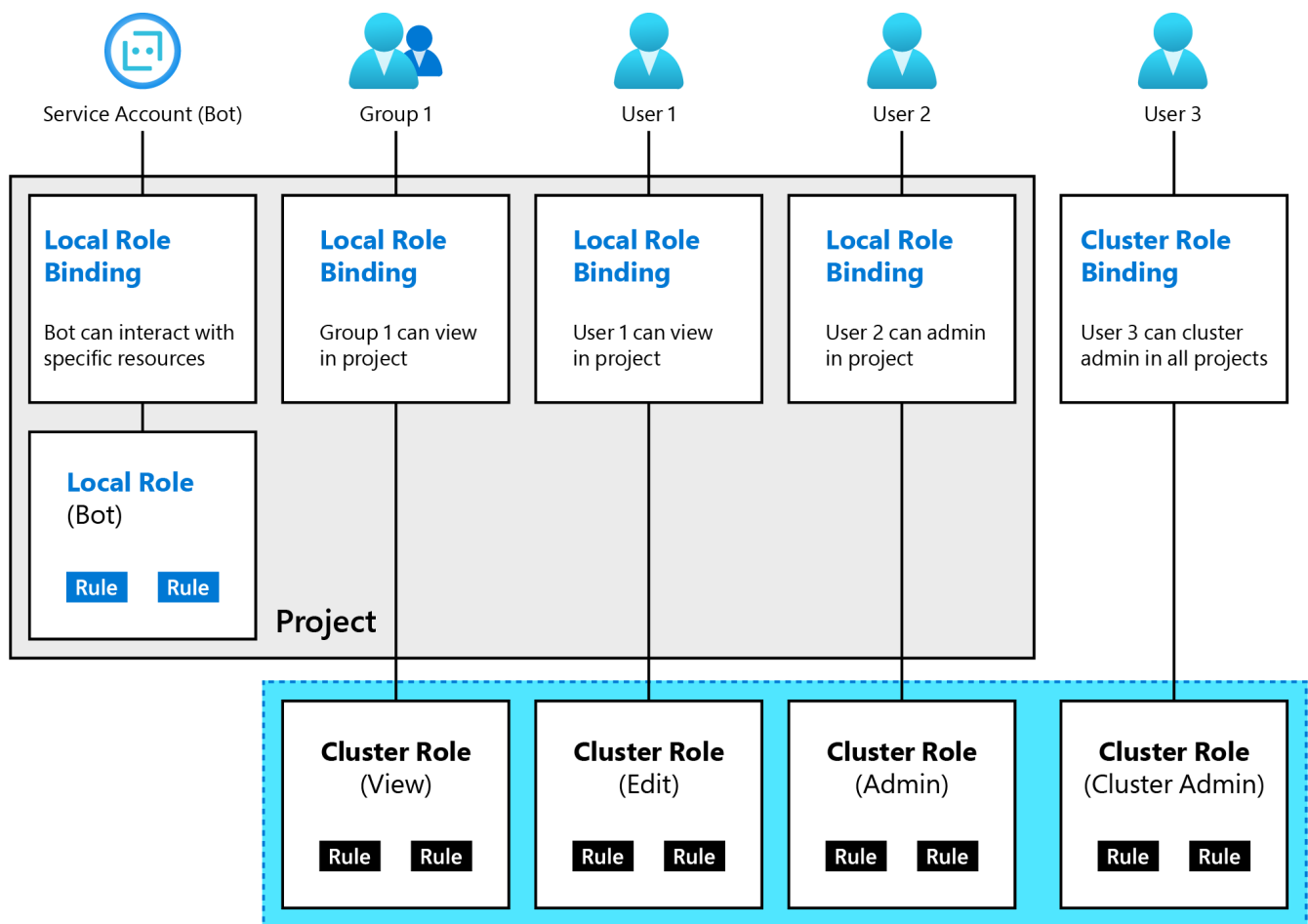


Figure 3.3: Relationships between roles

The Red Hat OpenShift documentation has [a full list of authentication providers](#).

Support

Azure Red Hat OpenShift is unique in the way support is managed. Microsoft and Red Hat support teams work together, alongside the global [Site Reliability Engineering \(SRE\)](#) team, to facilitate the operation of the service.

Customers request support in the Azure portal, and the requests are triaged and addressed by Microsoft and Red Hat engineers to quickly tackle them, whether those are at the Azure platform level or the OpenShift level.

Here is an outline of the integrated support process:

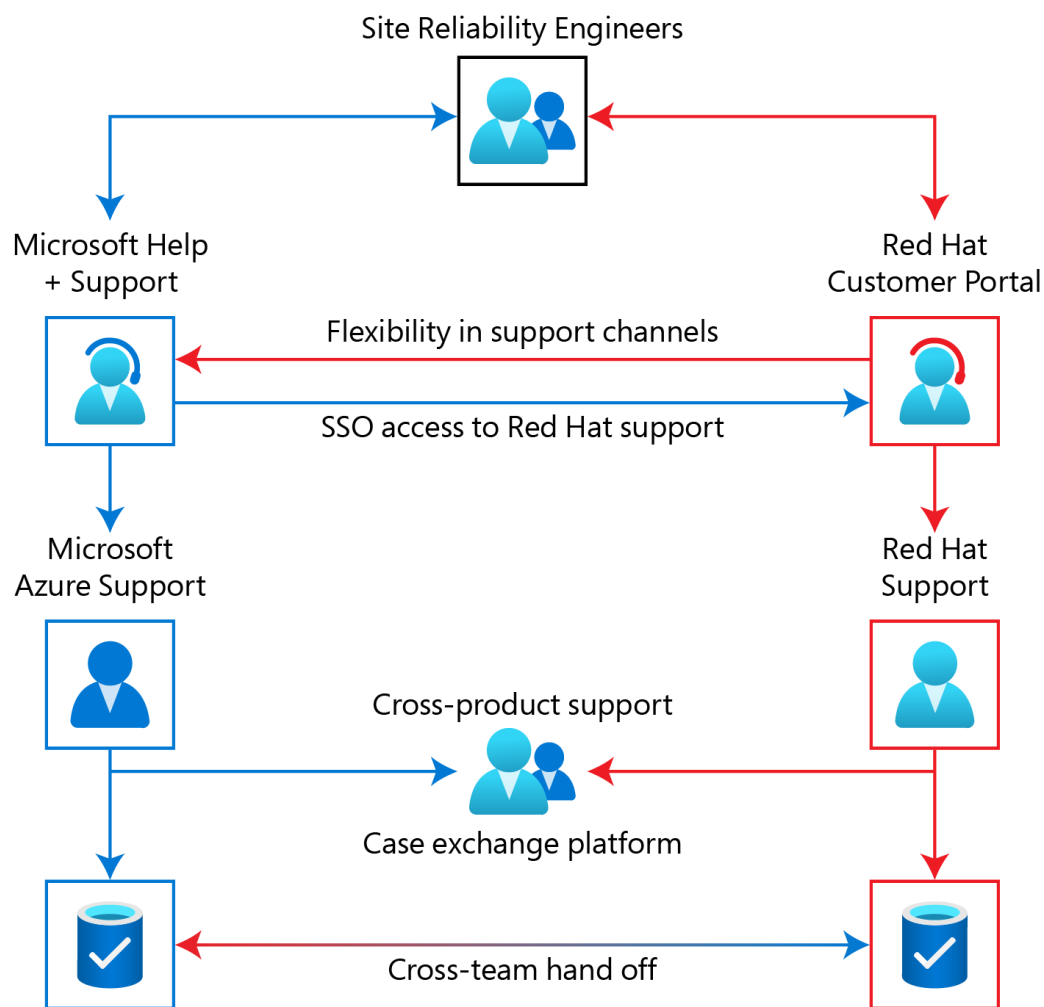


Figure 3.4: The integrated support process

Figure 3.4 shows that customers can raise a support ticket in either the Microsoft support portal or the Red Hat support portal. Note that for access to the Red Hat support portal, the cluster should be registered to OpenShift Cluster Manager.

Support engineers from Microsoft can seamlessly collaborate with Red Hat at any stage, with the consent of a customer via the case exchange platform. This is also the case for Red Hat support engineers when requesting collaboration with Microsoft. Support engineers from both companies have access to the SRE team, which can take remedial actions to repair clusters, if necessary.

Pricing and subscriptions

One major benefit of Azure Red Hat OpenShift over a Do-it-yourself (DIY) installation is that the compute, network, storage, and Azure infrastructure, as well as the OpenShift subscriptions, are all billed via your Azure Subscription, as opposed to being billed separately. To get an indicative idea of the costs of the solution, simply add Azure Red Hat OpenShift to the quote builder in the [Azure Pricing Calculator](#).

The following is a guide to understanding the pricing page:

1. Type *openshift* in the search box and add it to a new estimate.

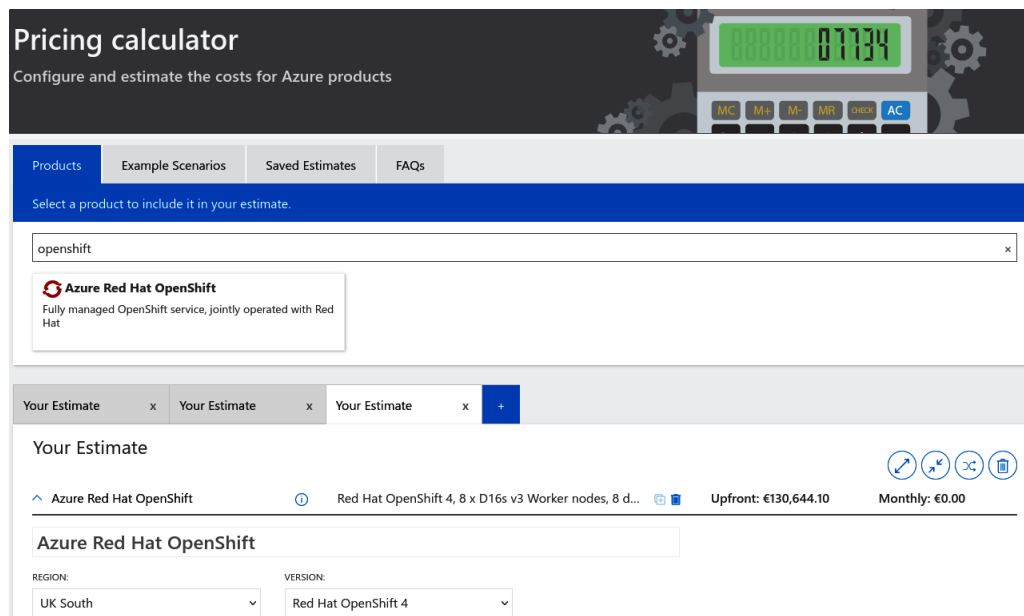


Figure 3.5: The Pricing calculator panel

2. At the bottom of the page, you'll find a drop-down combo box to change the pricing unit to your local currency. In this example, GBP (£) is used.
3. Set your Azure region for deploying Azure Red Hat OpenShift. The price will fluctuate slightly between regions to account for the different compute costs across Azure regions.

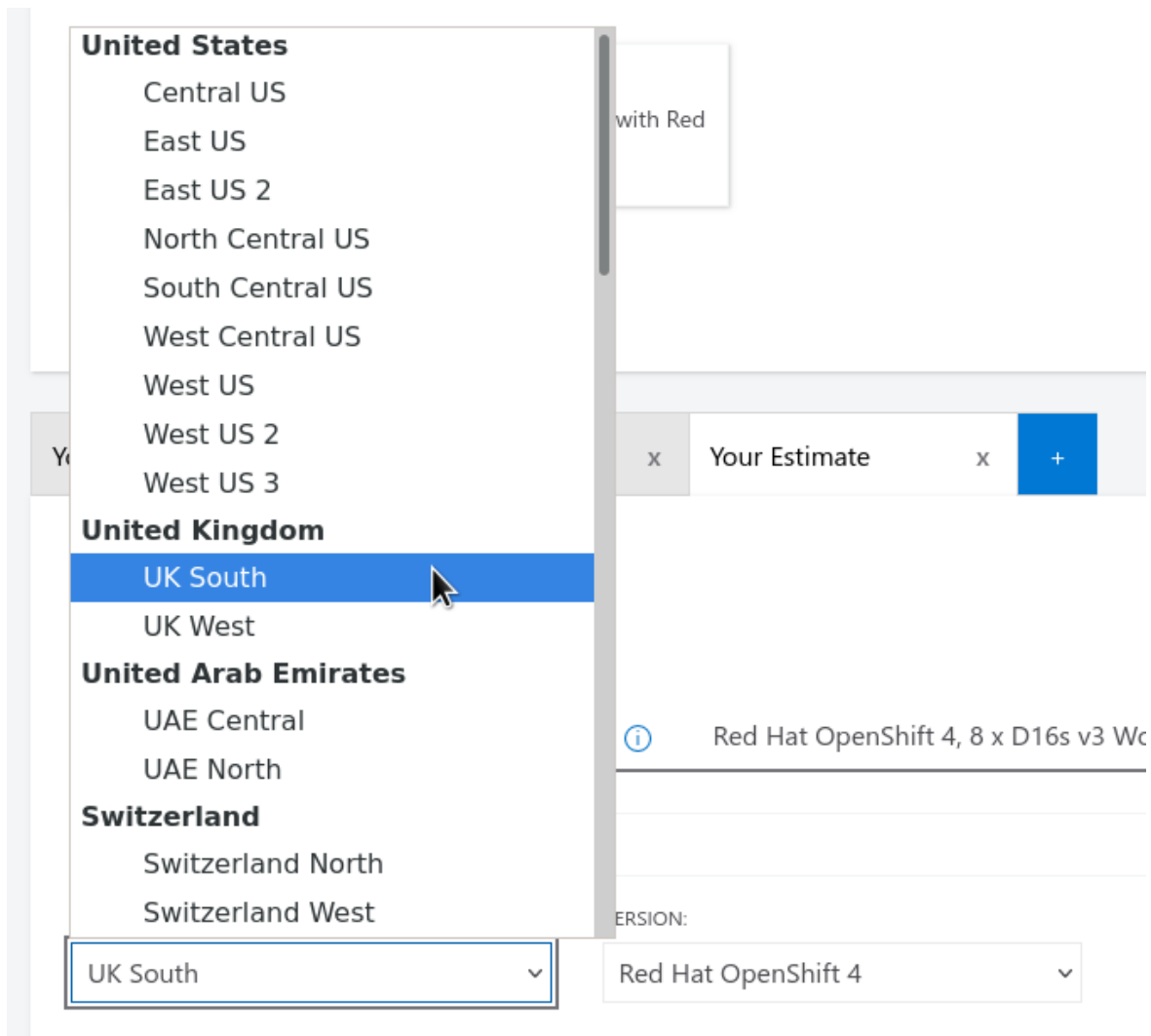


Figure 3.6: Selecting the Azure region

4. **Worker Nodes** are where your actual applications will run. Under **Savings Options**, you'll find two sections. **License** refers to the cost of the OpenShift subscriptions, whereas the **Virtual Machine** section refers to the cost of the compute services necessary to run the cluster. The maximum number of supported worker nodes in a cluster is 100.

Worker Nodes

INSTANCE:

D4s v3: 4 vCPU(s), 16 GB RAM

3

Worker Nodes

Savings Options

License

- Pay as you go
- 1 year reserved (~33% savings)
- 3 year reserved (~56% savings)

£187.07

Average per month

(£2,244.83 charged upfront)

Virtual Machine

- Pay as you go
- 1 year reserved (~37% savings)
- 3 year reserved (~57% savings)

PAYMENT OPTIONS:

Upfront

£239.99

Average per month

(£2,879.84 charged upfront)

Figure 3.7: Details of the Worker nodes

5. **Managed OS Disks** refers to the size of the disks used by Red Hat CoreOS for the operating system partitions. This does not refer to the storage used by application persistent volumes, which are provisioned separately.

Managed OS Disks

DISK SIZE:

P10: 128 GiB, 500 IOPS, 100 MB/sec

3

Disks

×

£17.77

Per month

Figure 3.8: Managed OS Disks

6. There is a similar section for **Master Nodes**, which are typically referred to as control nodes in modern parlance. Note that the control nodes have a much larger disk associated with them than the worker nodes. This is because they require a higher IOPS and throughput, which comes with larger disk sizes on Azure. The exact number of control plane nodes must always be three, for cluster stability.

Note that the calculator is designed to show indicative pricing, actual prices, of course, will fluctuate depending on usage.

Azure reserved virtual machine instances

A **reserved instance** refers to a commitment to consume that resource for a length of time—1 year or 3 years. There are reserved instance options for Azure Red Hat OpenShift virtual machines.

Organizations typically choose reserved instances to get a substantial discount on the IaaS, when they know that a cluster is going to be running for a longer period of time. For example, production environments are often run using reserved instances. It is worth noting that reserved instances do not change the service level or architecture of the cluster.

[More information about Azure reserved instances](#)

Summary

This chapter described the specifics of the Azure Red Hat OpenShift managed cloud service. It provided a high-level overview of the architecture, and briefly discussed integration with other Azure services (which we'll cover in more detail in *Chapter 9, Integration with other services*), along with management, authentication, support, and pricing considerations.

The next chapter will focus on the questions and decisions an organization will need to answer in the Pre-provisioning phase of deploying Azure Red Hat OpenShift.

Chapter 4

Pre-provisioning – enterprise architecture questions

Many organizations will see Azure Red Hat OpenShift within the Azure portal, and by reading through the documentation once, it's possible to successfully deploy a cluster with Red Hat OpenShift with very little additional effort. However, if a bit more time is given to planning deployments, and asking a few questions in advance, it's possible to save a lot more time in the future deleting and reprovisioning clusters.

This chapter is based on real-world, hands-on experience of working with many customers. It covers many of the typical Pre-provisioning questions that should be addressed first. This chapter covers:

- How many clusters are needed, including staging, production, and similar
- Public and private network visibility
- Hybrid connectivity, such as connection to on-premises solutions

We'll start with how to work out how many clusters you need.

How many clusters are needed?

There are many deployment patterns for OpenShift, but a common question is "How many clusters does my organization need?". Of course, that is a decision that depends on the organization, but in the paragraphs that follow, there is some guidance to help you get to that number.

Life cycle stages: development, testing, production

Most organizations of any scale will deploy their enterprise IT systems with some sort of life cycle stages. This approach is also sometimes called a staging pattern. The three most commonly seen stages are development, testing, and production. Having multiple stages allows changes to applications and deployments of applications to be tested in a safe environment before those changes reach a production environment. The most common, and recommended, staging pattern is to have at least three separate Azure Red Hat OpenShift clusters:

- **Development:** Where any developers and operators are allowed to test anything. This can be one large "sandbox" cluster, but it's more normal, and often safer, to have small, short-lived clusters where tests are created and destroyed frequently.
- **Testing:** Where upcoming cluster changes, such as patches or configuration changes, are tested and validated prior to being released to production. This is often called "preproduction" as well in some organizations, but preproduction can also be another environment entirely.
- **Production:** Where your real applications run.

In addition to the preceding, some organizations will have additional environments, such as integration testing environments, but only you will know how many staging environments best fit your organization. Look at other similar enterprise applications for common deployment patterns if you are unsure.

In situations where Azure Red Hat OpenShift is being used for non-critical applications, it could be acceptable within your organization to have just two clusters (merging development and testing), or even a single cluster that includes development, testing, and production. This has the advantage of keeping cloud costs down and reducing the overall number of clusters that need to be managed. When operating from just a single cluster, administrators may choose to use separate namespaces for development, testing, and production. However, running just a single cluster has disadvantages:

- Changes that affect the entire cluster (like software patches) could introduce issues in production that would have been picked up and prevented if they were run against a test environment.
- If an application in a testing or development environment behaves unexpectedly, such as spawning lots of containers or using up all available disk space, those will create issues for the production environment.

This book cannot give you a precise number of clusters that will work perfectly in your situation; as previously mentioned, it makes sense to look at similar enterprise applications in your organization to see how many life cycle stages are being used.

Business continuity, disaster recovery, and failover

Alongside the standard staging environments, many organizations will also look to create at least a failover production environment. A failover production environment is used in the event of a catastrophic failure that brings down the entire cluster or Azure region. This is often called **Disaster Recovery (DR)**. Typically, this would be deployed to a different Azure region than the production cluster.

The managed cloud service comes with a **Service-Level Agreement (SLA)** of 99.95%, which, for many business-critical applications, is acceptable. However, for some applications, a superior SLA may be required. It is important to understand that it is not possible to go beyond this SLA with a single cluster. Consider if your application needs a service level greater than 99.95%, or if this is enough for your application.

If it's not, you can get to higher levels of service availability (such as 99.999%) by calculating a composite SLA from running multiple clusters in parallel. The clusters could all be within the same region (for example, westeurope) or across multiple regions (for example, westeurope and northeurope) for even higher levels of availability. The following Azure documentation describes how to calculate composite SLAs when running multiple clusters.

[Azure documentation on composite SLAs](#)

Multicenter and multiregion deployments of Azure Red Hat OpenShift are outside of the scope of this book. This is because when building these more complex architectures, there are several challenges associated with getting traffic into the cluster, and choosing how, and which, data to share between clusters needs quite a lot of consideration.

Regions and availability zones

Azure Red Hat OpenShift is designed to make use of three availability zones in each region it is deployed into. In Azure, an [availability zone](#) is an autonomous datacenter within a region, with its own power, cooling, and network connectivity. If you inspect a deployment of Azure Red Hat OpenShift, you will see a single control node (virtual machine) and application node in each availability zone.

Figure 4.1 shows how the control nodes and application (worker) nodes are distributed across three availability zones within a single Azure region:

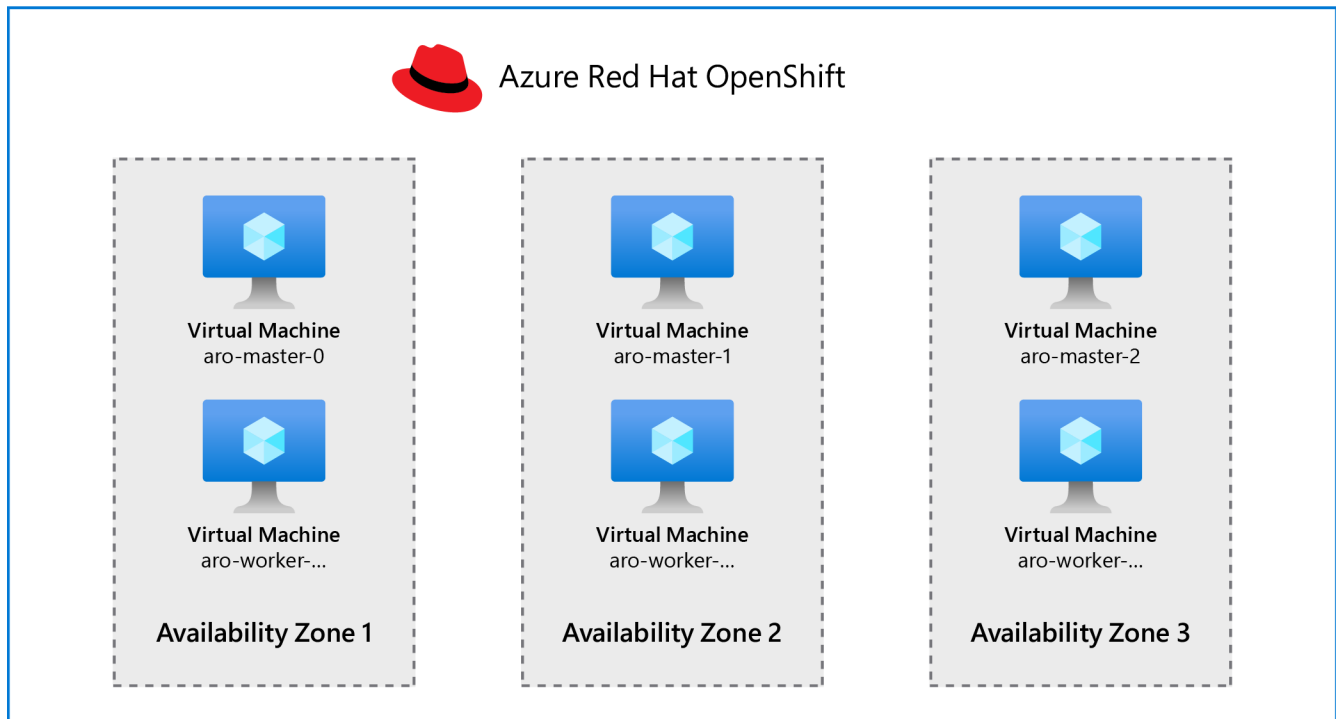


Figure 4.1: Distribution of control and application nodes across three availability zones within a single Azure region

Azure Red Hat OpenShift is designed to be offered as a fully managed, highly available service. Therefore, it is not possible to deploy fewer than three control nodes and three worker nodes.

Network concepts

Mentioned previously in the introduction to Azure Red Hat OpenShift was this excellent documentation page, on networking concepts, which can be found on the Microsoft documentation site:

- [Network concepts for Azure Red Hat OpenShift](#)

This page includes a detailed description of every networking component in Azure Red Hat OpenShift—the load balancers, the public and private IP addresses, the network security groups, and more.

A few key points to understand are:

- Azure Red Hat OpenShift deploys into an existing or new virtual network. Only a single virtual network is supported as well—but multiple networks would not bring any additional benefit, as OpenShift layers its own **Software-Defined Network (SDN)**, called OVS, on top.
- The minimum size of the master and application node subnets is /27.
- The default pod CIDR is 10.128.0.0/14.
- The default service CIDR is 172.30.0.0/16.
- Each node is allocated a /23 subnet (512 IP addresses) for its pods. This value cannot be changed.
- Egress IPs are not currently supported.
- It is possible to control egress traffic routing, specifically to send it via Azure Firewall. At the time of writing, this feature is in public preview and is documented here: [Control egress traffic](#).

Public or private network visibility

You may often hear that Azure Red Hat OpenShift can be deployed in both a public or a private deployment. While this is true, it's helpful to understand the difference between making the control plane public/private and making the applications on your cluster public/private.

For the API server visibility, it is a decision that you need to make at provisioning time; it is not possible to adjust public/private visibility after the cluster has been provisioned.

When you come to create an Azure Red Hat OpenShift cluster later in this chapter, you will run the `az aro create` command, which accepts arguments on the visibility of the cluster. The following example shows how to adjust visibility:

Both private

```
az aro create .... --apiserver-visibility Private --ingress-visibility Private
```

Private API server and public worker ingress

```
az aro create .... --apiserver-visibility Private --ingress-visibility Public
```

The visibility of the apiserver and application ingress map onto the Azure architecture diagram in *Figure 4.2*. This diagram shows how Azure Red Hat OpenShift uses internal and public Azure load balancers, where the API and the router are deployed depending on the visibility options selected.

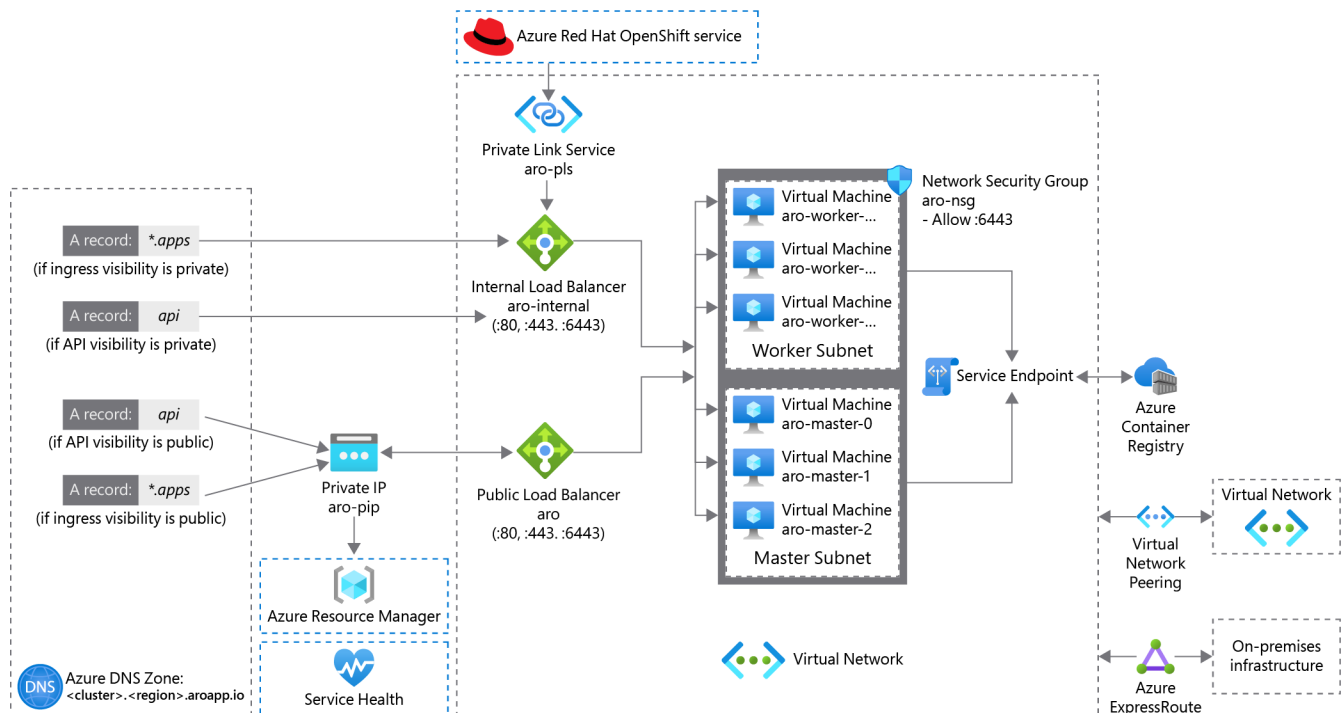


Figure 4.2: Azure Red Hat OpenShift using internal and public Azure load balancers

More detailed descriptions of the API server visibility and ingress visibility can be found as follows.

API server visibility (control plane)

--apiserver-visibility can be **public** or **private**:

- **Private** means that the Red Hat OpenShift control plane (historically called the "master nodes"), where the Kubernetes API runs, is not accessible from the public internet. This control plane is intended for developers and operators to control the cluster, and can be used to deploy, delete, or scale applications, as well as the cluster itself. Enterprises who have express route connections to Azure, or use a **Virtual Private Network (VPN)** to access compute resources, should choose private in most cases.
- **Public** means that the cluster control plane is accessible from the public internet. This exposes the cluster to risk of attack from anyone on the internet—even though access is authenticated. However, setting this to `public` is useful for lab or testing environments where you cannot control the source network of your users. For environments where real data is stored, or any kind of production environment, it is strongly recommended to set the API server visibility to `private`.

The following list provides some examples where API server connectivity is required—these should be taken into consideration when selecting public or private:

- Developers using scripts and tooling from their IDE (for example, `kubectl rollout`)
- Operators inspecting the status of their cluster (for example, `kubectl get nodes`)
- CI/CD servers that need to inspect or adjust the state of deployments (for example, Azure DevOps)
- Cluster security tools that connect to the cluster to examine the state
- Monitoring tools that rely on the Kubernetes API

In most cases, networking can be configured to come over **private** connections, but the preceding list may include additional examples from inside your organization—and you may find an unexpected requirement for **public** API server visibility.

Ingress visibility (applications)

--ingress-visibility can be public or private:

- **Private** means exposed services (which relate to applications running on the cluster) do not allow connections directly from the public internet. It is still possible to configure networking routing so that users first pass through Azure Firewall, or a web application firewall, optionally running in a separate Azure network before connecting to your applications. **Private** also works well if applications on the cluster are only intended to be used internally within your organization, for example, payroll processing, data analysis, or other internal web applications.
- **Public** means that the applications on your cluster are reachable from the public internet. It would still be necessary to configure an ingress or route resource to be able to access those applications, however. This would be the case if you were hosting a public e-commerce shopping website or other public application on Red Hat OpenShift.

Ingress is sometimes referred to as the OpenShift "router."

Recommendations for production

It is strongly recommended to:

- Access the cluster over a private connection—not via the public internet. Azure ExpressRoute is the best option when the cluster would need permanent connectivity from an on-premises network or corporate office. Otherwise, a VPN into Azure can also provide a private connection. More details on this can be found in the **Hybrid connectivity** section.
- Set the API server visibility (control plane) to private, and optionally further restrict access with firewalls.
- Set the ingress visibility (applications) to private for applications running within your organization. If there is a use case for applications on Azure Red Hat OpenShift that need to be hosted on the public internet, set up a separate Azure Red Hat OpenShift cluster—at least one cluster for internal apps and another for external apps with the ingress visibility set to public. However, mixing both public and private ingress within the same cluster is possible.

Naturally, most organizations will consult their Azure networking and security teams to determine any additional controls that may be necessary. For example, some organizations require web applications to be fronted by a web application firewall. This is also a common deployment pattern when deploying applications on Azure Red Hat OpenShift as well.

Hybrid connectivity

Most organizations that deploy Azure Red Hat OpenShift will run applications that need to connect back to supporting services running in on-premises environments. When connecting from Azure back to on-premises, this is commonly called hybrid connectivity, or a hybrid cloud architecture. There are a few ways to provide connectivity back to on-premises environments. The two most notable options are:

- **VPN**, which is suitable for low-complexity connectivity to Azure. Typically, VPNs are connected via Azure VPN Gateway. For more information, see [What is Azure VPN Gateway?](#)
- **Azure ExpressRoute circuits**, which are suitable for a dedicated, robust, permanent connection to Azure. For more information, see [What is Azure ExpressRoute?](#)

Irrespective of which connection method is used, both solutions can allow applications from on-premises to connect to applications running on Azure Red Hat OpenShift, and vice versa.

When connecting from an on-premises environment to Azure Red Hat OpenShift, it is common to connect via a hub virtual network. A diagram that explains how connectivity works with Azure ExpressRoute is shown in *Figure 4.3*:

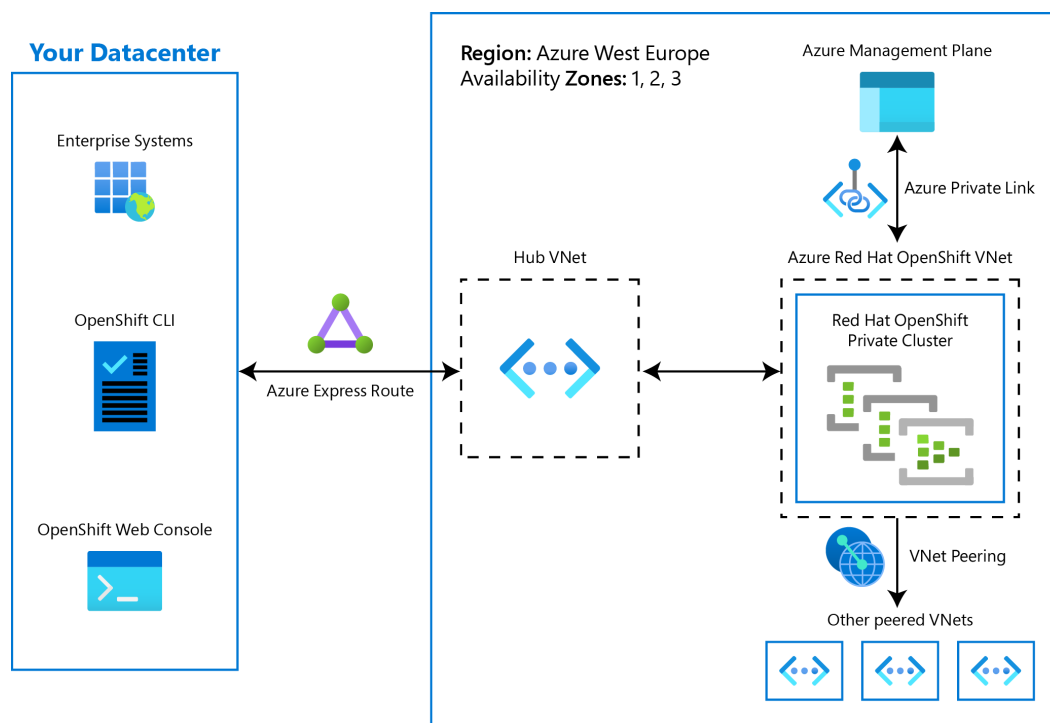


Figure 4.3: Connectivity with Azure ExpressRoute

The preceding diagram shows the high-level architecture of Azure ExpressRoute connecting to Azure Red Hat OpenShift. While this diagram shows Azure ExpressRoute, connectivity via a VPN is conceptually very similar.

Considerations for applications running in a hybrid architecture

When running applications on Azure Red Hat OpenShift that connect back to on-premises environments, there are a few considerations that application owners may encounter:

- **Connection latency:** While Azure ExpressRoute offers a relatively low-latency connection back to on-premises environments, VPN networks that travel over the public internet can be considerably higher latency. For some applications, such as web servers, where the connection is just transmitting HTTP traffic, this is unlikely to cause issues. However, if a server-side application running on that web server needs to connect to a database running on-premises, there could be a considerable impact on performance.
- **Cost of ingress/egress traffic:** Some connection types will charge for ingress and egress traffic—either via Azure or from the connection provider. It is a sensible precaution to measure costs in a testing environment first, and then project forward what those costs might be at peak load, just to make sure there are no surprises.
- **Failure scenario:** While Azure ExpressRoute connections are designed to be permanent, and robust, VPN connections are often subject to being interrupted or connected/disconnected frequently. Because they also run over the public internet, VPN connection latency and quality of service can vary quite frequently throughout the day. It's important to test the application performance under poor hybrid link conditions, as well as when the connection is running at optimum performance. Also, should the connection be broken for several hours, is the application running on Azure Red Hat OpenShift able to continue running in a degraded way, or is its availability dependent entirely on the hybrid connection?

Your organization may well require several other points to be checked off your own existing internal checklists, but the previous points should be enough to start considering how a hybrid architecture would work for you.

Summary

In this chapter, we discussed many of the normal pre-provisioning questions that you should ask and research prior to deploying Azure Red Hat OpenShift. The next chapter provides some useful pointers to resources when it comes to actually deploying the cluster.

Chapter 5

Provisioning an Azure Red Hat OpenShift cluster

Let's recap where we've got to so far. In this book you've covered the following:

- In *Chapter 2, Introduction to Red Hat OpenShift*, we briefly introduced you to Red Hat OpenShift and explained the advantages of choosing OpenShift over bare Kubernetes.
- In *Chapter 3, Azure Red Hat OpenShift*, we described the specifics of the Azure Red Hat OpenShift managed cloud service and covered key concepts, including its architecture, management, authentication, support, and pricing considerations.
- In *Chapter 4, Pre-provisioning – enterprise architecture questions*, we discussed the common questions that organizations should address before deploying Azure Red Hat OpenShift.

Now that we're ready to provision and deploy a cluster, you'll find the official deployment instructions on the documentation site (provisioning is part of deployment; to provision a deployment means making sure that the necessary IT infrastructure to support a deployment is in place).

[Tutorial – Create an Azure Red Hat OpenShift 4 cluster](#)

This chapter does not cover the individual commands that you need to type in order to create a cluster because the commands are subject to change over time, and the documentation already covers this in detail.

However, this chapter will provide guidance regarding the overall process, and what you need to consider when deploying a cluster.

Manual deployments – timing expectations

Some organizations need to understand how long it takes to provision a cluster. Let's explore this with a detailed description.

A high-level process of the deployment prerequisites is as follows:

- The az command line deployed on a system administrator's workstation
- The resource providers registered for use with your Azure subscription
- A Red Hat pull secret (optional)
- A domain for your cluster (optional)
- A virtual network, with two empty subnets, one for the control plane nodes and one for the application nodes

In terms of the time needed to set up these prerequisites, a skilled Azure and Red Hat OpenShift administrator could probably complete these tasks in 30 minutes the first time around, and in as little as 10 minutes if executing these steps repeatedly as part of a manual process.

Once the prerequisites have been met, the automated provisioning process generally takes between 25 and 40 minutes, depending on activity within the Azure region.

Deployment automation

With the understanding that the provisioning process of Azure Red Hat OpenShift is automated, it's typical that an organization will try to use tools to also automate the prerequisite steps—creating networks, subnets, service principles, and so on.

Many tools exist that can automate these steps. Some recommended tools are provided here:

- The az command-line tool: When automated, this tool is normally installed in a container, or similar, as part of a CI/CD process. Typical tools used here include Jenkins, Azure DevOps, or possibly Ansible. Note that the az command-line tool only needs to be deployed once, but additional clusters might need to set the Azure subscription ID to reflect different parts of the organization.
- The resource providers registered for use with your Azure subscription: As previously, this is part of the az command-line tool setup.
- A Red Hat pull secret (optional): Red Hat has a documented supported REST API for obtaining a pull secret, information on which can be found in this [article](#).
- A domain for your cluster (optional): This depends on how you create DNS records, but if using Azure DNS, then Terraform, Ansible, or other popular Azure automation tools could do this for you.
- A virtual network, with two empty subnets, one for the control plane (master) nodes and one for the application (worker) nodes: This would typically be automated by popular Azure automation tools—Terraform, Ansible, or similar tools could create this network and subnets for you.

With the prerequisite steps automated, it's realistic to get this time down from 30 or 10 minutes, as would be the case with a manual process, to just a minute or two. It's not possible to speed up the deployment of the cluster (that always takes between 25 and 40 minutes), but realistically, this could be a very quick end-to-end deployment process compared to on-premises.

Deployment automation has many advantages beyond just speeding up the process—customers using deployment automation get the advantage of building a robust, repeatable process that is easily logged and audited. It's also extremely common for customers to build self-service catalog items into their own portals, allowing teams to easily provision and deprovision Azure Red Hat OpenShift clusters with no interaction necessary from the cloud platform team.

Accessing the cluster

This section provides a simple reference for how to access your Azure Red Hat OpenShift cluster after it has been provisioned.

Via the web UI

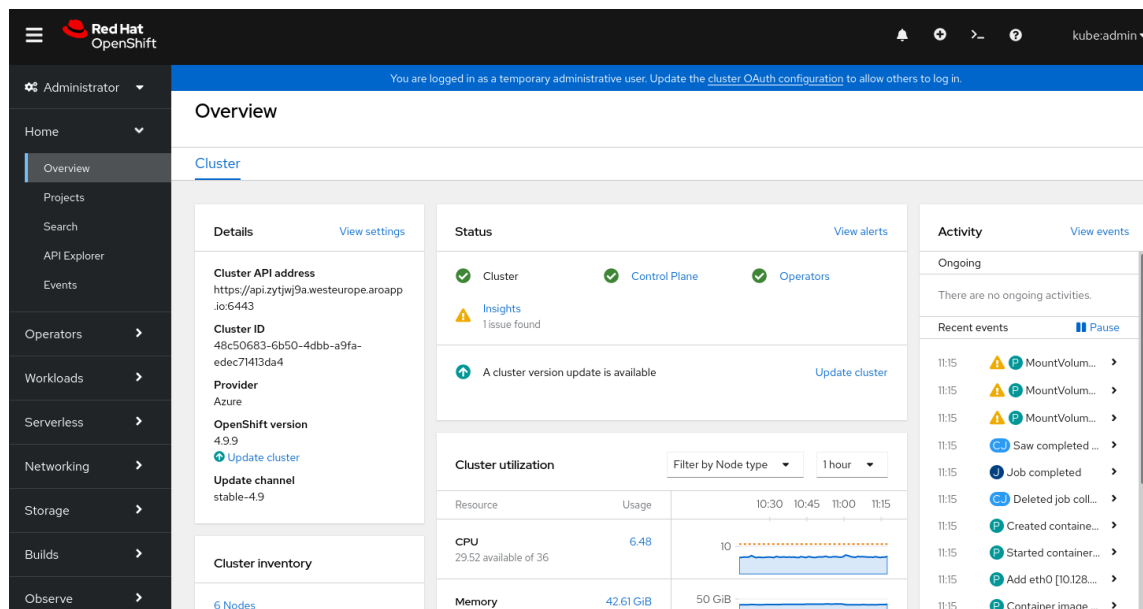
From a Bash shell, if you've installed the CLI, or from the Azure Cloud Shell (Bash) session in your Azure portal, retrieve your cluster sign-in URL by running the following command:

```
az aro show -n $CLUSTER_NAME -g $RG_NAME --query "consoleProfile.url" -o tsv
```

You should get back something like `openshift.xxxxxxxxxxxxxxxxxxxxxx.eastus.aroapp.io`. The sign-in URL for your cluster will be `https://` followed by the `consoleProfile.url` value; for example, `https://openshift.xxxxxxxxxxxxxxxxxxxxxx.eastus.aroapp.io`.

Open this URL in your browser. You'll be asked to log in with the `kubeadmin` user. Use the username and password that was provided to you by the installer during the installation process.

After logging in, you should be able to see the Azure Red Hat OpenShift web console.



The screenshot displays the Azure Red Hat OpenShift web console interface. The top navigation bar shows the Red Hat OpenShift logo and the user is logged in as 'kube:admin'. A blue banner indicates the user is logged in as a temporary administrative user and provides a link to update the OAuth configuration. The main content area is titled 'Overview' and is divided into several sections:

- Details:** Shows the Cluster API address (`https://api.zytjw9a.westeurope.aroapp.io:6443`), Cluster ID (`48c50683-6b50-4dbb-a9fa-edec71413da4`), Provider (Azure), OpenShift version (4.9.9), and Update channel (stable-4.9). There is a link to 'Update cluster'.
- Status:** Shows the overall cluster status as 'Cluster' (green checkmark), 'Control Plane' (green checkmark), and 'Operators' (green checkmark). There is a warning icon for 'Insights' (1 issue found) and a notification for 'A cluster version update is available' with a link to 'Update cluster'.
- Cluster utilization:** A table showing resource usage over time (10:30, 10:45, 11:00, 11:15). The CPU usage is 6.48 (29.52 available of 36) and Memory usage is 42.61 GiB (50 GiB total).
- Activity:** Shows a list of recent events, including 'MountVolum...', 'Saw completed...', 'Job completed', 'Deleted job coll...', 'Created containe...', 'Started container...', 'Add eth0 [10.128...', and 'Container image...'.

Figure 5.1: Azure Red Hat OpenShift web console

Take some time to explore the web console. Notice that it should be running a recent version of OpenShift, and all the components should be in a healthy status, or will soon be settling into a healthy status following installation.

Via the OpenShift CLI (oc)

You'll need to download the latest OpenShift CLI (oc). For this, simply log in to view the page at <https://console.redhat.com/openshift/downloads>.

Downloads

All categories ▾ > [Expand all](#)

Command-line interface (CLI) tools

Download command line tools to manage and work with OpenShift from your terminal.

Name	OS type	Architecture type	
> OpenShift command-line interface (oc)	Linux ▾	x86_64 ▾	Download
> OCM API command-line interface (ocm-cli) Developer Preview	Linux ▾	x86_64 ▾	Download
> Red Hat OpenShift Service on AWS (ROSA) command-line interface (rosa CLI)	Linux ▾	x86_64 ▾	Download

Figure 5.2: Downloading the OpenShift command-line interface

Extract the archive (.tar.gz or .zip) to somewhere on your system, and then put the oc command somewhere in your path. On Linux, it's quite normal to put the oc command in /usr/local/sbin/.

Running the OpenShift CLI and logging in to your cluster

To authenticate against your cluster from the command line, you will need to retrieve the login command and token from the web console. Log in to the web console with the browser, click on the username on the top right, and then click **Copy login command**.

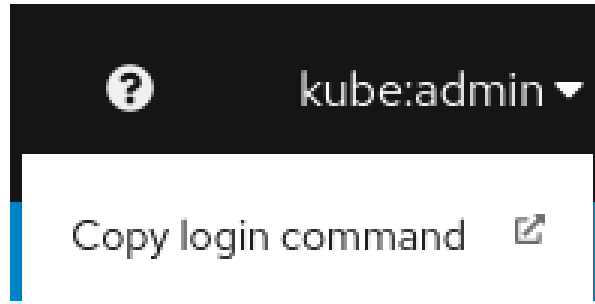


Figure 5.3: Copy login command

This will open a new page that looks something like this:

Your API token is

sha256-

Log in with this token

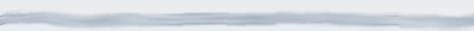

```
oc login --token= --server=https://api..westeurope.aroapp.io:6443
```

Figure 5.4: Log in with the API token

You can then copy this command and paste it into a terminal to log in to your Azure Red Hat OpenShift cluster.

For example, if you are using the Azure Cloud Shell (Bash) session in your Azure portal, paste that login command and afterward, the `oc status` command should look something like this:

```
user@Azure: oc status
In project default on server https://api.cyki1k6g.westeurope.aroapp.io:6443

svc/openshift - kubernetes.default.svc.cluster.local
svc/kubernetes - 172.30.0.1:443 -> 6443

View details with 'oc describe <resource>/<name>' or list everything with 'oc get all'.
```

You can take some time now to further explore the `oc` command line and get familiar with your new Azure Red Hat OpenShift environment. If you are an experienced OpenShift 4 user already, this cloud service of Azure Red Hat OpenShift should feel very familiar, and behave in exactly the same way as other OpenShift 4 environments you may have used in the past.

Summary

This was a very brief chapter, as the official provisioning instructions are well maintained, and should be considered the definitive guide to getting Azure Red Hat OpenShift running in your Azure subscription. You'll notice that the provisioning instructions are considerably simpler than the instructions required to provision a self-managed OpenShift environment. This is because a considerable amount of engineering has gone into the Azure Red Hat OpenShift service, providing tight integration into Azure, as well as deploying a prescriptive "one-size-fits-all" architecture that is well tested and well supported. Overall, this is a benefit for organizations as less time is spent in provisioning Azure Red Hat OpenShift, allowing more time to be spent on deploying applications.

Chapter 6

Post-provisioning – Day 2

After Azure Red Hat OpenShift has been deployed, there are a couple of post-provisioning activities that are typically needed before a cluster is made ready for production. This chapter covers many of the common post-provisioning activities, as follows:

- Authentication – Azure Active Directory – including how to synchronize user groups with a community operator
- Operators – including OperatorHub
- Understanding logging – including log forwarding
- Understanding monitoring – including monitoring OpenShift containers
- Upgrades and patches – including the supported version life cycle
- Cluster scaling – including scaling the cluster manually and automatically
- Application scaling – a brief note about application scaling
- Setting up a pull secret – registering to OpenShift Cluster Manager
- Limit ranges – including where a limit range can be applied
- Persistent storage – including the storage classes available and supported
- Security and compliance – a note about security controls

These sections, each detailing different post-provisioning tasks, will help grow your understanding of what it takes to take a freshly deployed cluster and make it ready for production applications.

Authentication – Azure Active Directory

Azure Red Hat OpenShift supports all of the authentication providers that are listed in the OpenShift documentation; check out the [full list of authentication providers](#). However, the majority of customers are likely to use Azure Active Directory, which is already available on Azure, to provide authentication and single sign-on to Azure Red Hat OpenShift.

Configuration for Azure Active Directory integration is deliberately a "Day 2" operation, as there may be cases where organizations want to use an alternative configuration provider. The setup and installation process for Azure Active Directory takes approximately 15 minutes the first time you are setting this up, and then, when you are more familiar with the process, you'll find you can run through this in as little as five minutes. Many organizations choose to automate parts of this provisioning using tools such as ARM templates or Ansible Playbooks.

- [Configure Azure Active Directory for Azure Red Hat OpenShift \(Graphical Portal\)](#)
- [Configure Azure Active Directory for Azure Red Hat OpenShift \(Command-Line Interface\)](#)

Using Active Directory user groups

The configuration of Azure Active Directory authentication will only allow users to log in with existing credentials. It does not bring the existing user groups for a user through to Red Hat OpenShift. It is quite common that an organization would want to import user groups from Active Directory so that they can be used to configure user permissions within OpenShift. A separate, community-supported operator is available to do this—the Group Sync Operator.

- [Group Sync Operator on GitHub](#)

Note that the group sync operator is community-supported, which means it is not officially supported by Red Hat or Microsoft at the time of writing. This book recommends following the detailed configuration and setup instructions for the operator that comes with the project's README file.

Operators

Operators in OpenShift 4 are one of the fundamental building components of the platform and provide a huge amount of value to consumers of Red Hat OpenShift. Operators are built from code and run a service in a container in the cluster. Some operators are responsible for maintaining things such as cluster networking, maintaining machine configuration, and cluster upgrades. These are often called cluster operators, and you can see a list of them in the **Administration** section of the OpenShift console.

Cluster Settings

Details ClusterOperators Global configuration







Name ↑	Status ↓	Version ↓	Message
 aro	✓ Available	-	-
 authentication	✓ Available	4.8.11	All is well
 baremetal	✓ Available	4.8.11	Operational
 cloud-credential	✓ Available	4.8.11	-
 cluster-autoscaler	✓ Available	4.8.11	at version 4.8.11
 config-operator	✓ Available	4.8.11	All is well

Figure 6.1: Cluster Settings

You can even see from the preceding screenshot that there is an operator just for Azure Red Hat OpenShift, which maintains parts of the service and tries to ensure the cluster stays in a supported configuration state.

Operators are capable of maintaining many things, like the health of a service, updates, patches, scaling, and several other functions besides.

OperatorHub

Beyond the standard cluster operators that run in the background on every cluster, administrators have access to many more operators via OperatorHub. OperatorHub makes it easy to find popular community and enterprise operators that an administrator might want to make available on the Red Hat OpenShift installation. OperatorHub can be found in the sidebar of every OpenShift cluster.

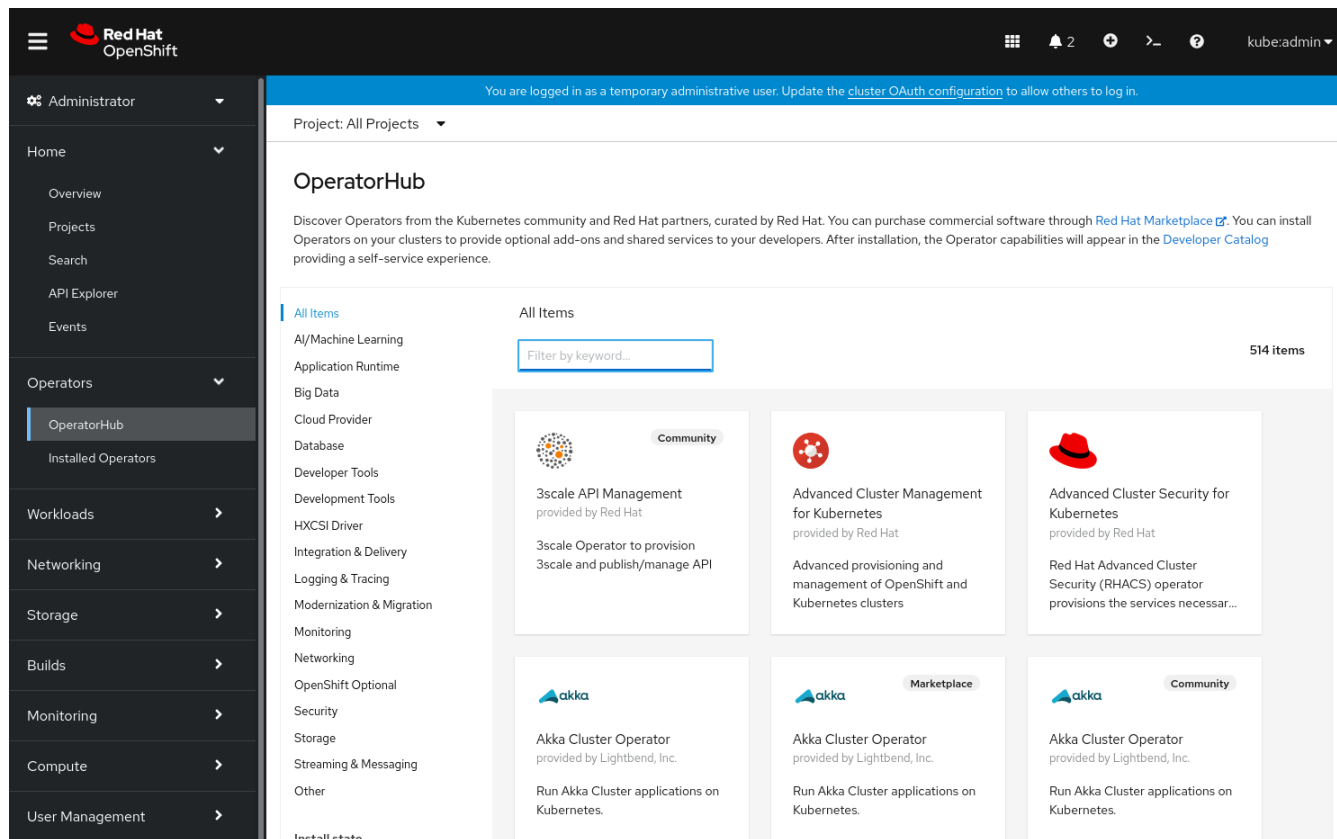


Figure 6.2: OperatorHub

Using operators, administrators can quickly and easily deploy and maintain popular software, without having to worry about Kubernetes configuration and code. Many Red Hat products are now packaged as operators, such as Advanced Cluster Management, Red Hat OpenShiftService Mesh, and Red Hat OpenShift Pipelines. However, there is also a huge library of operators for non-Red Hat software, such as the MariaDB database, Couchbase, or IBM block storage drivers.

For more information on operators, the following links are recommended:

- [Operatorhub.io](https://operatorhub.io)
- [Operators in the OpenShift](#)

Understanding logging

Azure Red Hat OpenShift uses the same logging and monitoring architecture as Red Hat OpenShift. Both offerings use the same operators for logging.

Logs are split into three categories:

- **Application:** Container logs generated by user applications running in the cluster, except infrastructure container applications.
- **Infrastructure:** Logs generated by infrastructure components running in the cluster and OpenShift Container Platform nodes, such as journal logs. Infrastructure components are pods that run in the openshift*, kube*, or default projects.
- **Audit:** Logs generated by auditd, the node audit system, which are stored in the `/var/log/audit/audit.log` file, and the audit logs from the Kubernetes API server and the OpenShift API server.

For a more detailed description of OpenShift logging, see [Understanding Red Hat OpenShift Logging](#) in the product documentation.

Using cluster log forwarding to Azure Monitor

A common integration is sending Azure Red Hat OpenShift logs to Azure Monitor's Container insights service. Sometimes, this is called Azure Log Analytics. This allows for persistent, low-cost log retention in Azure.

The OpenShift logging architecture runs Fluent Bit on each node, and the first approach an operator might take would be to adjust the configuration of this Fluent Bit service to send logs directly. However, adjusting the logging configuration in Azure Red Hat OpenShift is outside of the support policy. OpenShift has a built-in mechanism for forwarding logs while still remaining supported, and that is called `ClusterLogForwarder`.

Using `ClusterLogForwarder`, it is possible to forward logs to Elasticsearch, Flutentd, and syslog; however, Azure Monitor does not directly support any of these protocols. A workaround for this is to have an additional intermediate Fluent Bit instance, which receives logs from OpenShift and forwards them on to Azure Monitor. This approach is currently described in the [Microsoft documentation](#), and also in the [community documentation](#).

Understanding monitoring

As a managed service, it should not be necessary to implement complex custom monitoring for Azure Red Hat OpenShift as this is provided as part of the service that you pay for.

However, it's very common to want to monitor OpenShift containers using Azure Container insights. This is a feature that is currently in public preview, and is described in this [documentation](#).

Upgrades and patches

When you provision an Azure Red Hat OpenShift cluster, an update channel will not be selected. This means that your cluster will not start receiving updates by default.

To see your update channel, navigate to **Administration** → **Cluster Settings** from the navigation sidebar. Here's a view of the cluster settings soon after an Azure Red Hat OpenShift cluster has been provisioned:

Cluster Settings

[Details](#) ClusterOperators Global configuration


Last completed version	Update status	Channel
4.7.21	No update channel selected	- 

Figure 6.3: Cluster Settings after a cluster has been provisioned

To select an update channel, select the "-" link, and see the available options:

Update channel

Select a channel that reflects your desired version. Critical security updates will be delivered to any vulnerable channels.

[Learn more about OpenShift update channels](#)

Select channel

Select channel ▼

stable-4.7

fast-4.7

candidate-4.7

Figure 6.4: Selecting an update channel

To understand the differences between stable, fast, and candidate, see the [Upgrade channels and releases](#) documentation page.

It is strongly recommended that all clusters in production should run a **stable** channel.

Supported version life cycle

Knowing which versions of Azure Red Hat OpenShift are supported is important in planning your production deployment. On the formal life cycle page, there is information to help organizations understand which versions are supported and which are not.

[Azure Red Hat OpenShift support life cycle page](#)

An extracted simplification of the information contained on that page is as follows:

Azure Red Hat OpenShift supports two **Generally Available (GA)** minor versions of Red Hat OpenShift Container Platform:

- The latest GA minor version that is released in Azure Red Hat OpenShift (which we will refer to as N)
- One previous minor version (N-1)

Red Hat OpenShift Container Platform uses semantic versioning. Semantic versioning uses different levels of version numbers to specify different levels of versioning. The following table illustrates the different parts of a semantic version number, in this case, using the example version number 4.9.3:

Major version (x)	Minor version (y)	Patch (z)
4	9	3

Each number in the version indicates general compatibility with the previous version:

- **Major version:** No major version releases are planned at this time. Major versions change when incompatible API changes or backward compatibility may be broken.
- **Minor version:** Released approximately every three months. Minor version upgrades can include feature additions, enhancements, deprecations, removals, bug fixes, and security enhancements.
- **Patches:** Typically released each week, or as needed. Patch version upgrades can include bug fixes and security enhancements.

For a fuller understanding of the supported versions, read through the [Azure Red Hat OpenShift Support Life cycle page](#).

Cluster scaling

Red Hat OpenShift Container Platform and, by extension, Azure Red Hat OpenShift, is built with a scalable architecture in mind. When planning for scale in the context of OpenShift, generally administrators and application owners need to consider cluster scaling and application scaling as two distinct topics.

This section covers cluster scaling, and there is a brief note covering application scaling as a separate section.

Supported maximums

Cluster scaling refers to adding additional worker nodes to the cluster, which in turn provides additional compute capacity for applications running in the cluster. Naturally, the question arises about when it's necessary to scale the control plane, too. Azure Red Hat OpenShift already deploys three control plane nodes, which, in principle, come with enough capacity to scale to the maximum number of supported worker nodes in an Azure Red Hat OpenShift cluster—currently 60.

At the time of writing, there are three instance sizes of control plane nodes supported: Standard_D8s_v3, Standard_D16s_v3, and Standard_D16s_v3.

There are more Azure instance types supported for the worker nodes—Compute-optimized (F series), Memory-optimized (E series), and General-Purpose (D series).

A list of currently supported Azure instance types for Azure Red Hat OpenShift can be found on the [support policy page](#).

Minimum deployment and scale to zero

Occasionally, organizations might want to deploy smaller clusters for the purpose of test and development, or other use cases where reduced availability would be acceptable. At present, the minimum cluster size is three control plane nodes and three application nodes, and scaling smaller than that is not supported.

Manually scaling the cluster

Operators who look at the Azure portal might be tempted to try and add additional worker nodes to the Azure Red Hat OpenShift cluster by directly creating an Azure virtual machine. However, this is not possible, as the resource group that contains Azure Red Hat OpenShift is "locked" from an Azure administrator's perspective. This is the case regardless of permissions—even users with the **owner** permission cannot modify the contents of an Azure Red Hat OpenShift resource group. Therefore, you will experience permission denied errors if you attempt to create a virtual machine manually within the Azure Red Hat OpenShift resource group.

The intended mechanic for Azure Red Hat OpenShift scaling is to use the OpenShift MachineSets feature, whereby OpenShift will deploy a new application node when instructed. Users with cluster-admin privileges can see **MachineSets** under **Compute**.

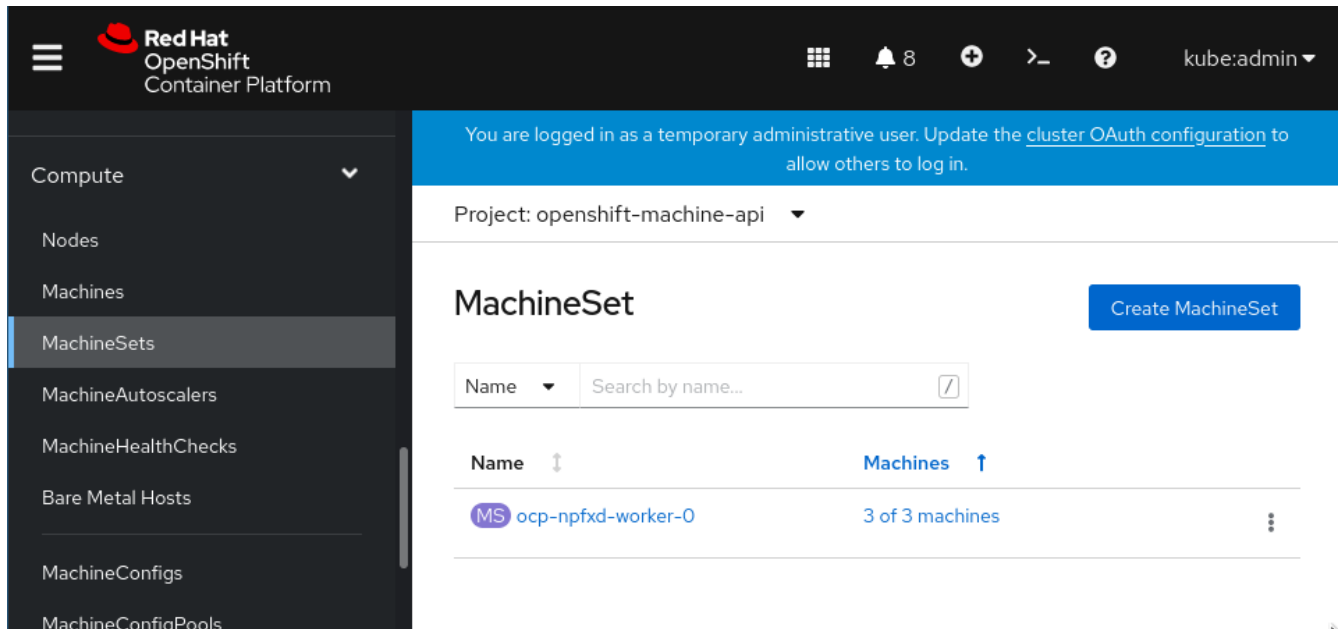


Figure 6.5: Admin access to MachineSets

Clicking on the "edit" menu of a MachineSet application node, you'll see you can simply provision a new application node virtual machine by selecting **Edit Machine Count**.

Edit Machine count

MachineSets maintain the proper number of healthy machines.

Cancel

Save

Figure 6.6: Editing the machine count

Once the count has been updated, administrators can either go to the Azure portal or the **Compute** → **Machines** view to see a new application node virtual machine being provisioned.

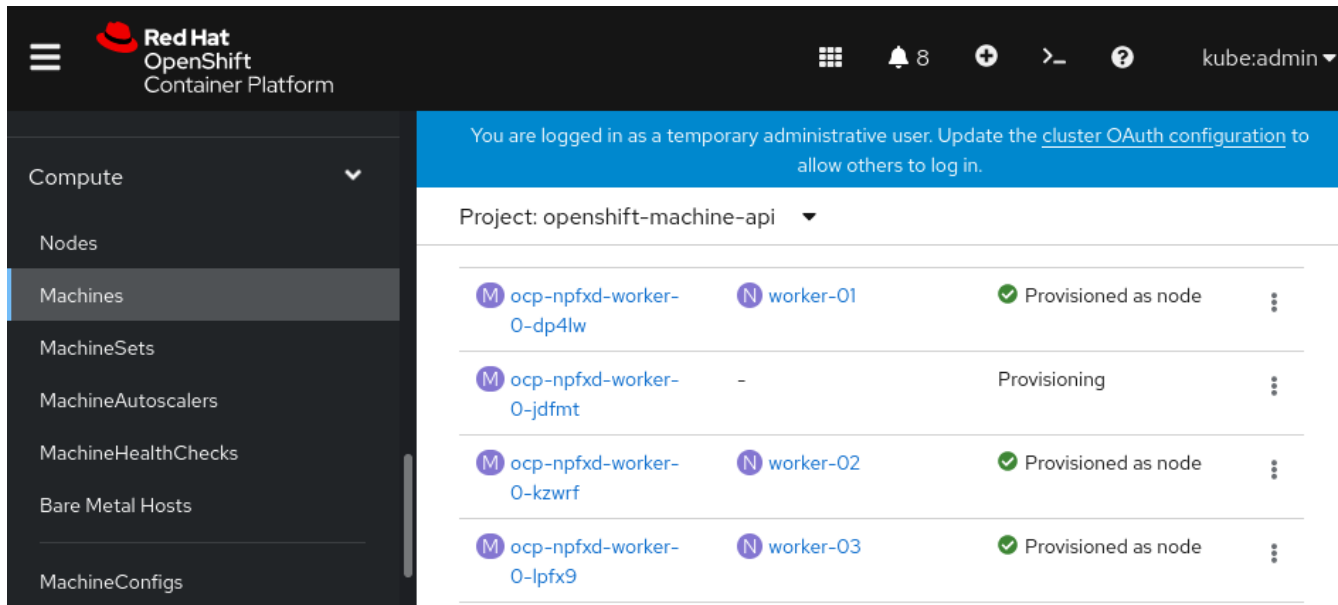


Figure 6.7: The machines view

It typically takes up to about five minutes to provision an additional virtual machine in most Azure regions.

Administrators do not need to do any post-provisioning activities to bring this new capacity online. OpenShift will automatically make it available to the cluster, and applications will make use of it when needed.

Cluster autoscaling

The default deployment of Azure Red Hat OpenShift is not configured with the autoscaling capability turned on, but enabling this functionality is straightforward. Administrators simply need to create a `MachineAutoscaler` resource, which can be found under the Azure Red Hat OpenShift **Compute** sidebar menu.

MachineAutoscaler resources operate on a MachineSet, which in turn will create or delete application node virtual machine capacity as needed. The following example shows that it is possible to maintain a minimum or a maximum number of machines in a MachineSet:

```
apiVersion: autoscaling.openshift.io/v1beta1
kind: MachineAutoscaler
metadata:
  name: worker-us-east-1a
  namespace: openshift-machine-api
spec:
  minReplicas: 1
  maxReplicas: 12
  scaleTargetRef:
    apiVersion: machine.openshift.io/v1beta1
    kind: MachineSet
    name: worker
```

OpenShift also has the concept of cluster autoscalers, which makes it possible to scale based on keeping a certain amount of RAM, CPU, or similar available. Choosing a MachineAutoscaler or ClusterAutoscaler depends on how you want to add and remove capacity in your cluster.

[Red Hat OpenShift documentation on MachineAutoscalers and ClusterAutoscalers](#)

Application scaling

Scaling microservice applications is a complex topic that is outside the scope of this book. However, here are two pointers to useful pages to get started in exploring this topic:

- [HorizontalPodAutoscaler](#): specify the minimum and maximum number of pods you want to run, as well as the CPU utilization or memory utilization your pods should target.
- [Serverless and scale-to-zero with Knative](#).

The cluster will monitor the running containers and remaining capacity on the cluster. Should Knative or HorizontalPodAutoscaler request more resources than is currently free in the cluster, then a ClusterAutoscaler or MachineSet scale would need to happen to add the requested resources to the cluster.

Using this approach, applications and the cluster itself can scale in tandem, both up and down, according to demand.

Setting up a pull secret (register to OpenShift Cluster Manager)

A fresh deployment of Azure Red Hat OpenShift does not have a "pull secret" configured for `ccloud.redhat.com`. This means that your clusters will not show up in the Red Hat Hybrid cloud console (<http://console.redhat.com>) by default—this is called OpenShift Cluster Manager.

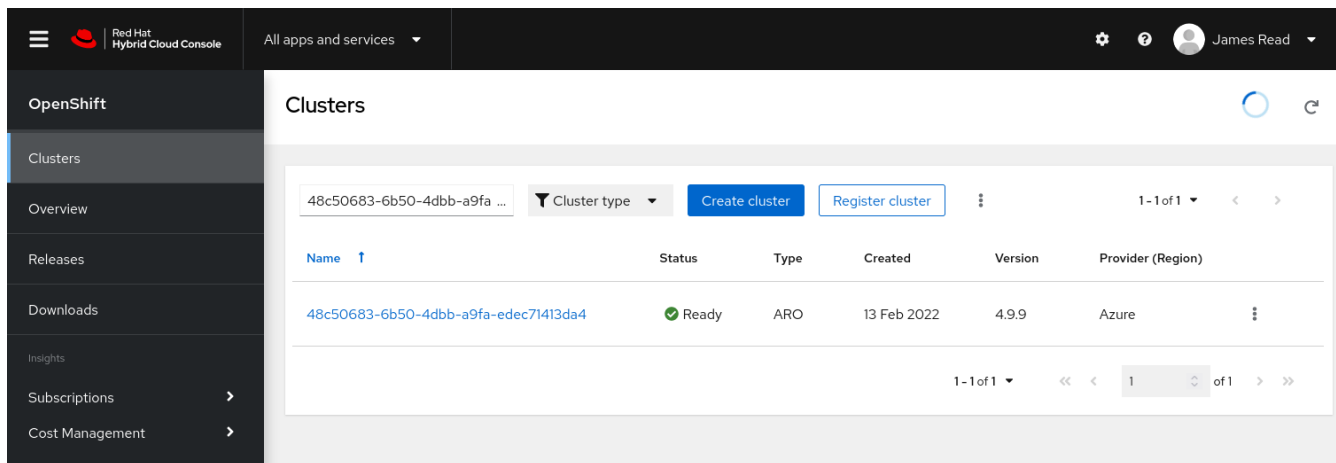


Figure 6.8: OpenShift Cluster Manager showing an Azure Red Hat OpenShift cluster

Configuring a pull secret for `ccloud.redhat.com` is very straightforward, and as well as showing up in the OpenShift Cluster Manager portal at <http://console.redhat.com>, you will also then be able to raise support tickets directly with Red Hat via the standard support ticketing system.

The instructions for how to set up a pull secret are here:

- [How to add or update a pull secret](#)

An additional benefit to setting up a pull secret is that customers are then able to raise support tickets directly with Red Hat. The pull secret grants an entitlement on your Red Hat account that allows your cluster to be seen by support staff. Opening a support ticket for Azure Red Hat OpenShift is then the same as any other Red Hat product.

The screenshot displays the 'Customer support' interface. At the top, there are three tabs: 'Cases', 'Troubleshoot', and 'Manage'. Below the tabs is a vertical sidebar with a numbered list of steps: 1. Create a case, 2. Select a product (highlighted in blue), 3. Describe your issue, 4. Case information, 5. Case management, 6. Review, and 7. Submit. The main content area shows two dropdown menus for 'Product' and 'Version', both set to 'OpenShift Managed (Azure)'. Below these menus is a text prompt: 'Have an account, billing, or subscription issue? [Contact customer service](#) for help.'

Figure 6.9: Creating a support case

Limit ranges

When a development or application team starts deploying containerized applications, it will only be a certain amount of time before one application "misbehaves" and starts consuming unnecessary resources on the cluster. An example might be a faulty application with a memory leak or an application that has mistakenly been set to scale after consuming just 10% CPU instead of 100%. To prevent scenarios where these misbehaving applications scale out of control and consume too many resources, using `LimitRange` is recommended.

`LimitRange` allows you to restrict the resource consumption for specific objects in a project. `LimitRange` can be applied to:

- Pods and containers: You can set minimum and maximum requirements for CPU and memory for pods and their containers.
- Image streams: You can set limits on the number of images and tags in an `ImageStream` object.
- Images: You can limit the size of images that can be pushed to an internal registry.
- **Persistent Volume Claims (PVCs):** You can restrict the size of the PVCs that can be requested.

An example of a limit range that applies to containers, limiting the minimum, maximum, and default CPU and memory requests allowed, is shown here:

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits"
spec:
  limits:
    - type: "Container"
      max:
        cpu: "2"
        memory: "1Gi"
      min:
        cpu: "100m"
        memory: "4Mi"
      default:
        cpu: "300m"
        memory: "200Mi"
      defaultRequest:
        cpu: "200m"
        memory: "100Mi"
      maxLimitRequestRatio:
        cpu: "10"
```

This LimitRange code snippet can be applied by copying and pasting the YAML directly into the editor of the Red Hat OpenShift console, or from a file with `oc apply -f limitrange.yaml`.

[Limit ranges documentation](#)

Persistent storage

The virtual machines deployed by Azure Red Hat OpenShift come with Azure disks attached, for the purpose of installing Red Hat CoreOS, and running the Azure Red Hat OpenShift service. These disks should not be used by applications, but only for the OpenShift cluster itself.

Applications that require persistent storage should use the Kubernetes PersistentVolume feature, and there is an excellent page in the OpenShift documentation called [Understanding persistent storage](#) that describes this concept in detail. While Azure Red Hat OpenShift technically supports all of the PersistentVolume providers as a self-managed OpenShift installation, on Azure, the most commonly used persistent storage is as follows:

Name	Type	Access Modes	Storage Class
Azure Files	Filesystem, non-POSIX-compliant	ReadWriteOnce	Azure File
Azure Disk	Block	ReadWriteOnce	Azure Disk
Red Hat OpenShift Data Foundation	Filesystem, Block, Object	(Several)	OCS/ODF docs

Security and compliance

The Red Hat OpenShift documentation now includes a considerable amount of content relevant to understanding how to implement many security controls and maintain compliance.

[OpenShift security and compliance documentation](#)

This section of the documentation includes:

- A detailed description of how container security works in OpenShift. It's important to understand that OpenShift offers many out-of-the-box security controls for containers that are not found in other Kubernetes-based services, and these controls help keep your organization and applications safe.
- Pod scanning for vulnerabilities
- Audit log access
- Configuring certificates

This also includes several helpful security-related operators, such as:

- **Compliance Operator:** run scans and provide recommendations for remediation for common security issues.
- **File integrity checking:** continually check that files, especially sensitive security configuration files, have not changed.

Summary

In this chapter, we have covered most of the tasks and topics that organizations typically consider in making a freshly deployed cluster suitable for production applications. While it won't be necessary to run through each of these topics with every subsequent deployment, the first cluster you deploy should consider persistent storage, limits, authentication, and the various other topics mentioned in this chapter.

Typically, an organization will try to automate post-provisioning tasks as much as possible. For example, the setup and configuration of Azure Active Directory can mostly be achieved using either a PowerShell script or a few ARM templates. This will help cut down on time spent on repetitive tasks if you are deploying many clusters.

The next chapter in this book moves on to deploy an example application on top of your production-ready Azure Red Hat OpenShift cluster.

Chapter 7

Deploy a sample application

The content of this guide is mainly aimed at helping technical audiences—those in developer and operations roles—who are looking to understand what is needed to get into production with Azure Red Hat OpenShift. This guide assumes the reader has a basic understanding of Red Hat OpenShift, as much of the architecture, management portals, and user experience is identical to Azure Red Hat OpenShift.

This chapter serves as a quick primer to explain how to deploy a simple sample application called the "Fruit Smoothies" application. This application should serve as a quick start and a reminder so that you can better understand how to use Azure Red Hat OpenShift. Note that this is a sample application maintained for Azure Kubernetes Service and deploying it on top of Azure Red Hat OpenShift is done to provide proof that OpenShift is 100% Kubernetes compatible.

This chapter is largely based on content from <http://aroworkshop.io>, and extra descriptions and context have been added.

An overview of the ratings application

The application architecture is simple, as follows:

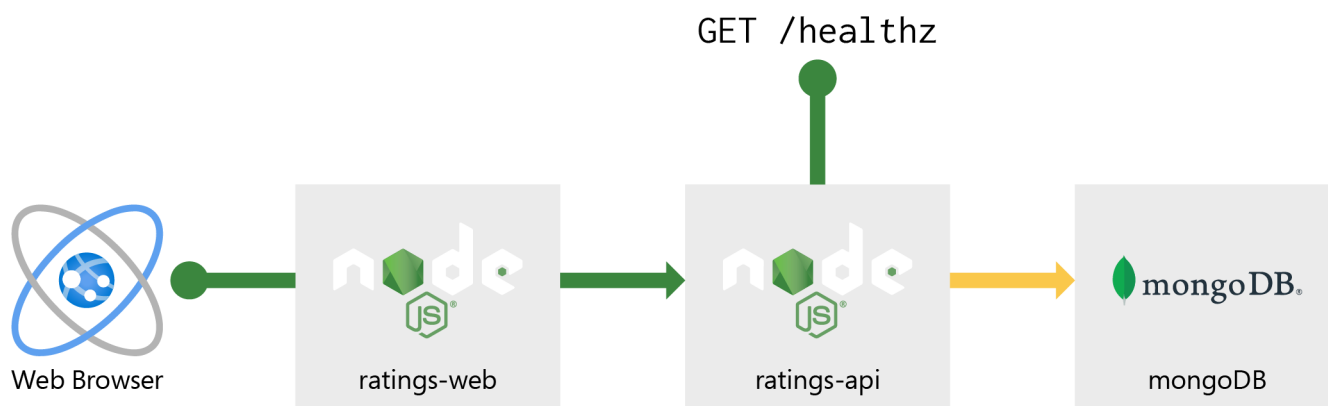


Figure 7.1: Architecture of the Fruit Smoothies application

In the preceding diagram, you can see that the application consists of three services and two public endpoints, as shown in the table below. The first endpoint on the left-hand side of the diagram represents a user's web browser, viewing the public HTML web application. The second endpoint—healthz—is a health check in the ratings-api service. Descriptions of the individual services and links to their repositories can be found in the following table:

Component	Description	GitHub repository
rating-web	A public-facing web front end—the "website."	GitHub repo
rating-api	This service takes input from the web UI and stores it in the database. It also serves results from the database back to the web application on port 3000.	GitHub repo
mongodb	A NoSQL database with preloaded data.	Data

You can visit the GitHub repository links to further explore and understand these applications. In the instructions that follow, there will be step-by-step guidance on how to deploy each of these applications.

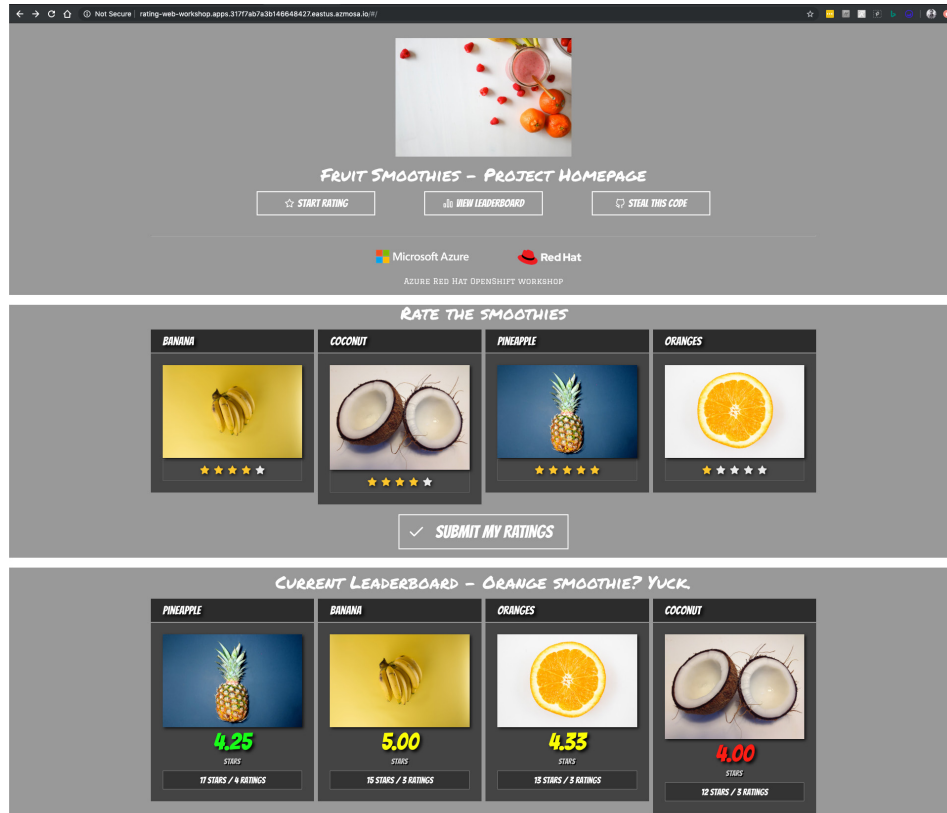


Figure 7.2: Screenshots showing an overview of the Fruit Smoothies application

Once you're done, you'll have a web application up and running that looks similar to the preceding screenshots. You should also have a better understanding of how developers and operations teams deploy applications to Azure Red Hat OpenShift, which should help with your own decision-making process when deploying applications to Azure Red Hat OpenShift yourself.

Create and connect to the cluster

The rest of this chapter assumes that you have a working Azure Red Hat OpenShift environment with which to work. There are no special requirements for this rating application, and it will deploy onto a blank, fresh Azure Red Hat OpenShift environment. If you have not yet provisioned a cluster, review the following chapters:

- *Chapter 4, Pre-provisioning – enterprise architecture questions*
- *Chapter 5, Provisioning an Azure Red Hat OpenShift cluster*

The *Accessing the cluster* section in *Chapter 5, Provisioning an Azure Red Hat OpenShift cluster*, is a helpful reminder of how to access a cluster that you may have provisioned previously.

Sign in to the web console

Each Azure Red Hat OpenShift cluster has a DNS address for the OpenShift web console. You can use the command `az aro list` to list the clusters in your current Azure subscription:

```
az aro list -o table
```

The cluster web console's URL will be listed. Open that link in a new browser tab and sign in with the `kubeadmin` user, or another user account that has permission to create projects.

After logging in, you should be able to see the Azure Red Hat OpenShift web console.

The screenshot shows the Azure Red Hat OpenShift web console interface. The top navigation bar includes the Red Hat OpenShift logo, a user profile dropdown for 'kube:admin', and a notification banner stating 'You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.' The main content area is titled 'Overview' and 'Cluster'. It is divided into three main sections: Details, Status, and Activity.

Details: Shows cluster information such as Cluster API address, Cluster ID, Provider (Azure), OpenShift version (4.9.9), and Update channel (stable-4.9). A link to 'Update cluster' is provided.

Status: Displays the overall health of the cluster with green checkmarks for Cluster, Control Plane, and Operators. It also shows an 'Insights' section with 1 issue found and a notification that a cluster version update is available, with a link to 'Update cluster'.

Cluster utilization: A table and chart showing resource usage. The table lists CPU usage (29.52 available of 36) and Memory usage (42.61 GiB). The chart shows usage over time (10:30 to 11:15).

Activity: Shows a list of recent events, including 'MountVolum...', 'Saw completed...', 'Job completed', 'Deleted job coll...', 'Created containe...', 'Started containe...', 'Add eth0 [10.128...', and 'Container image ...'.

Figure 7.3: Azure Red Hat OpenShift web console

Install the OpenShift client

Open [Azure Cloud Shell](#), or use your local Linux Terminal, and install the OpenShift command-line client. This is necessary to access the cluster from the command line. The instructions to do this are as follows:

```
cd ~
curl https://mirror.openshift.com/pub/openshift-v4/clients/ocp/latest/openshift-client-linux.tar.gz >
openshift-client-linux.tar.gz

mkdir openshift

tar -zxvf openshift-client-linux.tar.gz -C openshift

echo 'export PATH=$PATH:~/openshift' >> ~/.bashrc && source ~/.bashrc
```

Alternatively, for Windows or Mac, the download links are as follows:

- <https://mirror.openshift.com/pub/openshift-v4/clients/ocp/latest/openshift-client-windows.zip>
- <https://mirror.openshift.com/pub/openshift-v4/clients/ocp/latest/openshift-client-mac.tar.gz>

You should be able to run the `oc` command from either of those downloaded packages.

Retrieve the login command and token

After the client is installed, it is necessary to get a token to log in to the cluster. Log in to the OpenShift web console, click on the username in the top right, and then click **Copy Login Command**.

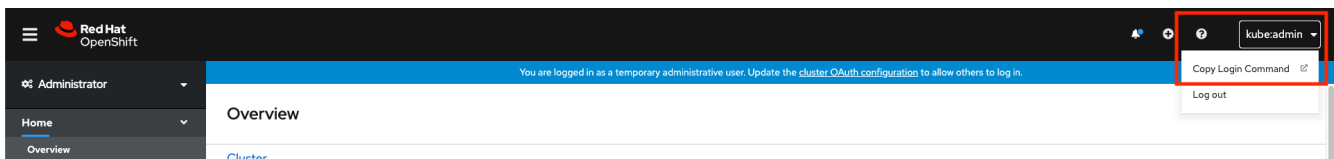


Figure 7.4: Copy Login Command to log in to the cluster

Paste the login command into your shell (local Linux Terminal or Azure Cloud Shell). You should be able to connect to the cluster.

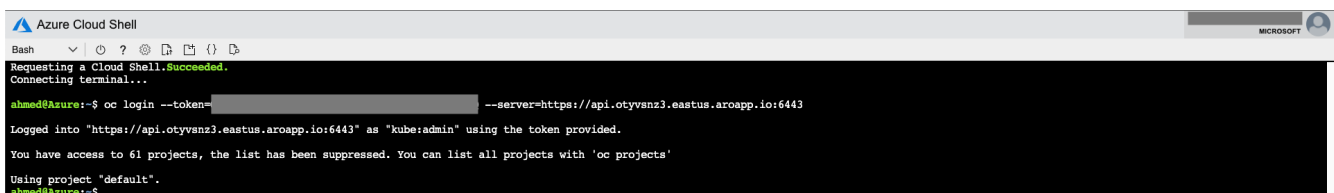


Figure 7.5: Use the login command to connect to the cluster

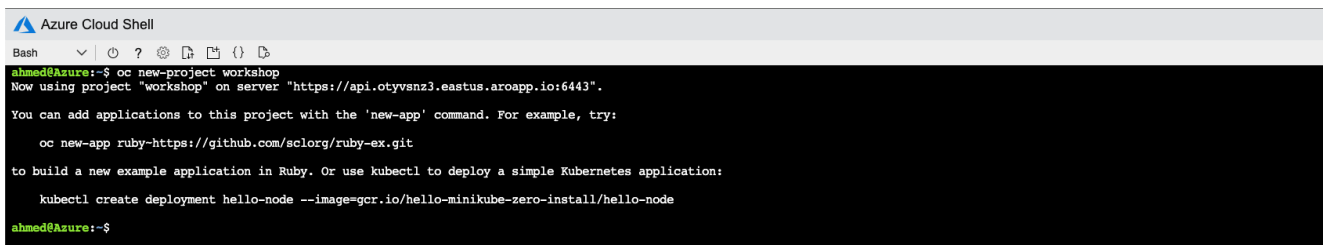
Once connected, we can go on to create a project.

Create a project

A project in OpenShift is like a logical folder to contain the ratings application. In Red Hat OpenShift, all containers and applications need to be within a project of some kind. You can use projects to separate out applications, or even departments. The next few instructions explain how to create a project.

While you can create a project from the web interface, these instructions use the command line:

```
oc new-project workshop
```



```
Azure Cloud Shell
Bash
ahmed@Azure:~$ oc new-project workshop
Now using project "workshop" on server "https://api.otyvsnz3.eastus.aroapp.io:6443".
You can add applications to this project with the 'new-app' command. For example, try:
  oc new-app ruby-https://github.com/sclorg/ruby-ex.git
to build a new example application in Ruby. Or use kubectl to deploy a simple Kubernetes application:
  kubectl create deployment hello-node --image=gcr.io/hello-minikube-zero-install/hello-node
ahmed@Azure:~$
```

Figure 7.6: Create a new workshop in Azure Cloud Shell

Once the project has been created, you can switch to it using `oc project workshop`. The next step will be to deploy the first of the three microservices that were introduced at the start of this chapter—MongoDB. It will be deployed into the project we just created.

Resources

- [Azure Red Hat OpenShift documentation – Getting started with the CLI](#)
- [Azure Red Hat OpenShift documentation – Projects](#)

Deploy MongoDB

Azure Red Hat OpenShift provides a container image and template to make creating a new MongoDB database service easy. The template provides parameter fields to define all the mandatory environment variables (user, password, database name, and so on) with predefined defaults, including the autogeneration of password values. It will also define both a deployment configuration and a service.

The MongoDB instance will be deployed directly as a container image from Docker Hub. OpenShift allows you to do this entirely via the web console. Switch to the developer perspective at the top of the menu, navigate to the **Add** page, and select **Container images**.

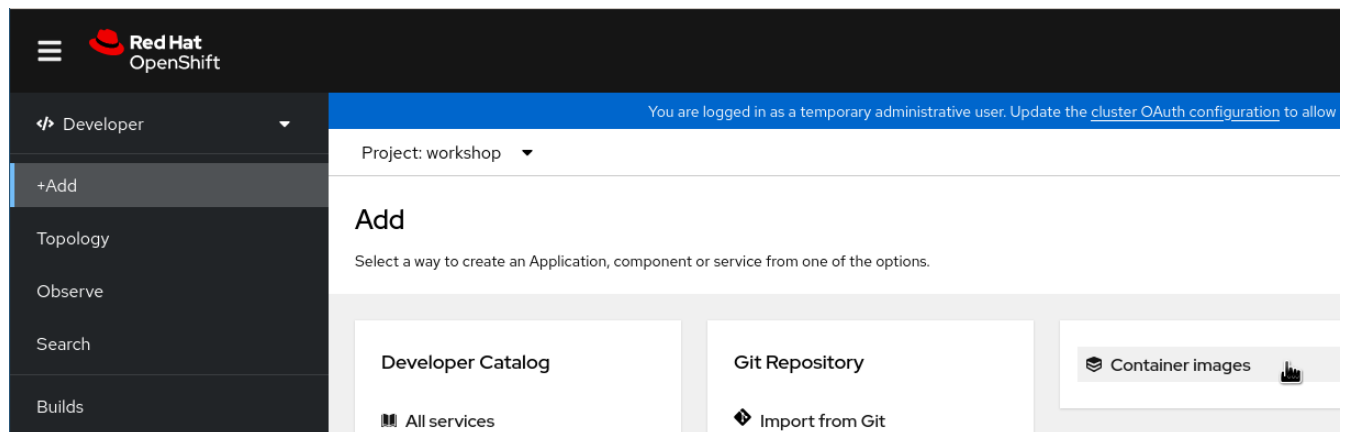


Figure 7.7: Adding a container image

This will take you to the **Deploy Image** page. Fill out the form as follows:

The screenshot shows the Red Hat OpenShift console interface. On the left is a dark sidebar with a menu including 'Developer', '+Add', 'Topology', 'Observe', 'Search', 'Builds', 'Helm', 'Project', 'ConfigMaps', and 'Secrets'. The main content area is titled 'Deploy Image' and contains the following elements:

- A blue header bar with the text: "You are logged in as a temporary administrative user. Update the g..."
- Breadcrumbs: "Project: mongodb-community-operator" and "Application: all applications".
- Section title: "Deploy Image"
- Section title: "Image"
- Text: "Deploy an existing Image from an Image Stream or Image registry."
- Radio button selection: "Image name from external registry" is selected.
- Input field: Contains "docker.io/mongo" with a green checkmark icon on the right.
- Text: "Validated"
- Text: "To deploy an Image from a private repository, you must [create an Image pull secret](#) with your Image registry credentials."
- Checkbox: "Allow Images from insecure registries" is unchecked.
- Radio button selection: "Image stream tag from internal registry" is unselected.
- Section title: "Runtime icon" (partially visible)

Figure 7.8: Fill out the form to deploy an image

Make sure to set the values in the form as follows:

Field	Value
Image name from external registry	docker.io/mongo
Runtime icon	mongodb
Application name	ratings
Name	mongodb
Create route to application	Unticked—a route would allow access to the database externally, which is not required for this application
Resource type	Deployment

When you get to the bottom of the form, select the Deployment link to expand the form, making it possible to set environment variables.

The following environment variable acts as a default to initialize the database when it first starts up:

Name	Value
MONGO_INITDB_DATABASE	ratingsdb

No username and password are being set for the MongoDB database—this is the default configuration, and authentication will be disabled.

Check the environment variable again, and then click the **Create** button to continue and deploy the MongoDB container.

After a few moments, the MongoDB instance should be up and running in the workspace project. You can view this deployment by switching to the **Topology** view.

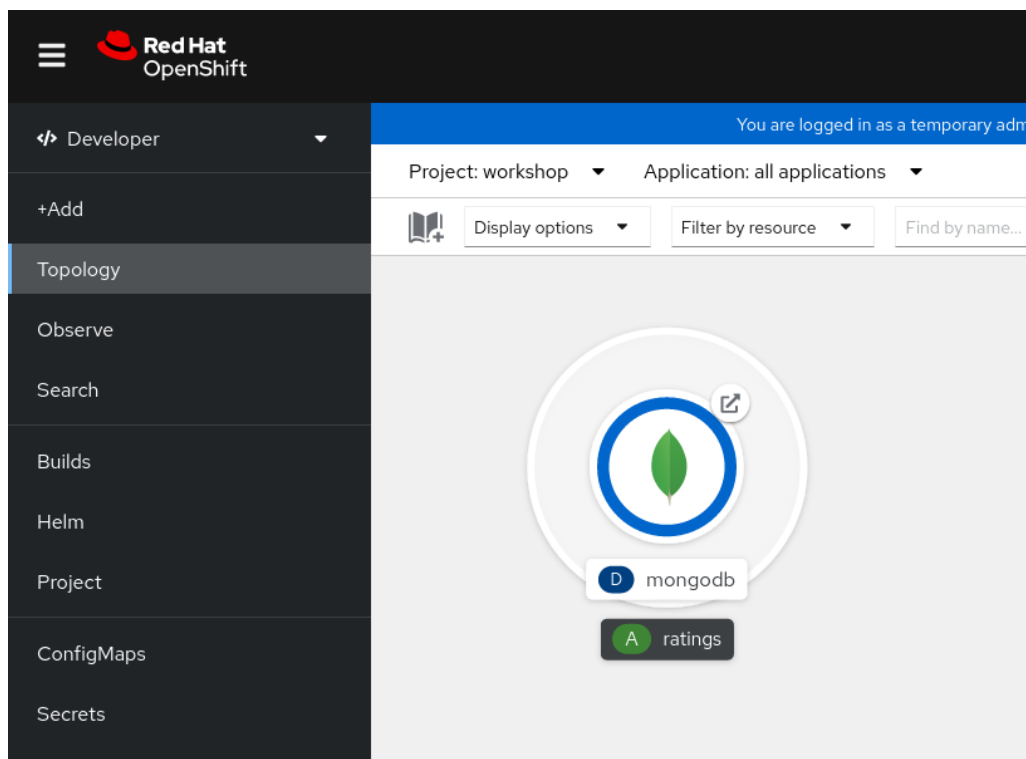


Figure 7.9: The MongoDB instance is up and running

Run the `oc get all` command to view the status of the new application and verify if the deployment of the MongoDB template was successful. An example output of `oc get all` is as follows:

```

user@host: oc get all
NAME                                READY   STATUS    RESTARTS   AGE
pod/mongo-6c6fcb45b8-8wvdm         1/1     Running   0           29s

NAME                TYPE          CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE
service/mongo       ClusterIP     172.30.88.119 <none>       27017/TCP  30s

NAME                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/mongo 1/1     1             1           30s

NAME                DESIRED   CURRENT   READY   AGE
replicaset.apps/mongo-6c6fcb45b8 1         1         1       30s

NAME                IMAGE REPOSITORY
TAGS                UPDATED
imagestream.image.openshift.io/mongo  image-registry.openshift-image-registry.svc:5000/workshop/mongo
latest              30 seconds ago

```

If everything is working correctly, the STATUS column should show the container is Running.

Retrieve MongoDB service host name

Once the deployment is complete, we need to find the service that was created to allow us to access the database from within the cluster. `svc` is short for services:

```

user@host: oc get svc mongodb
NAME     TYPE          CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE
mongo    ClusterIP     172.30.88.119 <none>       27017/TCP  77s

```

The service will be accessible at the following DNS name, `mongodb.workshop.svc.cluster.local`, which is formed of `[service name].[project name].svc.cluster.local`. This resolves only within the cluster.

Deploy Ratings API

It's now time to deploy the second application, which is `rating-api`. It is a Node.js application that connects to a MongoDB instance to retrieve and rate items. Here are some of the details that you'll need to deploy this application:

- `rating-api` on [GitHub](#)
- The container exposes port `3000`
- A MongoDB connection is configured using an environment variable called `MONGODB_URI`

Keep these details noted down, as you will need to refer to them in the next few sections.

Fork the application to your own GitHub repository

To be able to make changes, such as adding CI/CD webhooks, you'll need your own copy of the `ratings-api` code. With Git, this is called a "fork." You can fork the application into your personal GitHub repository. Go to the preceding GitHub repository and click the **Fork** button.

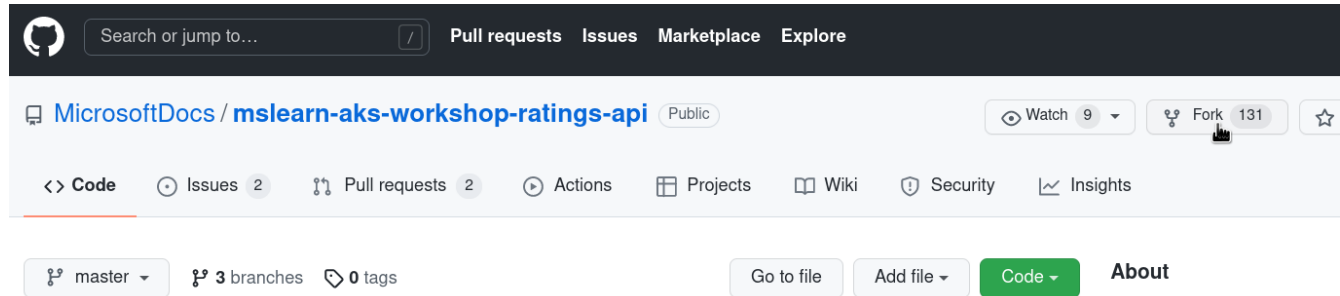


Figure 7.10: Fork the application in the GitHub repository

Make a note of the new repository address—you will need to use this in the instructions that follow.

Use the OpenShift CLI to deploy `rating-api`

OpenShift is able to deploy code directly from a Git repository by looking at the contents and selecting a "builder image" based on the contents—Java, PHP, Perl, Python, or similar. In this case, `rating-api` is a JavaScript application. The builder image will download the JavaScript dependencies using `npm`, and the result will be a new container image stored in the internal OpenShift container registry. This build strategy is called **Source 2 Image (S2I)** and is described in more detail in the glossary.

You can start a new S2I build using `oc new-app`:

```
user@host: oc new-app https://github.com/<your GitHub username>/mslearn-aks-workshop-ratings-api
--strategy=source --name=rating-api

--> Found image 0aea15f (3 weeks old) in image stream "openshift/nodejs" under tag "14-ubi8" for
"nodejs"

Node.js 14
-----
Node.js 14 available as container is a base platform for building and running various Node.js 14
applications and frameworks. Node.js is a platform built on Chrome's JavaScript runtime for easily
building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model
that makes it lightweight and efficient, perfect for data-intensive real-time applications that run
across distributed devices.

Tags: builder, nodejs, nodejs14

* The source repository appears to match: nodejs
* A source build using source code from https://github.com/MicrosoftDocs/mslearn-aks-workshop-
ratings-api will be created
  * The resulting image will be pushed to image stream tag "rating-api:latest"
  * Use 'oc start-build' to trigger a new build
```

Switch to the **Topology** view within the web console, and you should see the application build start and the deployment succeed after a few minutes.

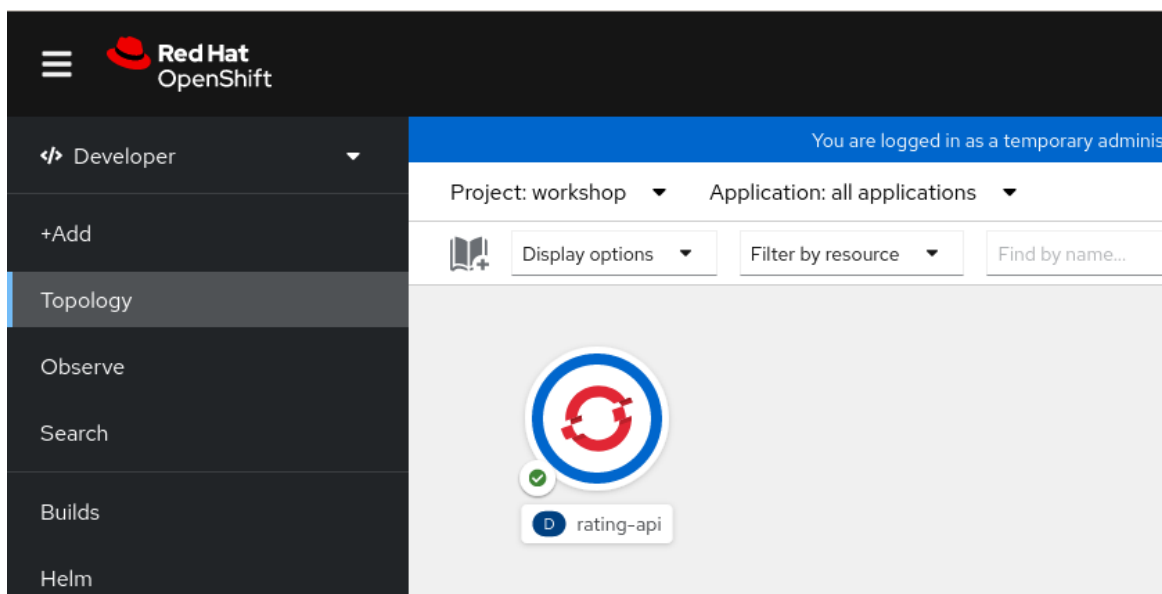


Figure 7.11: The Topology view

The build and startup of the pod should take only a minute or two.

Configure the required environment variables

At this point, the database is deployed, as well as the ratings API. However, it is necessary to tell the ratings API how to connect to the database. Configuring container-based applications is commonly achieved by using environment variables.

Click on the deployment and edit it. Create the following environment variable:

Name	Value
MONGODB_URI	mongodb://mongodb.workshop.svc.cluster.local:27017/ratingsdb

Once you have saved this new environment variable, it will trigger a redeployment of the ratings-api service to use the new environment variable.

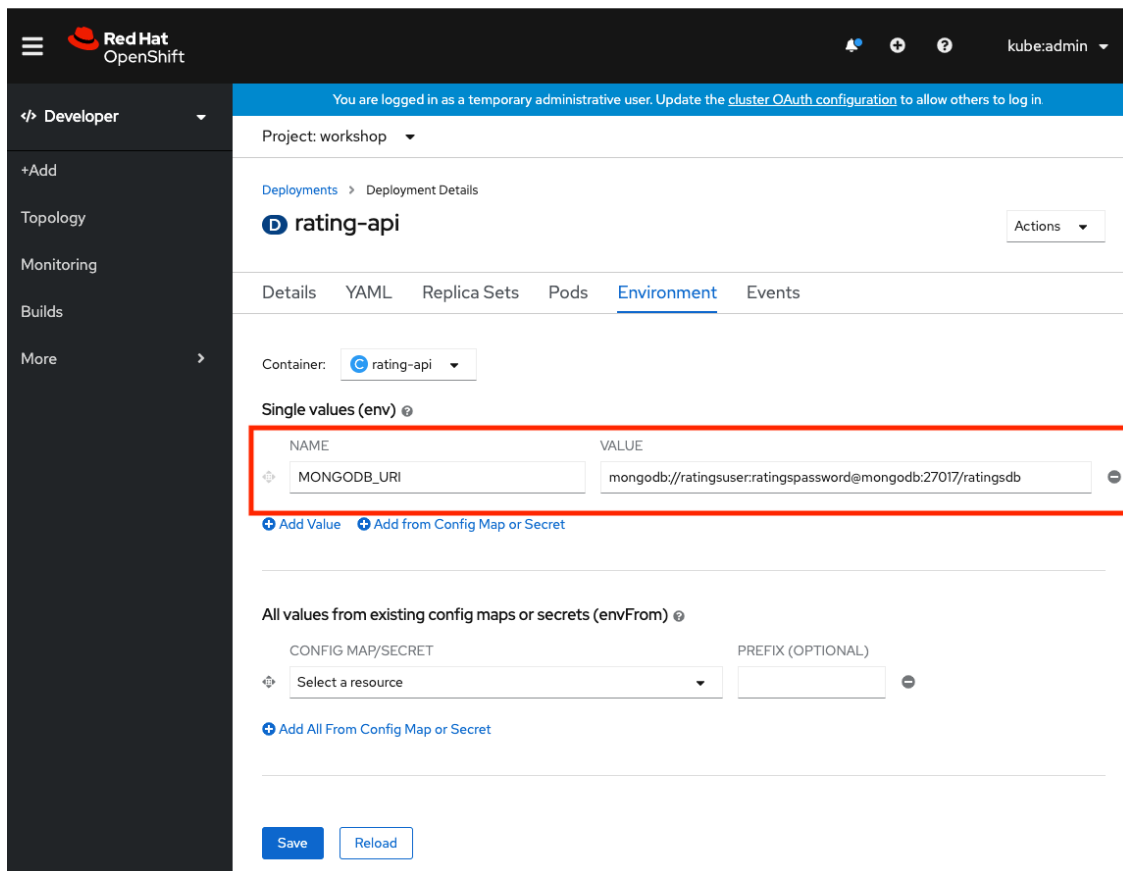


Figure 7.12: Setting the MONGODB_URI environment variable via the web console

It can also be done on the command line, like this:

```
oc set env deploy/rating-api MONGODB_URI=mongodb://mongodb.workshop.svc.cluster.local:27017/ratingsdb
```

Whichever method you use, OpenShift will need to restart the container to make use of the new environment variables.

Verify that the service is running

If you navigate to the logs of the `rating-api` deployment, you should see a log message confirming the code can successfully connect to MongoDB. To do that, on the deployment's details screen, click on the **Pods** tab, then on one of the pods.

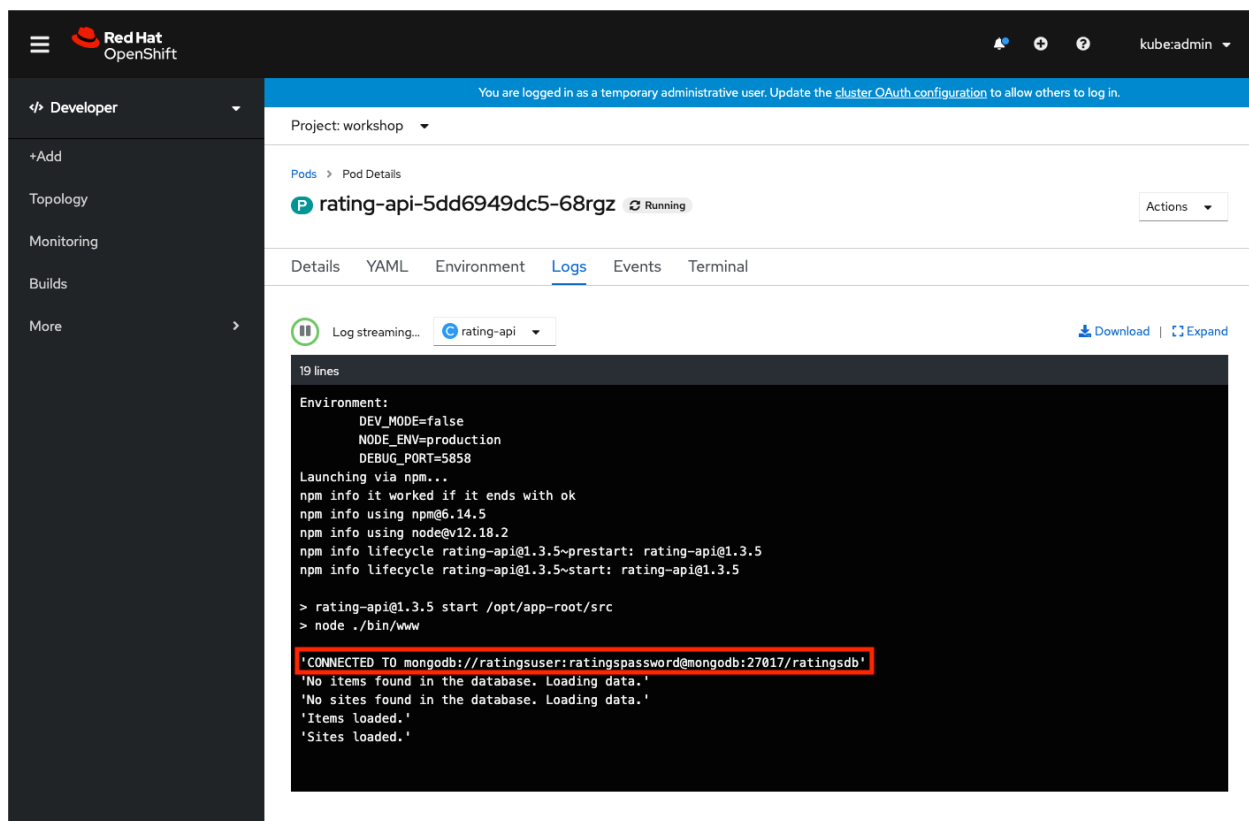


Figure 7.13: Log message confirms that the code can successfully connect to MongoDB

Adjust the rating-api service port

OpenShift will create a service using port `8080`. However, due to a library update, this service runs on port `3000`. It is necessary to edit the default service.

Navigate to the **Networking** → **Services** menu and, from the list of services, edit the `rating-api` service as follows (simply replace `8080` with `3000`):

```
ports:
  - name: 3000-tcp
    protocol: TCP
    port: 3000
    targetPort: 3000
```

Restart the service to use the new port:

```
user@host: oc rollout restart deploy/rating-api
```

Now the service will be bound to the correct port—`3000`, instead of `8080`.

Retrieve the `rating-api` service host name

We will need to validate that there is a `rating-api` service, as it will be used in the next section, when the `rating-web` application is deployed:

```
oc get service rating-api
```

The service will be accessible at the `rating-api.workshop.svc.cluster.local:3000` DNS name, over port `3000`, which is formed of `[service name].[project name].svc.cluster.local`. This resolves only within the cluster.

Deploy the ratings front end

`rating-web` is a Node.js application that connects to `rating-api`. The following are some of the details that you'll need to deploy this application:

- `rating-web` on [GitHub](#)
- The container exposes port `8080`
- The web app connects to the API over the internal cluster DNS, using a proxy through an environment variable named `API`

Use the OpenShift CLI to deploy rating-web

As with the `rating-api` application, this application can be deployed with S2I, using `oc new-app`. However, this time the app will be created with a Dockerfile strategy—using a Dockerfile that is already in the Git repository. Therefore, do not specify a `--strategy` argument:

```
user@host: oc new-app https://github.com/<your GitHub username>/mslearn-aks-workshop-ratings-web
--name rating-web

--> Found container image e1495e4 (2 years old) from Docker Hub for "node:13.5-alpine"

  * An image stream tag will be created as "node:13.5-alpine" that will track the source image
  * A Docker build using source code from https://github.com/MicrosoftDocs/mslearn-aks-workshop-
ratings-web will be created
    * The resulting image will be pushed to image stream tag "rating-web:latest"
    * Every time "node:13.5-alpine" changes a new build will be triggered

--> Creating resources ...
imagestream.image.openshift.io "node" created
imagestream.image.openshift.io "rating-web" created
buildconfig.build.openshift.io "rating-web" created
deployment.apps "rating-web" created
service "rating-web" created
--> Success
```

It will take a short while to build the dependencies into a container—allow 2–3 minutes or so for the build to complete before heading back to the **Topology** view, to see the service come online.

Configure the required environment variables

Create the API environment variable for the `rating-web` deployment config. The value of this variable is going to be the host name/port of the `rating-api` service.

Instead of setting the environment variable through the Azure Red Hat OpenShift web console, you can set it through the OpenShift CLI:

```
oc set env deploy rating-web API=http://rating-api:3000
```

Expose the rating-web service using a route

Exposing the service means there is a publicly accessible URL that users can use to access the service. If the service is not exposed, it can only be accessed within the cluster:

```
user@host: oc expose svc/rating-web
route.route.openshift.io/rating-web exposed
```

Finally, get the URL of the service that was just exposed:

```
user@host: oc get route rating-web
```

NAME	HOST/PORT	PATH	SERVICES	PORT
rating-web	rating-web-workshop.apps.zytjwj9a.westeurope.aroapp.io		rating-web	8080-tcp
Termination	Wildcard			
None				

Notice the **fully qualified domain name (FQDN)** comprises the application name and project name by default. The remainder of the FQDN, the subdomain, is your Azure Red Hat OpenShift cluster-specific apps subdomain.

Try the service

Open the host name in your browser. You should see the ratings app page. Play around, submit a few votes, and check the leaderboard.

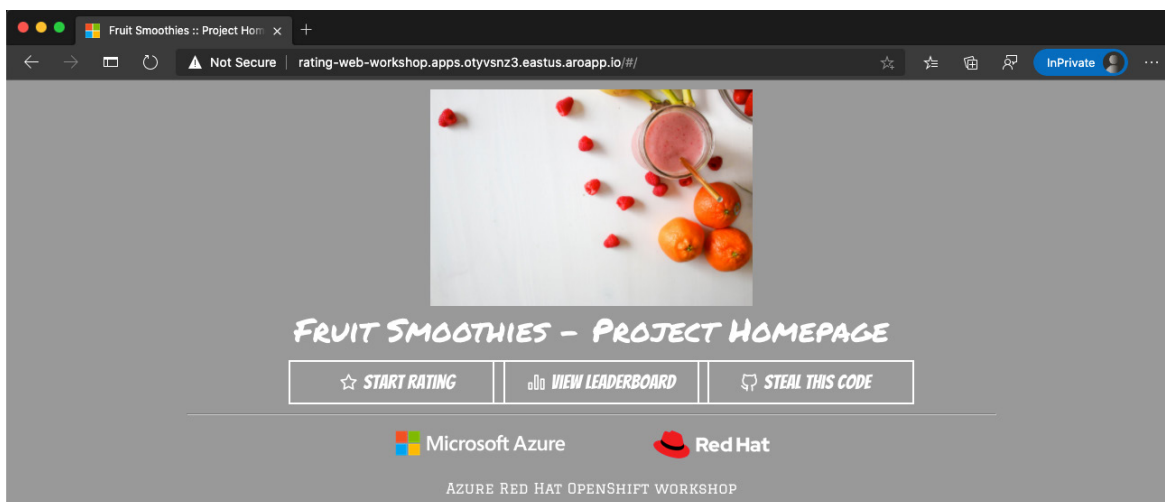


Figure 7.14: Trying the ratings app service

Set up the GitHub webhook

To trigger S2I builds when you push code into your GitHub repo, you'll need to set up the GitHub webhook:

1. Retrieve the GitHub webhook trigger secret. You'll need to use this secret in the GitHub webhook URL:

```
user@host: oc get bc/rating-web -o=jsonpath='{.spec.triggers..github.secret}'  
3ffcc8d5-a243
```

Make a note of the secret key, as you'll need it in a couple of steps.

2. Retrieve the GitHub webhook trigger URL from the build configuration:

```
user@host: oc describe bc/rating-web  
...  
Webhook GitHub:  
  URL:      https://api.quwhfg7o.westeurope.aroapp.io:6443/apis/build.openshift.io/v1/namespaces/  
workshop/buildconfigs/rating-web/webhooks/<secret>/github  
...
```

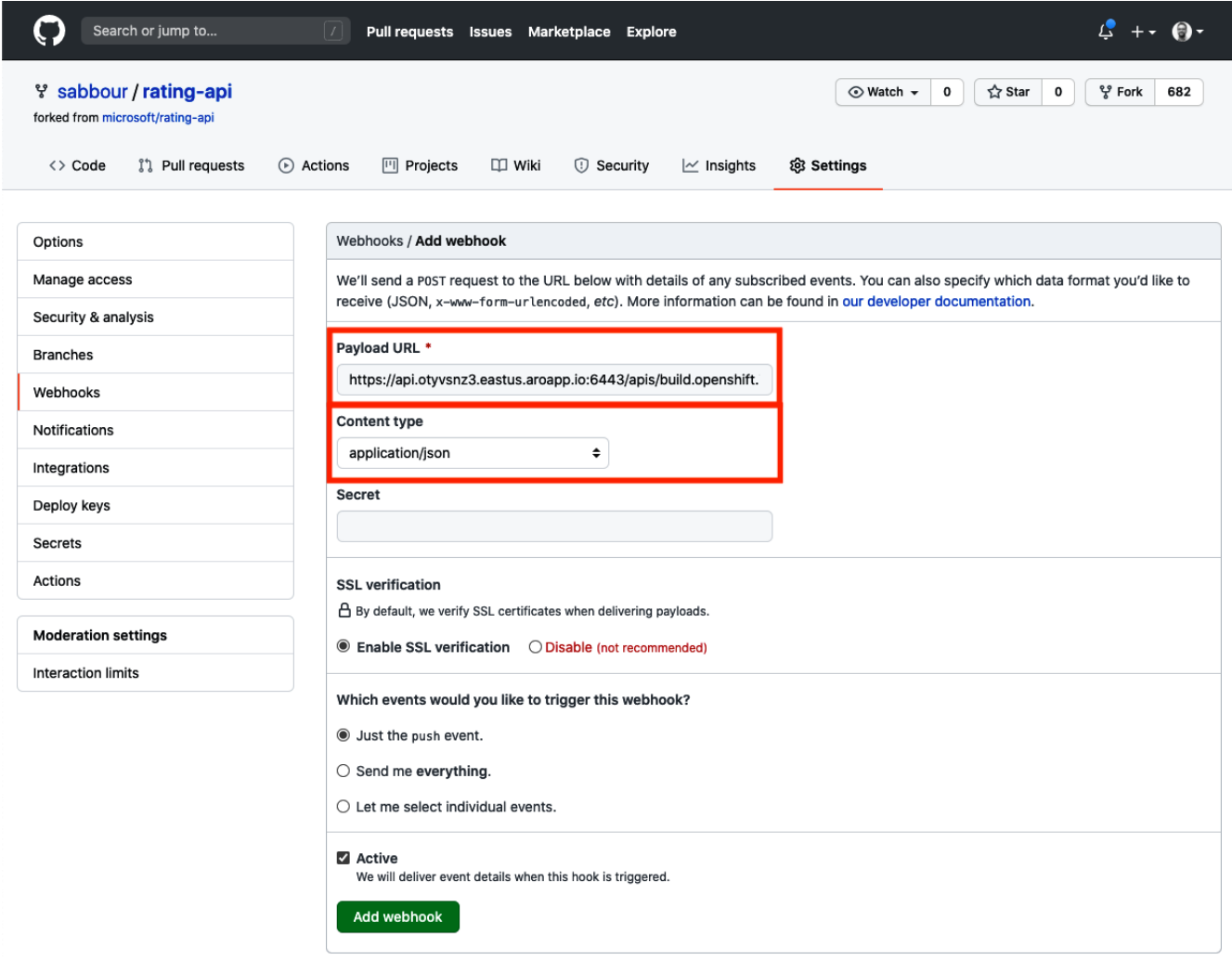
3. Replace the <secret> placeholder with the secret you retrieved in the first step. In this case, the secret is 3ffcc8d5-a243, and so the resulting URL is:

```
https://api.quwhfg7o.westeurope.aroapp.io:6443/apis/build.openshift.io/v1/namespaces/workshop/  
buildconfigs/rating-web/webhooks/3ffcc8d5-a243/github
```

You'll use this URL to set up the webhook on your GitHub repository.

4. Navigate to your GitHub repository. Select **Add Webhook** from **Settings** → **Webhooks**.
5. In the **Payload URL** field, paste your GitHub URL with the <secret> value changed to use your secret.
6. In the **Content type** field, change the default application/x-www-form-urlencoded to application/json.

7. Click **Add webhook**.



The screenshot shows the GitHub repository settings page for 'sabbour / rating-api'. The 'Settings' tab is selected, and the 'Webhooks / Add webhook' section is active. The 'Payload URL' field is highlighted with a red box and contains the URL 'https://api.otyvsnz3.eastus.aroapp.io:6443/apis/build.openshift.'. The 'Content type' is set to 'application/json'. The 'Secret' field is empty. The 'SSL verification' section is expanded, showing 'Enable SSL verification' selected. The 'Which events would you like to trigger this webhook?' section has 'Just the push event.' selected. The 'Active' checkbox is checked, and the 'Add webhook' button is visible at the bottom.

Figure 7.15: Add a webhook

You should see a message from GitHub stating that your webhook was successfully configured.

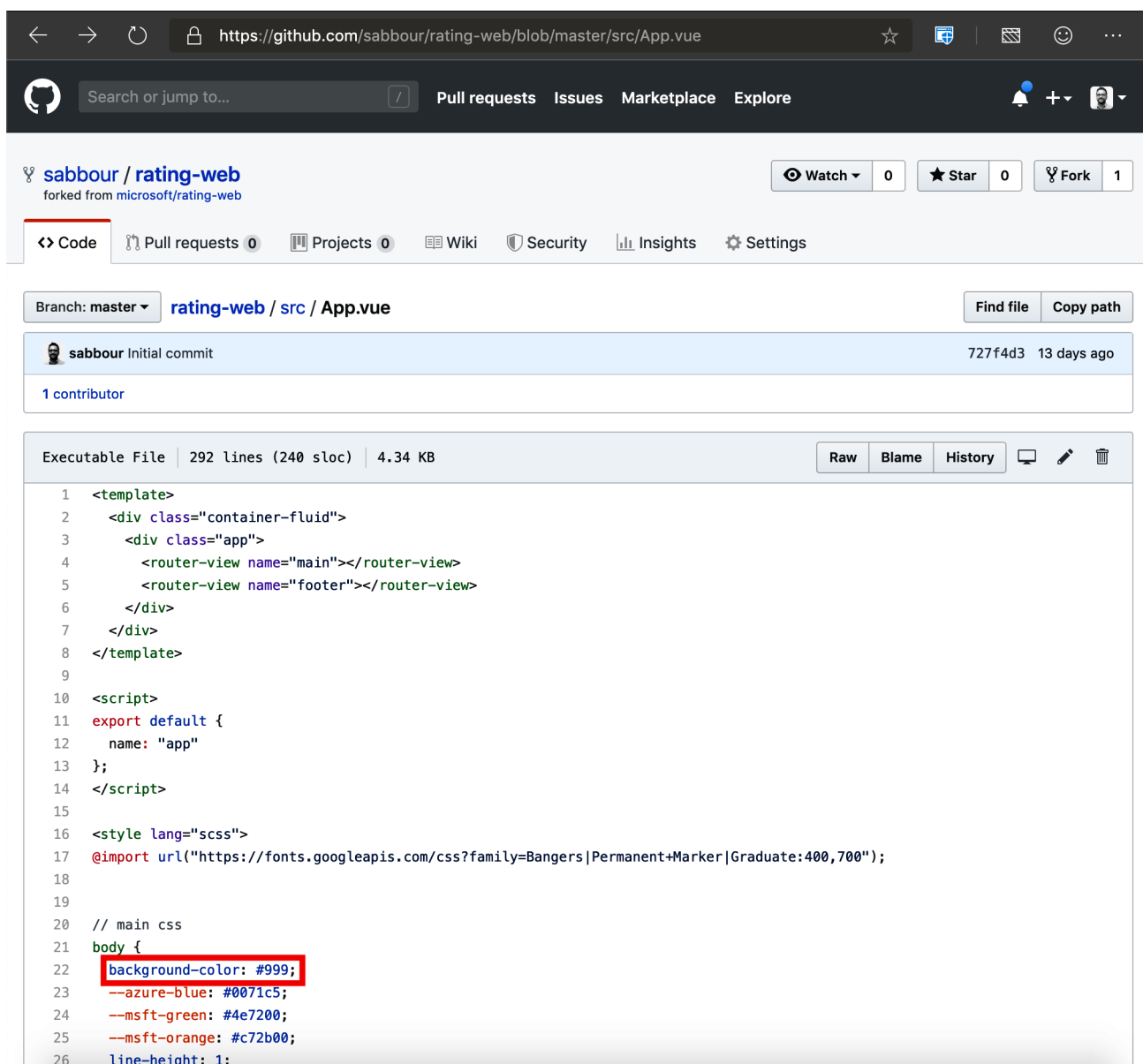
Now, whenever you push a change to your GitHub repository, a new build will automatically start, and upon a successful build, a new deployment will start.

Make a change to the website app and see the rolling update

Go to the <https://github.com/<your GitHub username>/rating-web/blob/master/src/App.vue> file in your repository on GitHub.

Edit the file; change the `background-color: #999`; line to `background-color: #0071c5`.

Commit the changes to the file into the master branch.



The screenshot shows a GitHub repository page for 'sabbour / rating-web'. The file 'rating-web / src / App.vue' is selected, and the code editor displays the following content:

```
1 <template>
2   <div class="container-fluid">
3     <div class="app">
4       <router-view name="main"></router-view>
5       <router-view name="footer"></router-view>
6     </div>
7   </div>
8 </template>
9
10 <script>
11 export default {
12   name: "app"
13 };
14 </script>
15
16 <style lang="scss">
17 @import url("https://fonts.googleapis.com/css?family=Bangers|Permanent+Marker|Graduate:400,700");
18
19
20 // main css
21 body {
22   background-color: #999;
23   --azure-blue: #0071c5;
24   --msft-green: #4e7200;
25   --msft-orange: #c72b00;
26   line-height: 1;
```

Figure 7.16: Commit the changes to the file into the master branch

Immediately, go to the **Builds** tab in the OpenShift web console. You'll see a new build queued up that was triggered by the push. Once this is done, it will trigger a new deployment and you should see the website's color has been updated.

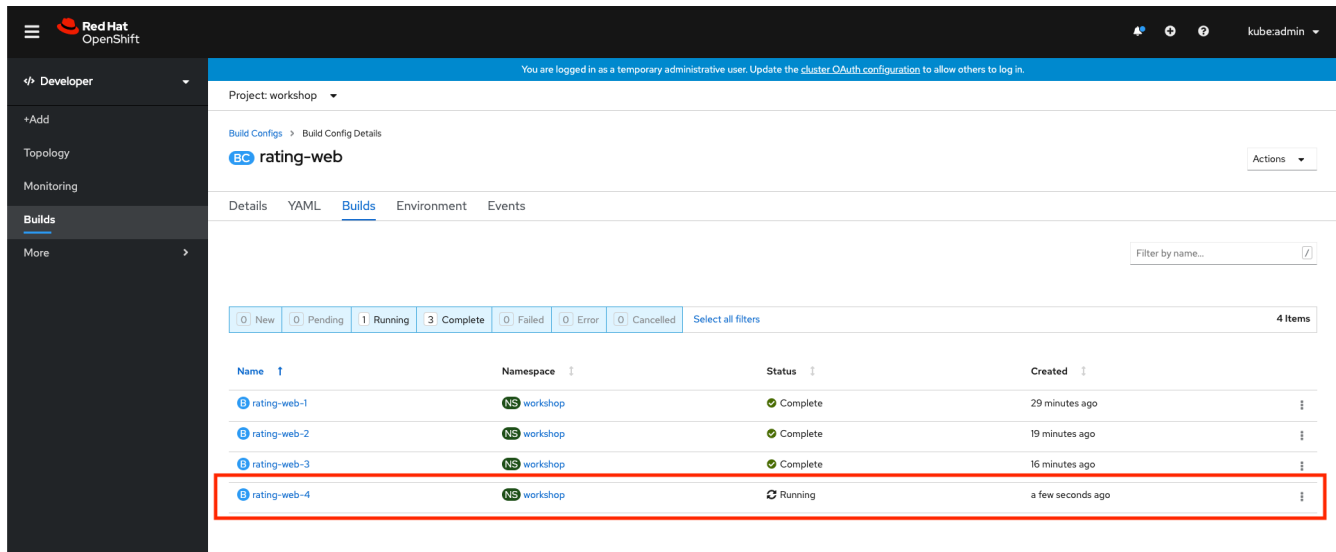


Figure 7.17: The Builds tab shows a new build is running

Now go back to your ratings-web page, and if all has gone well, you will see the background color has changed!

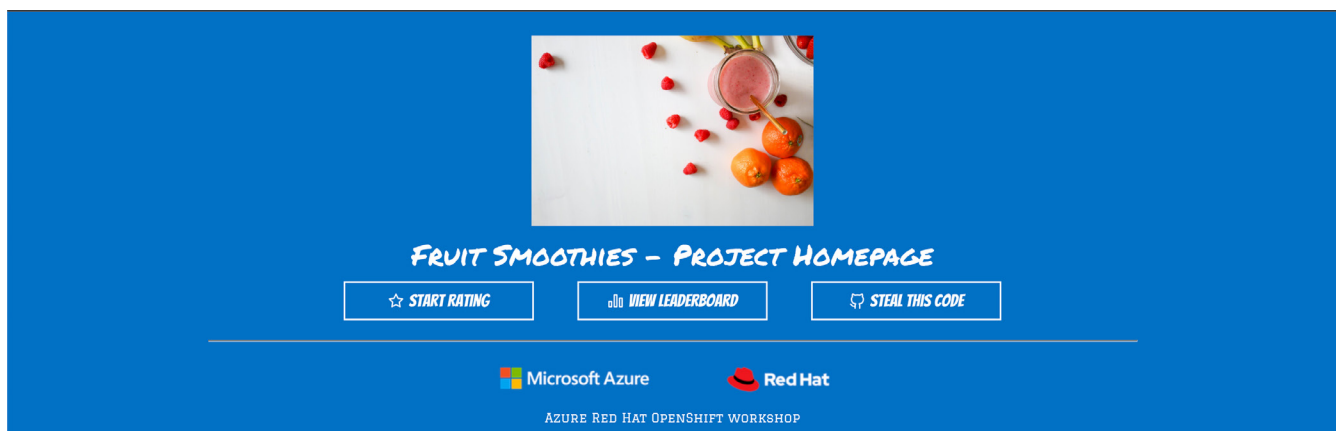


Figure 7.18: The Fruit Smoothies homepage with the new color

Summary

In this chapter, we deployed a basic application that is made up of three smaller microservice applications—a MongoDB database, `ratings-api`, and the `ratings-web` code. While this is not that similar to a production application, it serves as a quick reminder of the concepts when deploying applications onto Azure Red Hat OpenShift. You can use the instructions as a baseline from which to deploy your own applications.

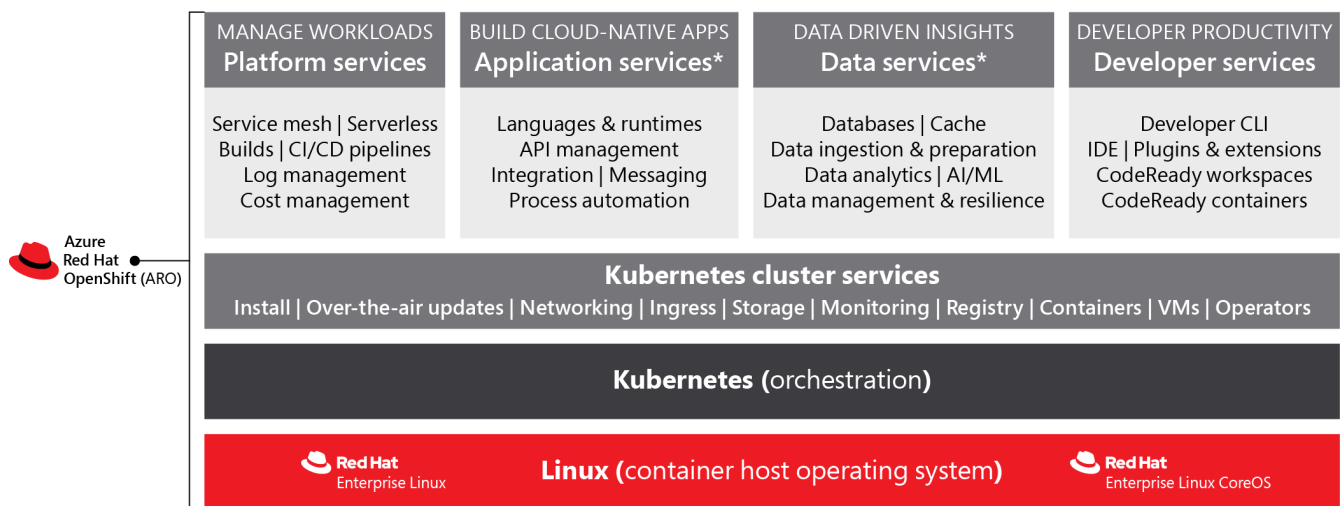
Red Hat OpenShift also supports deploying applications from OperatorHub, as well as Helm Charts, and external CI/CD systems such as Azure DevOps. Exactly which deployment strategy is best for your organization depends on the tools and technologies you use internally.

In the next chapter, we'll explore some of the additional value of Azure Red Hat OpenShift—value that distinguishes OpenShift as an application platform, built on top of Kubernetes. We'll dig into the features and functions that are designed to support the complex needs of enterprise applications.

Chapter 8

Exploring the application platform

In previous chapters, we've learned how Red Hat OpenShift provides a variety of services built on top of Kubernetes. Each of these services combines to create a true application platform and can be grouped into one of five categories—platform services, application services, data services, developer services, and Kubernetes cluster services.



*Red Hat OpenShift® includes supported runtimes for popular languages/frameworks/databases. Additional capabilities listed are from the Red Hat Application and Data Services portfolio.

Figure 8.1: The services included in Azure Red Hat OpenShift

The components grouped in **OpenShift Platform Plus—Multicloud Management, Cluster Security, and Global Registry**—are additional products that require a subscription. They are compatible with Azure Red Hat OpenShift, but not included in the Azure Red Hat OpenShift offering.

In the sections that follow, we'll explore some of the key benefits offered by those services and provide links to where you can find more information.

Cluster services – integrated container registry

Red Hat OpenShift comes with an integrated, internal container registry as soon as the cluster is deployed. This registry is used both for cluster-internal services, such as cluster operators, and by default for customers' application containers as well. This registry is standard, and requires no additional setup—it is even maintained by an infrastructure operator.

All customers who deploy a Kubernetes container service are likely to need to deploy their own container registry as well to keep their container images private. With OpenShift, that additional Day-2 installation and setup are not necessary as the built-in container registry is already available within the cluster. This is an example of a simple, yet time-saving, feature within OpenShift.

[Integrated OpenShift Container Platform Registry overview](#)

Frequently, a cluster administrator might choose to expose this container registry outside of the cluster so that users who are external to OpenShift can push container images to the registry. This is fully supported within Azure Red Hat OpenShift, and documentation for how to do this can be found in [the standard OpenShift documentation on exposing the registry](#).

Platform services – OpenShift Pipelines

Customers of Red Hat OpenShift have many ways of building their applications, and many popular CI/CD tools, such as Jenkins, CircleCI, and GitHub Actions, have plugins that support OpenShift. However, OpenShift also provides functionality on the platform for building applications using cloud-native container pipelines as well, via the OpenShift Pipelines operator.

OpenShift Pipelines is based on the community project called [Tekton](#). Each stage of the pipeline, such as pulling code from a Git repository, running a Java compiler, or assembling an RPM package, is run within a container. This means that developers and operators can form complex and sophisticated pipelines, using the full advantages offered by containers, to build software.

Installing OpenShift Pipelines is possible via OperatorHub. Simply navigate to **OperatorHub** and select the operator to begin the installation. No configuration is required, and the operator installation will normally complete in less than a minute.

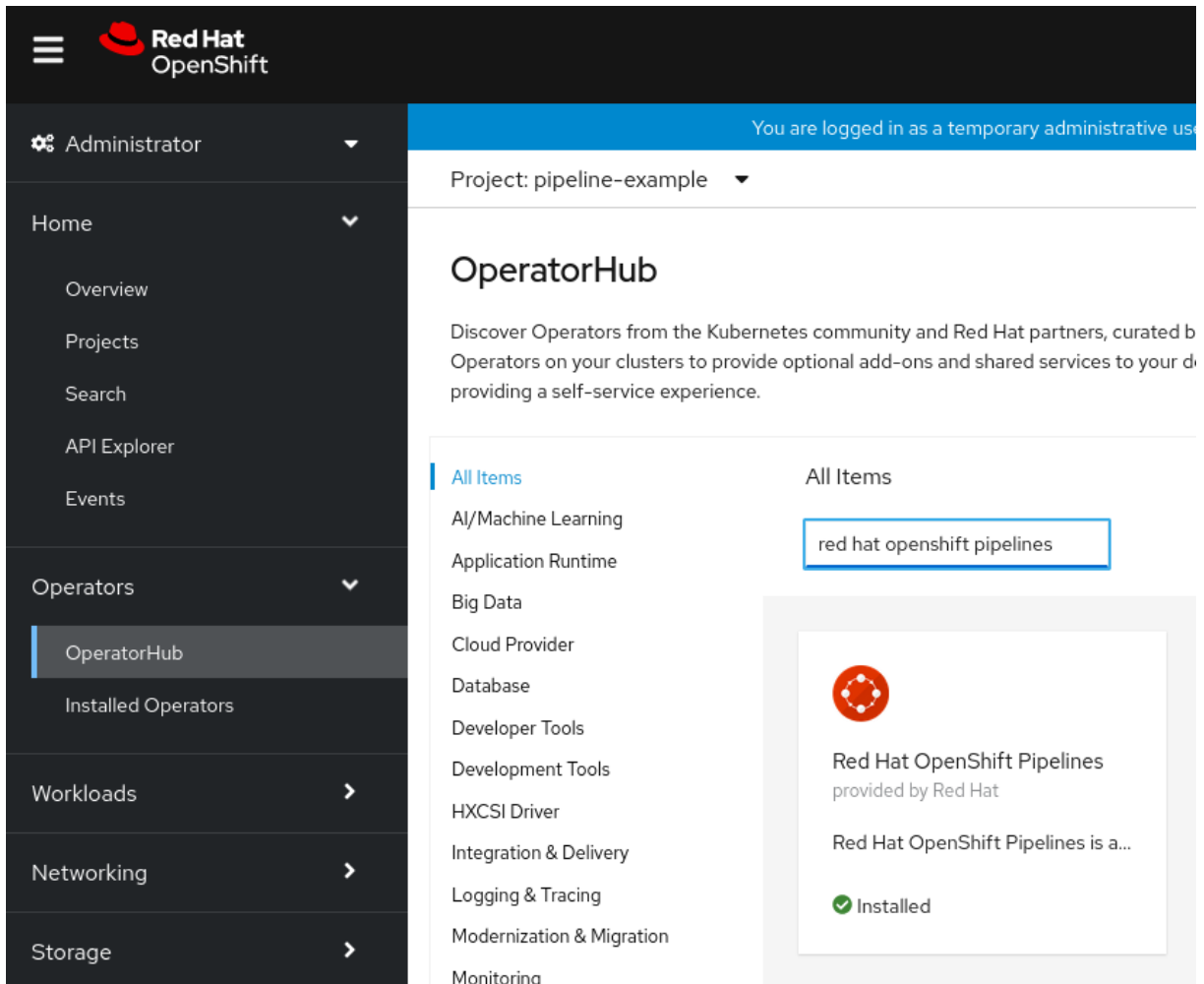


Figure 8.2: Installing OpenShift Pipelines via OperatorHub

When the OpenShift Pipelines operator is installed, you'll see a new **Pipelines** section in the sidebar, as well as the option to add pipelines to catalog items, such as when building the Node.js example.

Pipelines

Add pipeline

Hide pipeline visualization



Figure 8.3: Adding pipelines

Using OpenShift Pipelines, it's possible to use the visual builder to set up and build complex, branching pipelines as well. Here is an example screenshot of a more complex pipeline:

Pipeline builder

Configure via: Pipeline builder YAML view

Name *

complex-pipeline

Tasks *

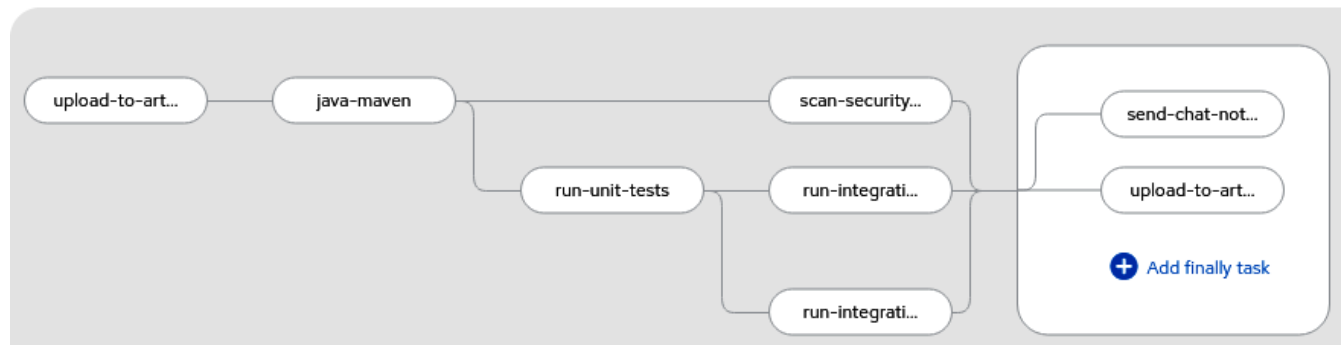


Figure 8.4: A complex pipeline

While many organizations will use a variety of tools and technologies to build their software, OpenShift Pipelines makes it easy to integrate the build directly within OpenShift, using all the advantages afforded by containers. It provides for a consistent, easy-to-use CI/CD solution that requires extremely minimal setup, regardless of the underlying infrastructure being used.

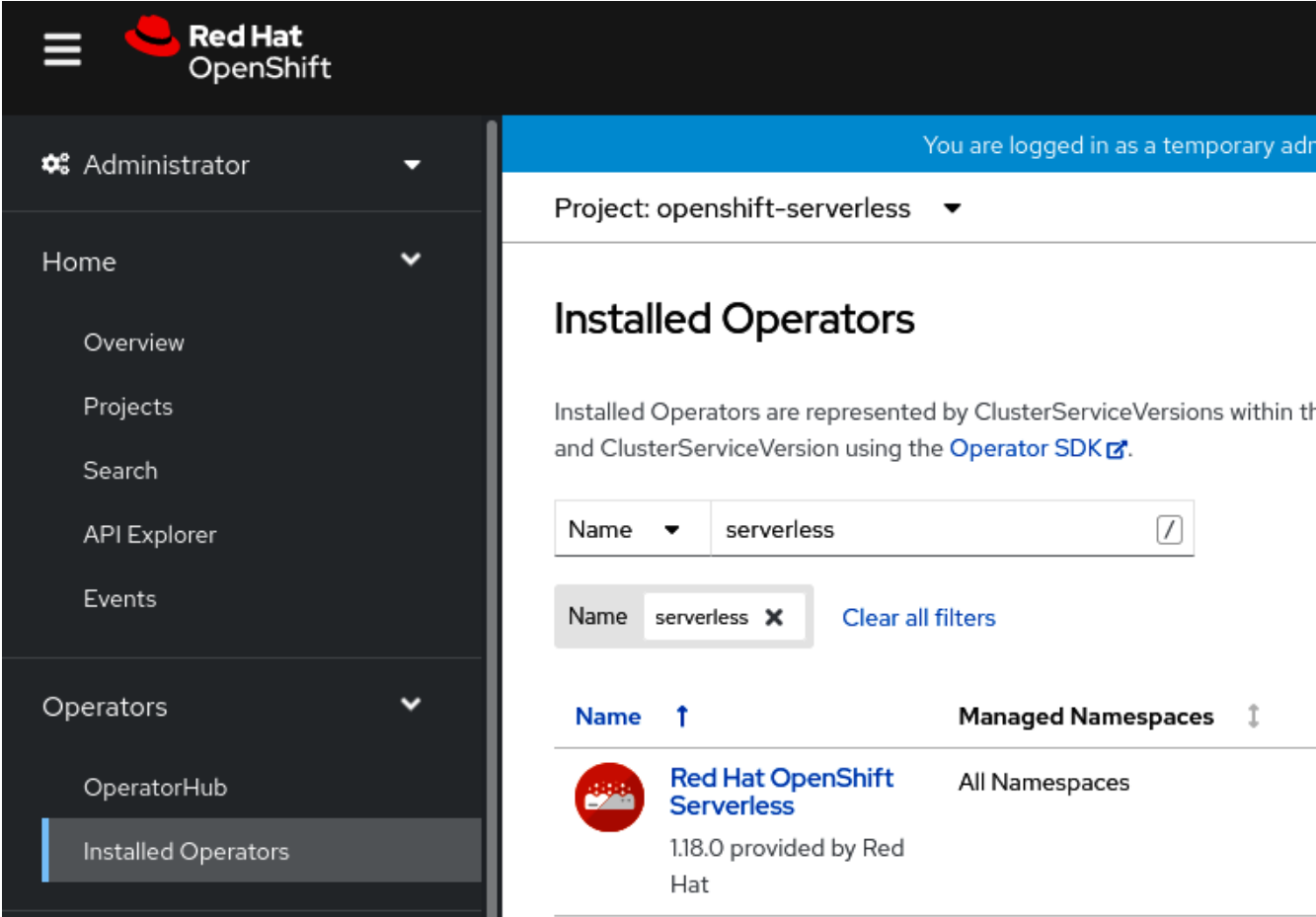
Further reading

- [Understanding OpenShift Pipelines in the OpenShift](#)
- [Tekton community site](#)

Platform services – OpenShift Serverless

It's a common misconception that containers are just a useful technology for long-running services. Actually, many short-lived jobs and serverless functions execute based on short-lived containers. Because of the advantages that containers offer in terms of speed of startup, consistency, and ease of shutdown, they also lend themselves well to serverless workloads. Of course, all serverless services do require underlying servers to execute code, and for this reason, serverless is sometimes referred to as "function as a service."

Red Hat OpenShift enables serverless, or function as a service, workloads via the OpenShift Serverless operator. This operator is based on the popular open-source project called Knative.



The screenshot displays the Red Hat OpenShift console interface. On the left, a dark sidebar contains a navigation menu with options: Administrator, Home, Overview, Projects, Search, API Explorer, Events, Operators, OperatorHub, and Installed Operators. The 'Installed Operators' option is highlighted. The main content area shows the 'Project: openshift-serverless' and a section titled 'Installed Operators'. Below the title, there is a search filter for 'Name' set to 'serverless'. A table lists the installed operators:


Name	Managed Namespaces
 Red Hat OpenShift Serverless 1.18.0 provided by Red Hat	All Namespaces

Figure 8.5: Viewing Installed Operators

Once deployed in the cluster (which again, takes a minute or two typically), it's necessary to do a bit of setup for the operator. There are two **Custom Resource Definitions (CRDs)** in particular to pay attention to—**Knative Serving** and **Knative Eventing**.

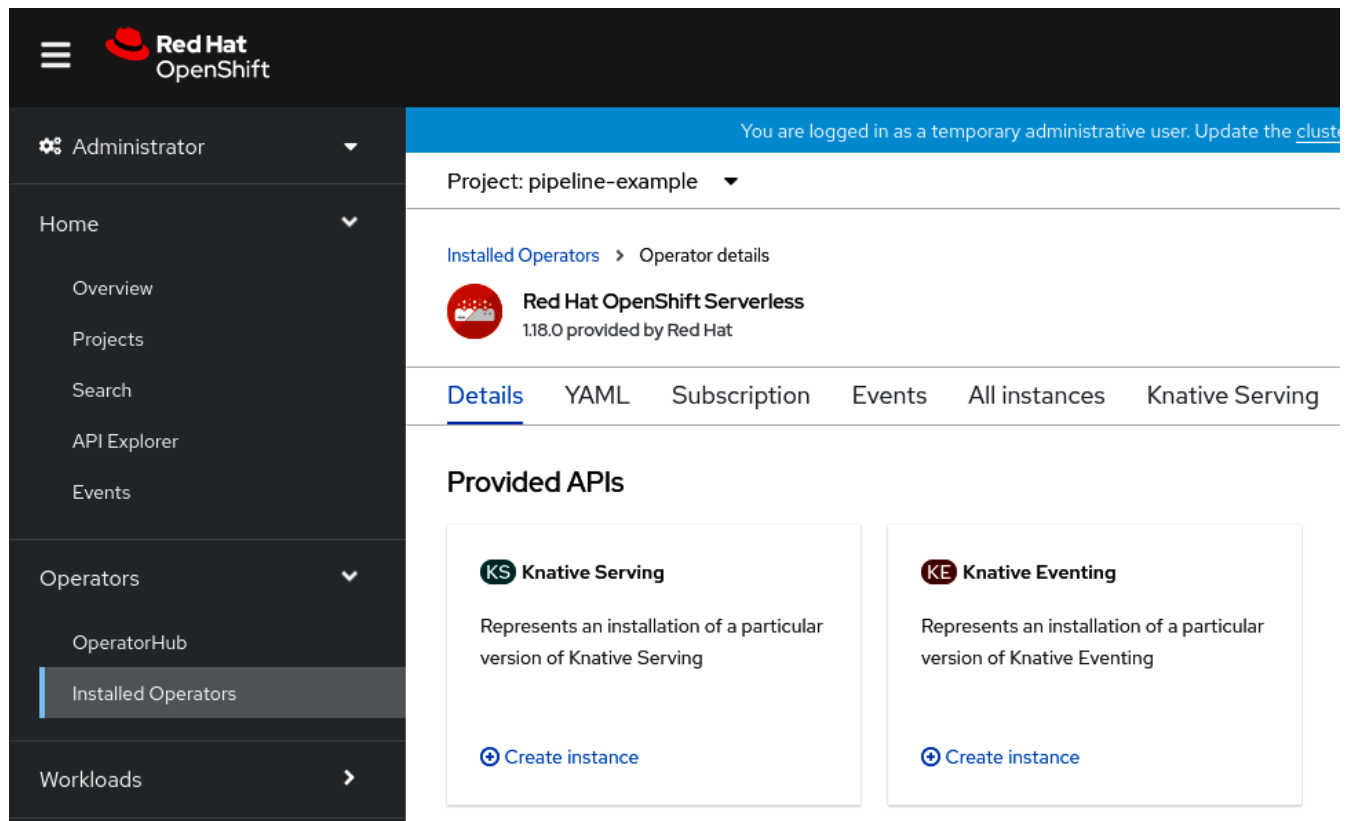


Figure 8.6: Knative Serving and Knative Eventing in the Operators menu

- **Knative Serving** simplifies the application deployment, dynamically scales based on incoming traffic, and supports custom rollout strategies with traffic splitting; for example, a function that uploads an image to a storage bucket, and logs that upload to a NoSQL database.
- **Knative Eventing** allows "late binding" of event sources at application runtime (as opposed to at build time); for example, an application that responds to new image uploads in a storage bucket—that application doesn't need to know about the storage bucket at build or event deploy time, but Knative Eventing makes it easy to "bind" that event source to the application at runtime.

The two sections that follow include examples for each of these types of serverless applications on OpenShift.

Platform services – OpenShift Serverless – Knative Serving example

Let's demonstrate how Knative Serving enables autoscaling of an application, in particular, scaling to zero when there are no requests. One very simple application we can use is a web page that serves ASCII pictures of pets:

- [php-ascii-pets GitHub repository](#)

Adding this repository from Git is straightforward—Red Hat OpenShift automatically detects a compatible builder image of PHP.

OpenShift detects how to build this project automatically.

Import from Git

Git

Git Repo URL *



Validated

> [Show advanced Git options](#)



Builder Image detected.

A Builder Image is recommended.



PHP 7.4 (UBI 8)



[Edit Import Strategy](#)

BUILDER PHP

Build and run PHP 7.4 applications on UBI 8. For more information about using this builder image, including OpenShift considerations, see <https://github.com/sclorg/s2i-php-container/blob/master/7.4/README.md>.

Figure 8.7: A builder image is automatically detected and recommended

The important part that makes this a "serverless" deployment is when the deployment type is selected. Notice that when OpenShift Serverless is installed, a "serverless" deployment becomes available.

Resources

Select the resource type to generate

Deployment

apps/Deployment

A Deployment enables declarative updates for Pods and ReplicaSets.

DeploymentConfig

apps.openshift.io/DeploymentConfig

A DeploymentConfig defines the template for a Pod and manages deploying new Images or configuration changes.

Serverless Deployment

serving.knative.dev/Service

A type of deployment that enables Serverless scaling to 0 when idle.

Figure 8.8: Serverless deployment type

It will take a few moments for the first build to complete. Once the build finishes, there should be a pod deployed. The topology view should look something like this:

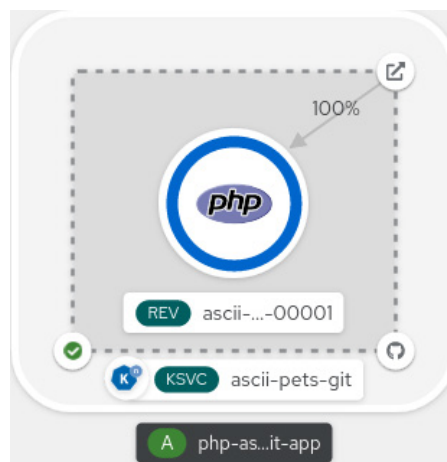


Figure 8.9: A Knative Serving app

The actual application page looks like *Figure 8.9*. It is a very simple application, but the "pet," the ASCII art, is tied to the pod hostname. In our simple demo application, when we put lots of additional load on this application, Knative Serving will spawn additional pods, and you will be able to visually see that different "pets" are served up!

However, if the application is left without any requests for a minute, Knative Serving will scale this application down to zero. Looking at the topology view shows that the application is no longer running.

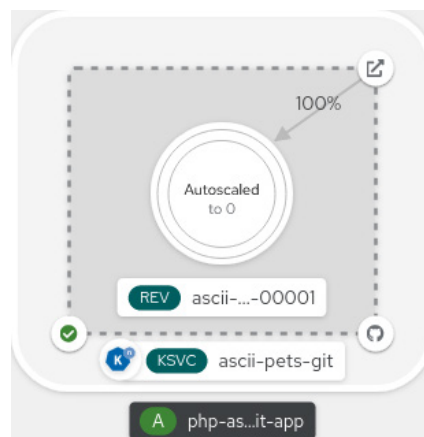


Figure 8.10: A serverless deployment, using Knative Serving, with an application scaled to zero

Finally, if someone visits the page, then the application will be autoscaled back up to 1 replica, or 2, or 3, or more, depending on how many instances OpenShift believes are necessary to serve demand.

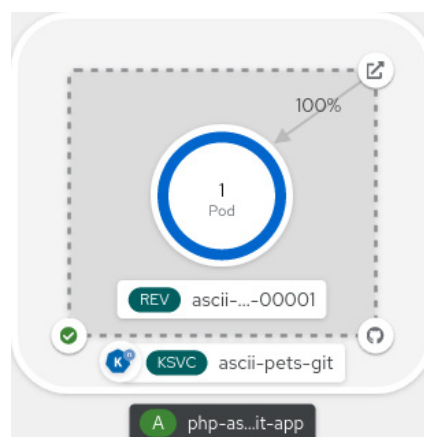


Figure 8.11: Autoscaling in response to traffic

Red Hat OpenShift Serverless, with Knative Serving, allows organizations to scale up, and down, dynamically, with very little additional configuration and without any changes to applications. This can be incredibly useful for keeping deployments the right size and preventing resources from being used and paid for unnecessarily.

Further reading

- [About OpenShift Serverless](#)
- [learn.openshift.com, which includes a course on OpenShift Serverless](#)
- [Knative community site](#)

Platform services – OpenShift Serverless – Knative Eventing example

Building on the example application and fundamentals from the previous chapter, OpenShift Serverless can also scale an application based on metrics besides just ingress traffic. Especially in scenarios such as event-driven architecture, it's desirable to be able to scale up instances of an application to process events from a message queue, or wake up according to a timer.

Using the same deployment, the OpenShift console allows easy configuration of different event sources.

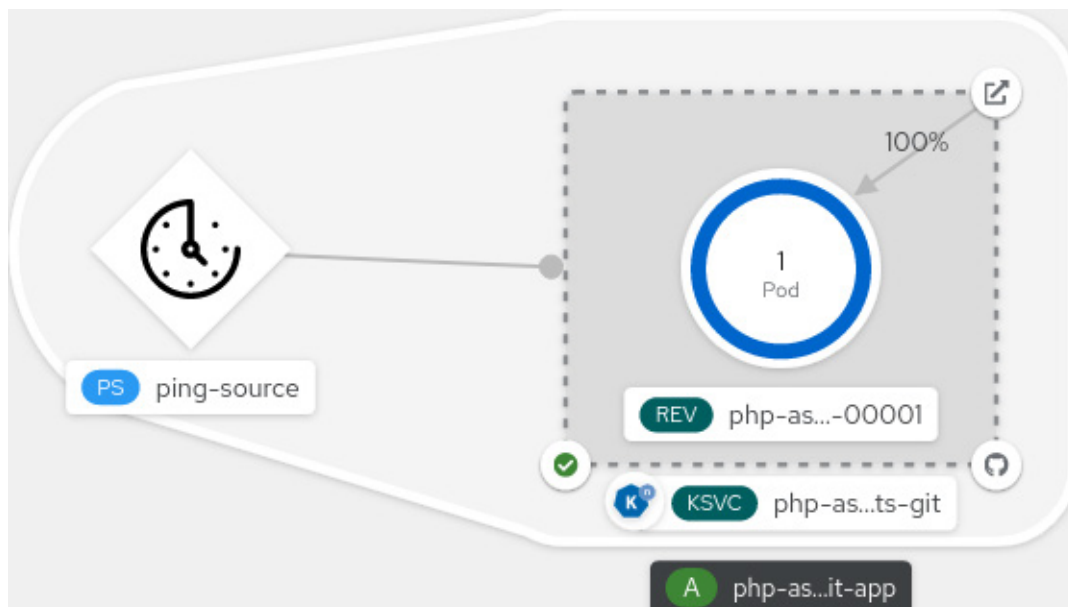


Figure 8.12: A "ping" event source, which pings the application according to a timer

A simple use case for a ping event source would be to "wake up" a backup job container every night at midnight. Another use of this ping timer might be for a monitoring container that runs periodically.

Further reading

- [OpenShift Serverless Eventing explained in 5 minutes](#)
- [About OpenShift Serverless](#)

Platform services – OpenShift Service Mesh

Red Hat OpenShift Service Mesh, based on the open-source Istio project, adds a transparent layer to existing distributed applications without requiring any changes to the service code. You add Red Hat OpenShift Service Mesh support to services by deploying a special sidecar proxy throughout your environment that intercepts all network communication between microservices. You configure and manage the service mesh using the control plane features.

Use cases that can be enabled with OpenShift Service Mesh include:

- Transparent encryption to inter-service communication, with automatic mTLS.
- Serving multiple versions of a service, and enabling A/B testing, for example.
- Observability into how microservices are communicating, with Kiali.
- Tracing service-to-service calls, with Jaeger.

Like all other platform features of OpenShift, Service Mesh is installed with a Red Hat operator.

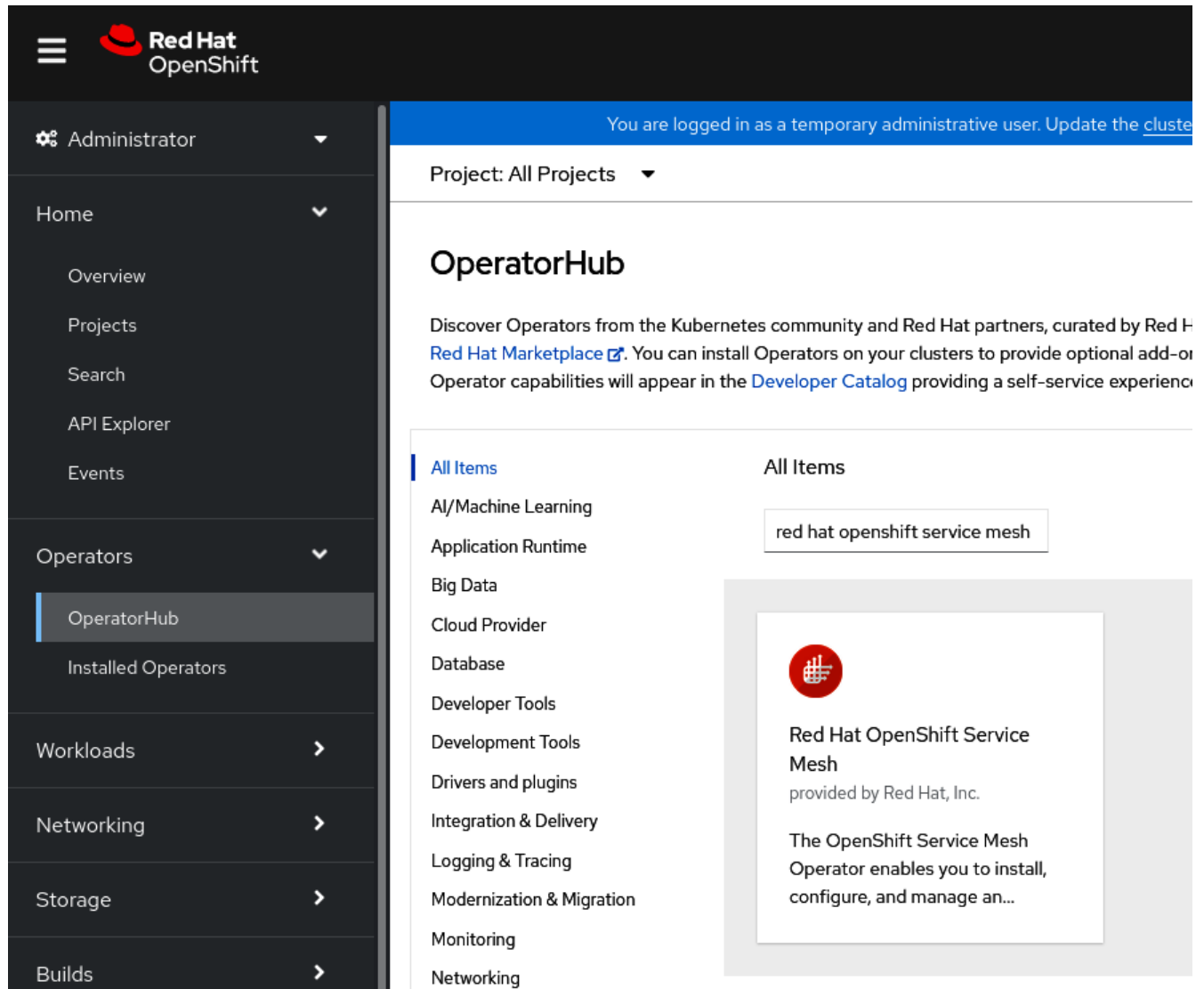


Figure 8.13: Installing Service Mesh via OperatorHub

The [documentation for OpenShift Service Mesh](#) comes with a fantastic example application called the BookInfo demo. This is a simple application that emulates a bookstore, running on OpenShift.

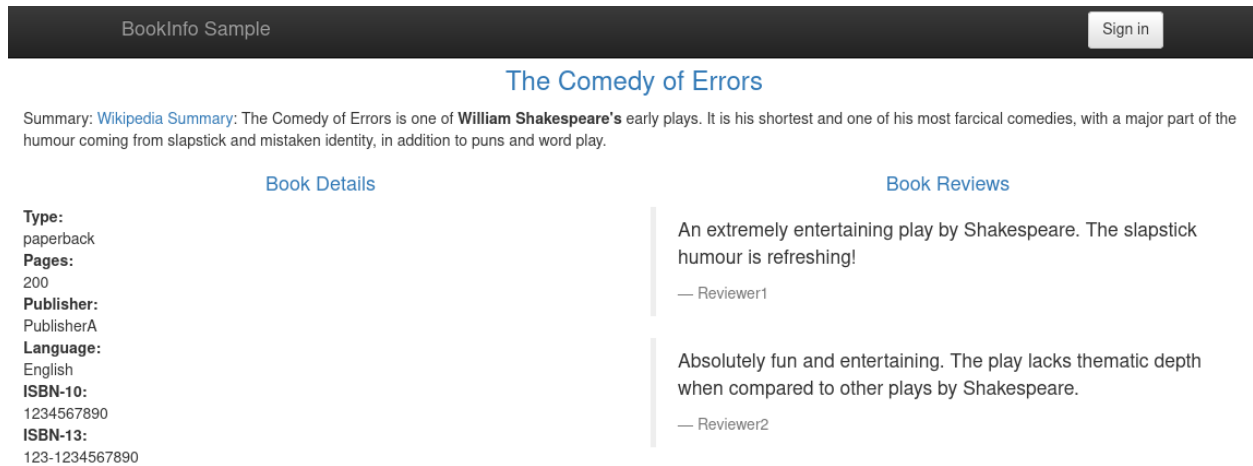


Figure 8.14: The BookInfo application

To get the BookInfo application working with Service Mesh, it is necessary to create a Service Mesh control plane. This is easy to do via the OpenShift graphical interface, as per *Figure 8.15*:

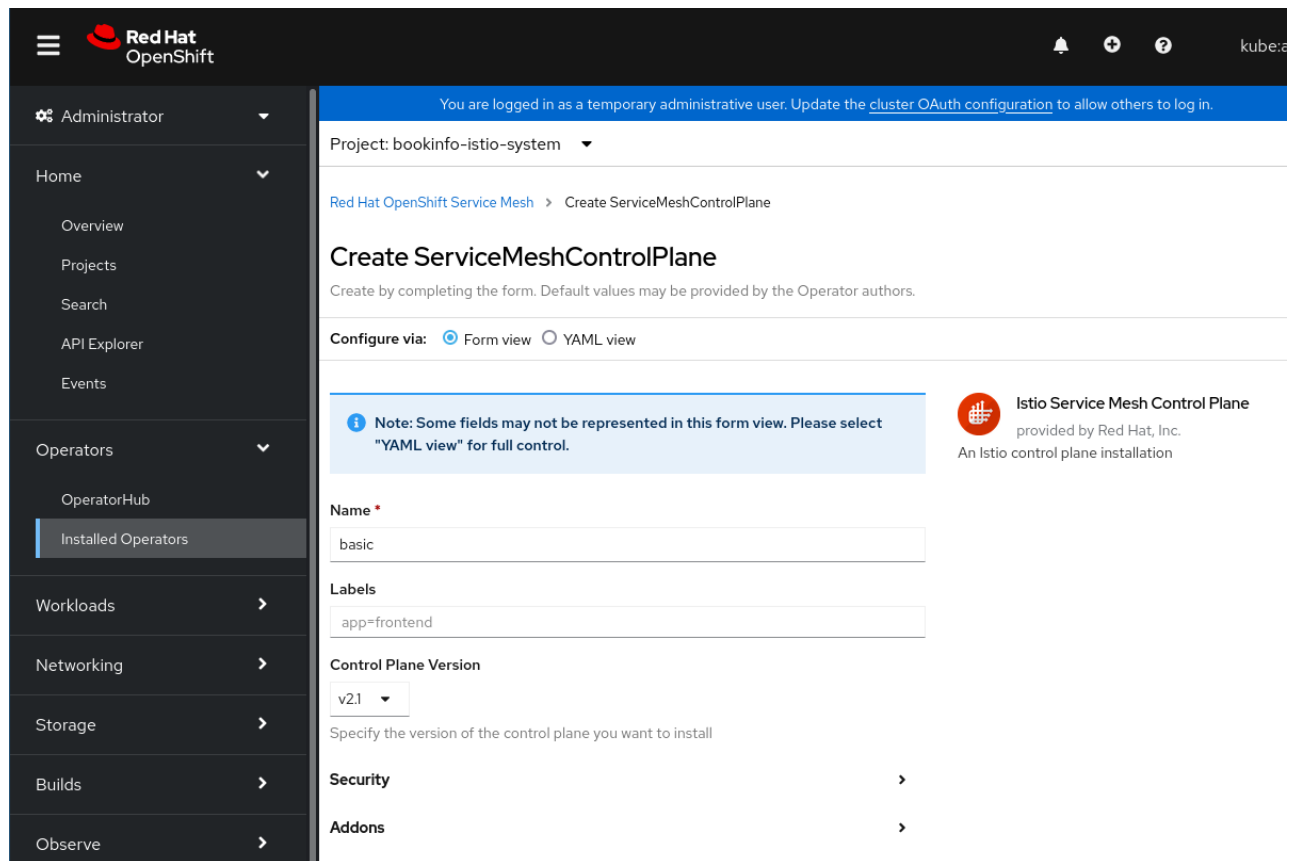


Figure 8.15: Setting up the control plane in the BookInfo bookinfo-istio-system project

The BookInfo project will accept ingress traffic via the control plane, and in this case, a separate project was created to house the control plane, called `bookinfo-istio-system`.

It isn't required to separate the Service Mesh control plane and your project, and it is even possible to share a Service Mesh control plane across multiple projects.

In the next two sections, we'll explore two Service Mesh use cases in much more detail—observability and distributed tracing.

Platform services – OpenShift Service Mesh – observability with Kiali

Looking at the underlying pods that make up this application, you can see from the Red Hat OpenShift Topology view that it is comprised of six microservices.

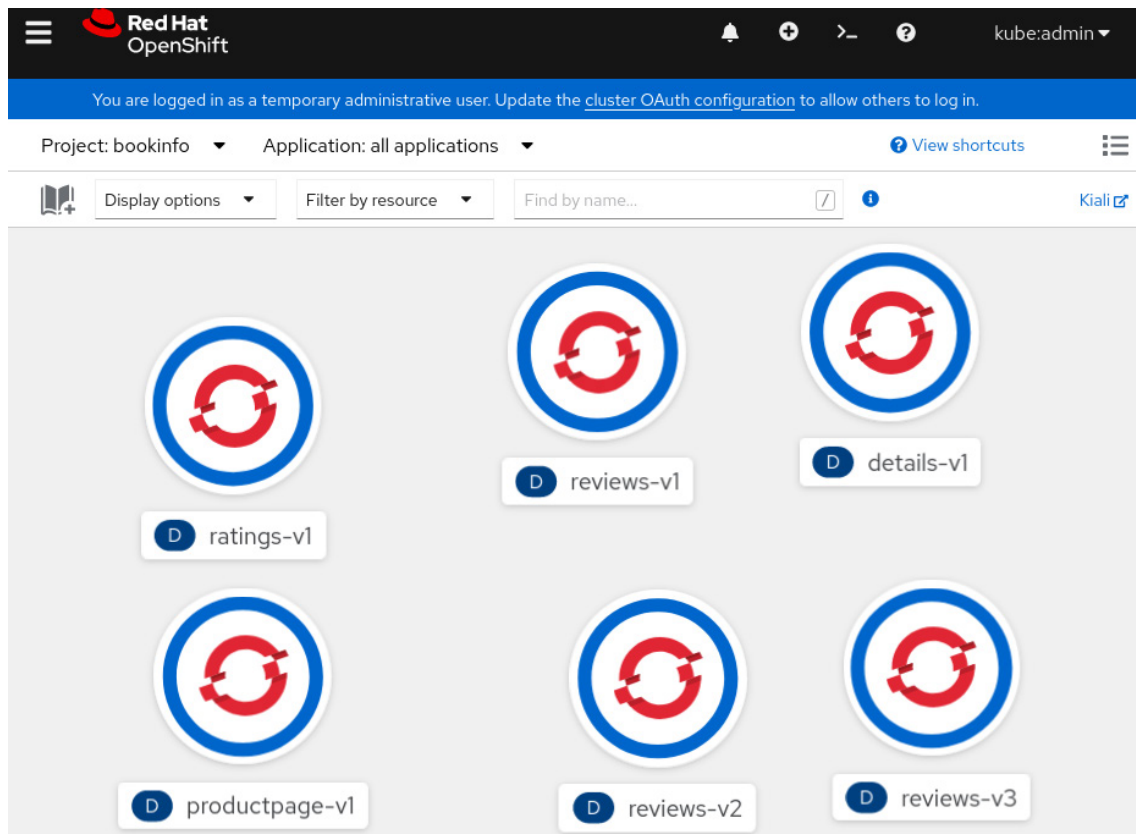


Figure 8.16: The six services that make up the BookInfo application

While this view is useful, it does not really convey how these services are connected. Enter the first benefit of Service Mesh—observability. If we open a slightly different console, called Kiali, which comes with Service Mesh, we can see that there is a far more refined architectural representation of those same six services.

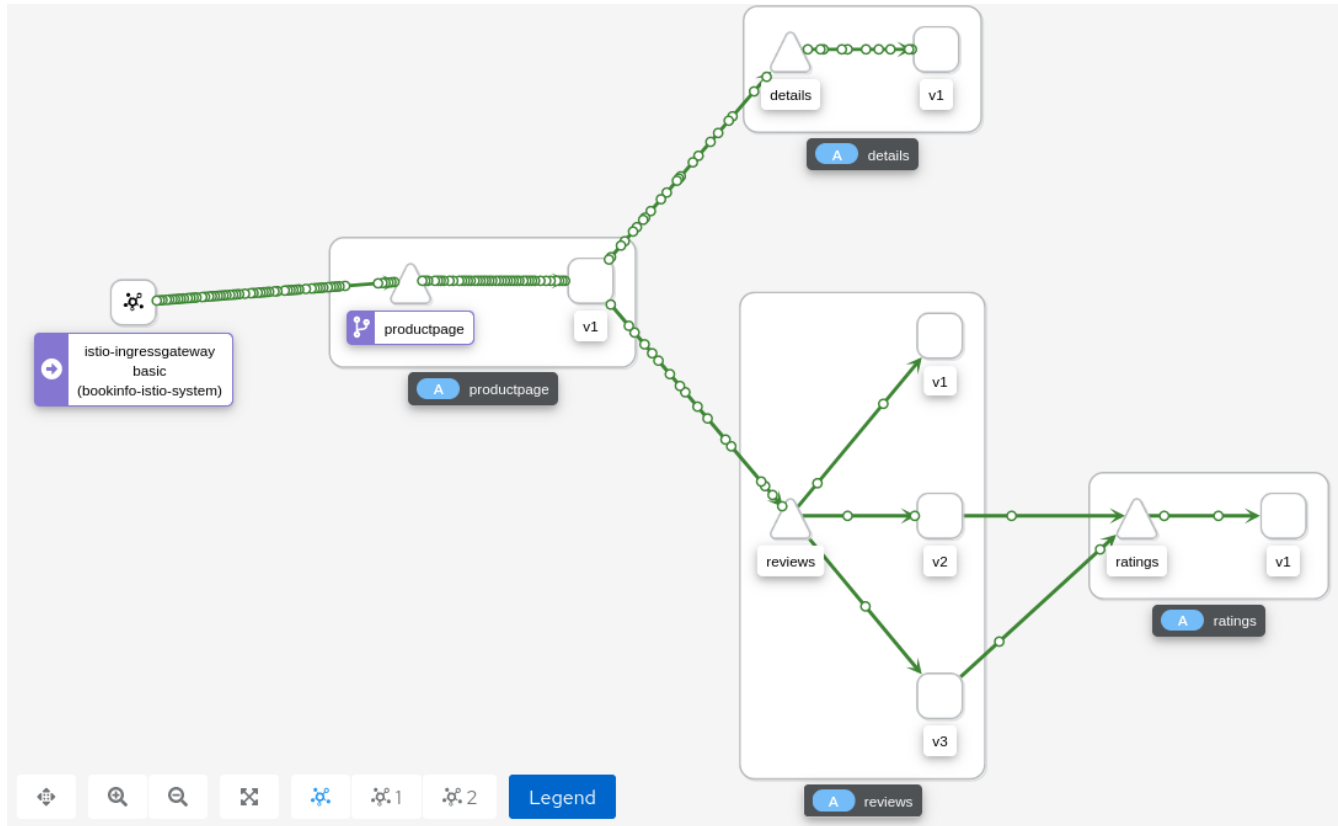


Figure 8.17: The automatically generated application architecture graph, enabled by Service Mesh

This view shows not only the real connectivity between the services but can also a real-time traffic diagram. In this case, the application is getting a sizable amount of traffic, with each request represented by a circle animation on the connection.

Taking this observability further, an administrator can click on one of the service connections and see the traffic profile. In this case, we can see that the traffic mostly has the **HTTP OK** status.

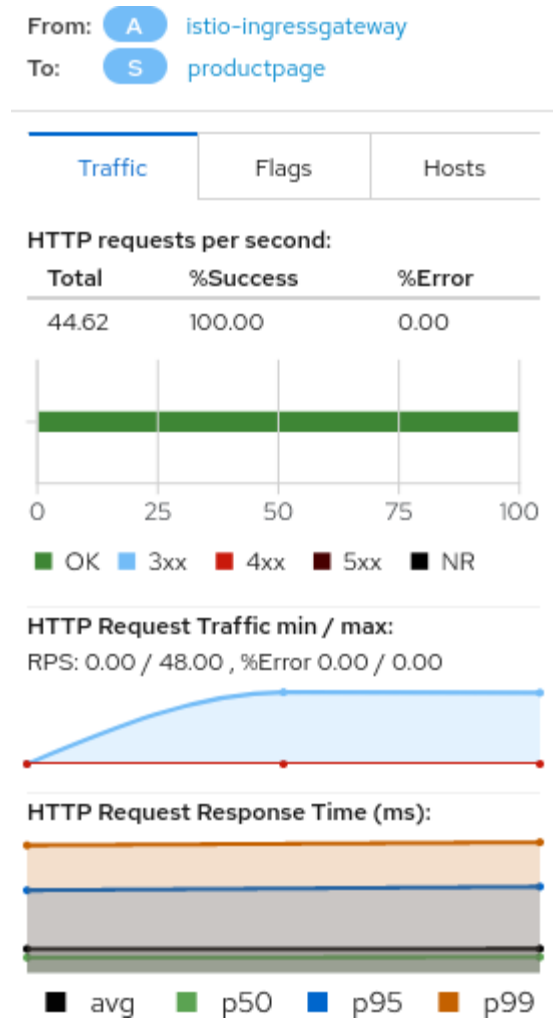


Figure 8.18: Various HTTP statuses

We can introduce an artificial error into this application by shutting down the `details` microservice, scaling it to zero replicas:

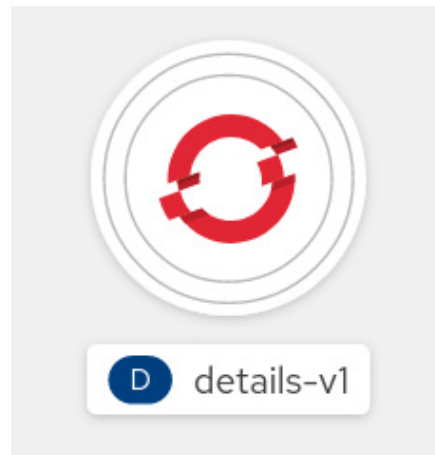


Figure 8.19: Zero replicas of the `details` service, to simulate a failure

Now, if we return to the Kiali view, you'll notice a few extra components have been added to the diagram, but most notably, the connection to the `details` service is highlighted in red to indicate an error.

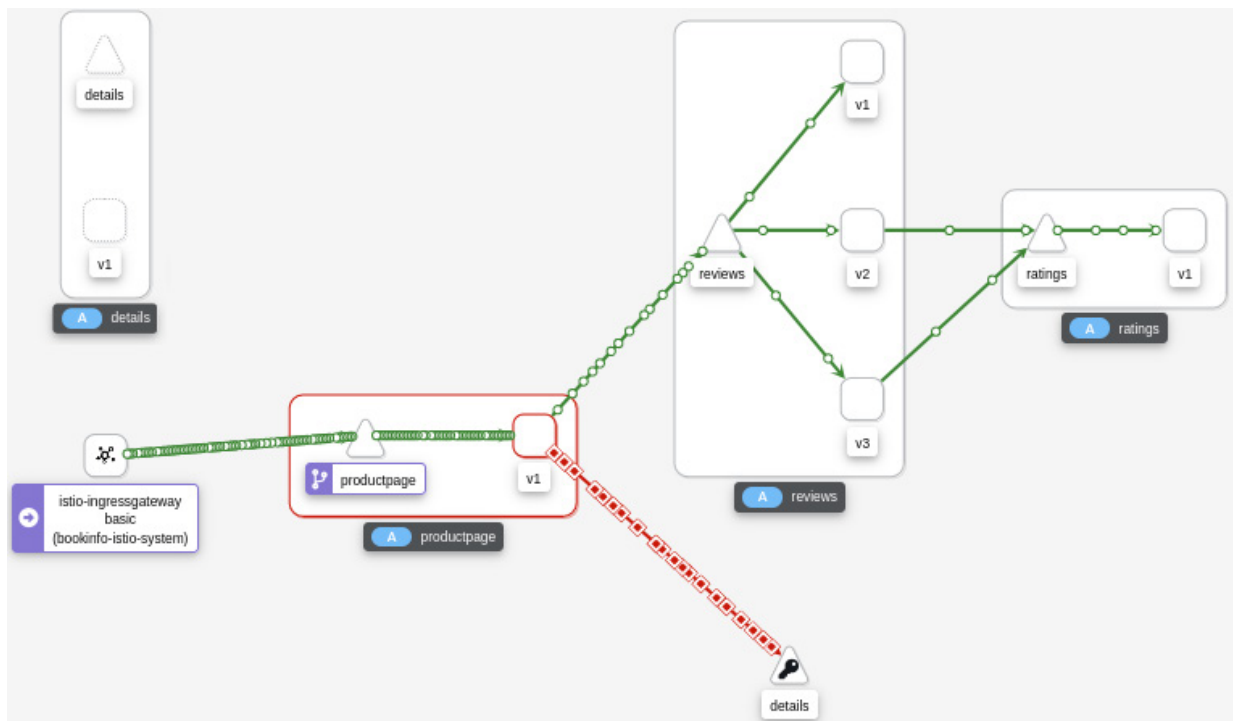


Figure 8.20: The red connection indicates a problem with this service

We've learned that Service Mesh's Kiali really helps with observability. In smaller applications such as BookInfo, Service Mesh is helpful, but as applications become larger and more complex, sometimes involving 20 or more microservices and lots of different types of interactions, having tools such as Service Mesh and Kiali becomes incredibly valuable, and almost essential.

Platform services – Red Hat OpenShift Service Mesh – distributed tracing with Jaeger

Another hugely valuable service provided by Service Mesh is Jaeger. This enables distributed tracing for complex microservices applications. In the previous section, we looked at the BookInfo application and saw that when the `details` service was taken offline, requests started to fail.

The cause of the failure in this case was simple to identify; the `details` service was unavailable, and we caused it! However, if the cause was more mysterious and required further investigation, then Jaeger's distributed tracing would have been helpful.

Jaeger tracks requests from the first service, through subsequent connections through the system. While some additional application code is required to enable Jaeger, the client libraries and minimal code changes make this relatively easy for developers.

Looking at the `productpage` service (which is written in Python), we can pick out the relevant code that enables Jaeger distributed tracing in that service. This code imports the Jaeger client library in the `productpage` service:

```
from jaeger_client import Tracer, ConstSampler
from jaeger_client.reporter import NullReporter
from jaeger_client.codecs import B3Codec
```

Once the client library is imported, the product page then needs to set up a new tracer. This code initializes a Jaeger Tracer in the productpage service:

```
tracer = Tracer(  
    one_span_per_rpc=True,  
    service_name='productpage',  
    reporter=NullReporter(),  
    sampler=ConstSampler(decision=True),  
    extra_codecs={Format.HTTP_HEADERS: B3Codec()})  
)
```

Lastly, still in the productpage service, we tell Jaeger that we want to create a new span—a new request to several services:

```
span = tracer.start_span(  
    operation_name='op', child_of=span_ctx, tags=rpc_tag)  
)
```

Jaeger will then track this tracer through several services. Those also need to be adapted to work with Jaeger, but client libraries are available for most popular programming languages.

The full source code for the productpage service can be found on [GitHub](#).

There are two options for instrumenting code—using the Jaeger client libraries, which are now considered deprecated, or using the open standard options from the OpenTelemetry project, which are preferred:

- [OpenTelemetry libraries](#)

After this effort of instrumenting an application is complete, you'll see traces looking like this:

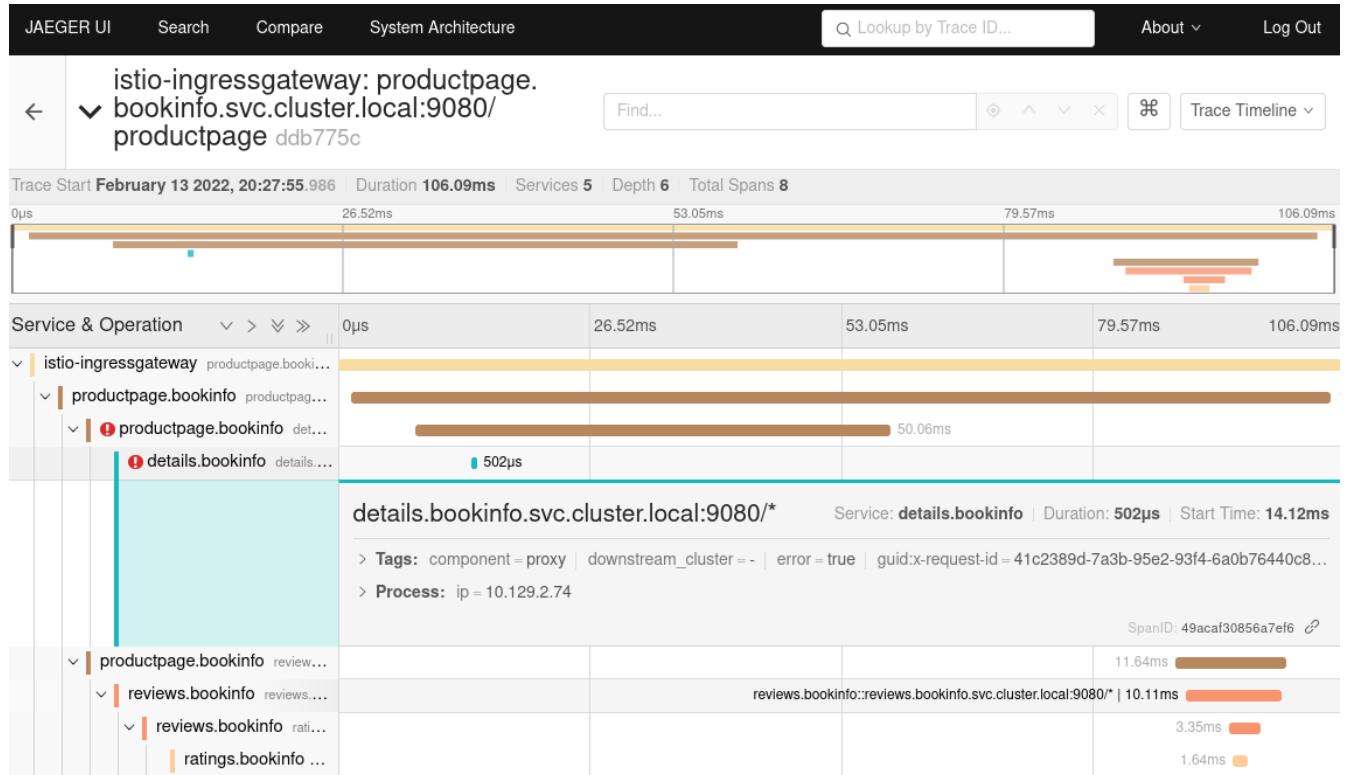


Figure 8.21: A call trace

This view shows a call trace, much like Kiali did, but here it's shown in a hierarchical view. Each of the rows shown can be expanded, revealing details about the request duration, start time, source IP address, and several other details. Again, this level of information becomes almost essential when dealing with complex microservices applications.

Looking back at the error we were trying to diagnose, the trace view clearly shows that the `details` microservice is experiencing issues. In this case, the error is straightforward, but expanding the details view shows that the service is issuing HTTP 503 error codes.



The screenshot shows a trace view for the `details.bookinfo` service. The trace entry is for `details.bookinfo.svc.cluster.local:9080/*` with a duration of 502µs and a start time of 14.12. The trace is expanded to show the following tags:

Tags	
guid:x-request-id	41c2389d-7a3b-95e2-93f4-6a0b76440c83
http.method	GET
http.protocol	HTTP/1.1
http.status code	503

Figure 8.22: Error details

HTTP 503 is the standard error code for "service unavailable." In this case, it's simply because the service is scaled to zero replicas. Scaling the service back up again restores the service.

Jaeger and Kiali combined are powerful tools as microservices applications become more complex. They are provided as part of the Azure Red Hat OpenShift service and do not require additional spending or subscriptions.

Summary

This chapter covered a few of the key value-added services offered by the application platform—Red Hat OpenShift—as opposed to just an "empty" Kubernetes environment. While it is possible to replicate a similar level of functionality of OpenShift by installing all of the respective upstream open-source projects on OpenShift, integration, life cycle, and support are the complexity that is delivered with Azure Red Hat OpenShift.

These various platform services, application services, data services, developer services, and cluster services ultimately make your developers and operators more productive when working with Kubernetes, and when deploying multiple complex enterprise applications.

Chapter 9

Integration with other services

It is quite normal that an application cannot exist entirely within Red Hat OpenShift—most applications rely on external databases or services on Azure in some form or another.

Azure Red Hat OpenShift does not "break" any kind of connectivity here. If an application relies on Azure CosmosDB, for example, it should still be able to connect out from Azure Red Hat OpenShift to Azure CosmosDB without any changes to the application. Depending on your application and your organization, you may deploy some of these external dependencies separately from Azure Red Hat OpenShift, or at the same time.

If you think it would be beneficial to deploy these external dependencies with your application, then Azure Service Operator can greatly simplify this process. Instead of having to use the Azure CLI, the Azure Cloud Shell, or ARM templates, you can deploy these resources as if they were native in Kubernetes by utilizing Azure Service Operator.

Azure Service Operator

Azure Service Operator is yet another operator that makes your life easier, as a developer or administrator deploying applications on Azure Red Hat OpenShift. One advantage is that it allows the easy provisioning of Azure services without having to leave the OpenShift console, which makes it faster and easier to get applications deployed that might have Azure service dependencies, such as Azure CosmosDB.

Another, possibly more significant, advantage of using Azure Service Operator is that it allows for those dependencies on Azure services to be stored alongside the definition of the OpenShift application, in an infrastructure-as-code approach, using standard Kubernetes YAML files.

The way that it works is straightforward: Azure Service Operator looks for new **CRDs**, defined in YAML, that match a definition for a resource on Azure. Azure Service Operator will then translate this YAML into the necessary Azure API calls to create whatever is being asked for on Azure. Once the operator is installed, users can browse the OpenShift Developer Catalog and see the creation screen for many common Azure resources, such as Azure public IP addresses, Azure SQL databases, or Azure Firewall.

The screenshot displays the Red Hat OpenShift Developer Catalog interface. The top navigation bar includes the Red Hat OpenShift logo, a hamburger menu, a notification bell with '16' alerts, a plus sign, a question mark, and the user 'kube:admin'. The left sidebar contains navigation options: Developer, +Add, Topology, Monitoring, Search, Builds, Pipelines, Helm, Project, Config Maps, and Secrets. The main content area is titled 'Developer Catalog' and shows a list of 49 items. The items are categorized by type, with 'Operator Backed' selected. The visible items include:

- APIMgmtAPI**: Creates an API with the specified properties in the specified API Management service.
- ApimService**: Deploys an API Management instance into a specified Resource Group at the specified location.
- AppInsights**: Deploys an Application Insights instance into a specified Resource Group at the specified location.
- AppInsightsApiKey**: Creates an Api Key to access a given Application Insights instance.
- AzureLoadBalancer**: Deploys an Azure Load Balancer (LB) into a specified Resource Group at the specified location.
- AzureNetworkInterface**: Deploys an Azure Network Interface (NIC) into a specified Resource Group at the specified location.
- AzurePublicIPAddress**: Deploys an Azure Public IP Address (PIP) into a specified Resource Group at the specified location.
- AzureSqlAction**: Allows you to roll the password for the specified SQL server.
- AzureSqlDatabase**: (Two instances shown)
- AzureSqlFailoverGroup**: (Two instances shown)

Figure 9.1: Some of the Azure services exposed by Azure Service Operator

Azure Service Operator can be installed via OperatorHub and made available to all users of OpenShift.

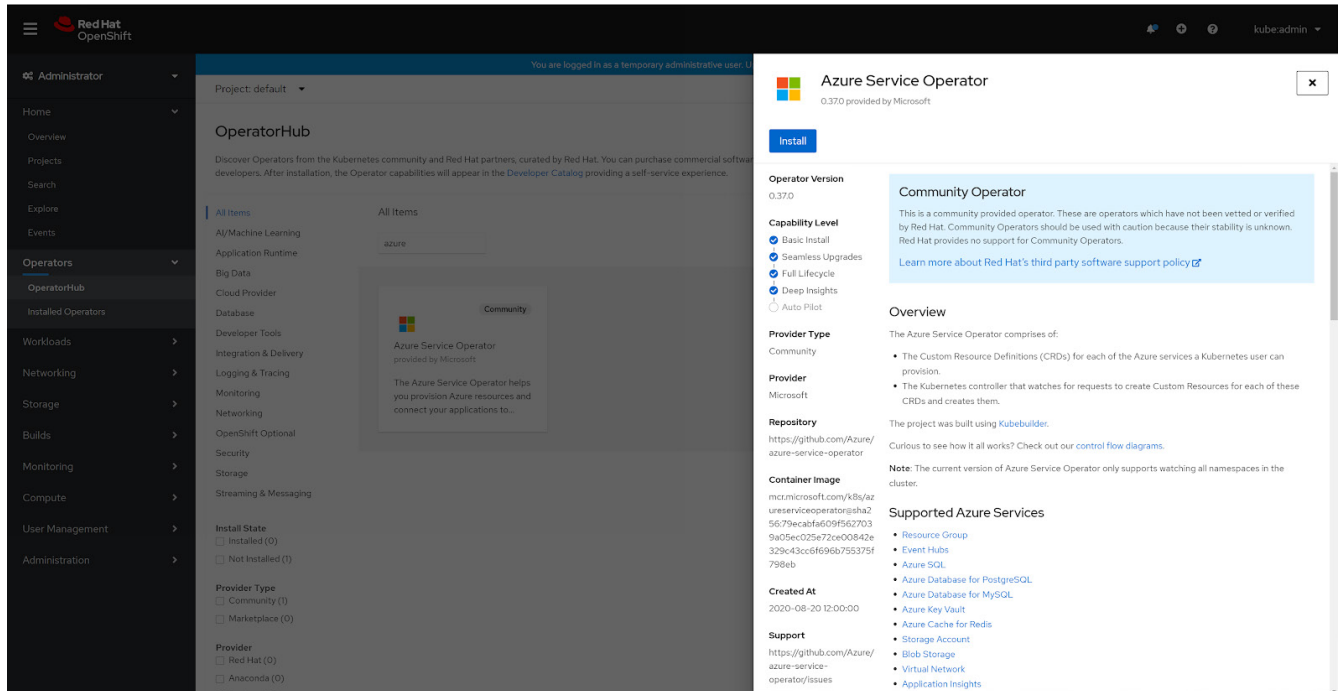


Figure 9.2: The installation screen for Azure Service Operator, with the OperatorHub gallery in the background

Use of Azure Service Operator is not mandatory for services running on Azure, and it is important to understand that the underlying Azure services are deployed natively onto Azure—not onto OpenShift. However, Azure Service Operator makes this quicker and easier for applications that have Azure service dependencies.

A popular use case for Azure Service Operator is to deploy dependent databases, along with the application. For example, for a web application that uses an instance of Azure CosmosDB, deploy Azure CosmosDB with Azure Service Operator as part of deploying the application on OpenShift. Azure Service Operator includes support for Azure SQL Database, Azure Database for MySQL, Azure PostgreSQL, as well as a few others.

Assuming Azure Service Operator is installed, the following Kubernetes manifest could be stored with the OpenShift application and used to provision a MySQL database:

Source: [Taken from the Azure Service Operator samples repository](#)

```
apiVersion: azure.microsoft.com/v1alpha1
kind: MySQLServer
metadata:
  name: aso-wpdemo-mysqlserver
spec:
  location: eastus2
  resourceGroup: aso-wpdemo-rg
  serverVersion: "8.0"
  sslEnforcement: Disabled
  minimalTLSVersion: TLS10 # Possible values include: 'TLS10', 'TLS11', 'TLS12', 'Disabled'
  infrastructureEncryption: Enabled # Possible values include: Enabled, Disabled
  createMode: Default # Possible values include: Default, Replica, PointInTimeRestore (not
  implemented), GeoRestore (not implemented)
  sku:
    name: GP_Gen5_4 # tier + family + cores eg. - B_Gen4_1, GP_Gen5_4
    tier: GeneralPurpose # possible values - 'Basic', 'GeneralPurpose', 'MemoryOptimized'
    family: Gen5
    size: "51200"
    capacity: 4
```

It would be unusual to use Azure Service Operator for services involving many Azure services with complex dependencies, and tools such as Azure ARM templates or Bicep may be more suitable in those cases.

For an extended introduction to Azure Service Operator, check the following blog posts:

- [Blog post – September 2020](#)
- [Azure Service Operator on OperatorHub.io](#)
- [Azure Service Operator on GitHub](#)

Integration with Azure DevOps

Along with OpenShift Pipelines, many users will want to integrate Azure Red Hat OpenShift with Azure DevOps. These solutions can happily co-exist, with some projects using one solution and some another—or sometimes, projects make use of both! The decision about when to use either solution depends on a few factors.

Generally speaking, Azure DevOps offers a high level of integration with other Azure tooling, but can easily deploy to OpenShift, as well as other compute resources.

On the other hand, OpenShift Pipelines is a well-integrated offering that comes with OpenShift and offers a consistent multicloud experience.

Azure DevOps simply treats OpenShift just like any other Kubernetes cluster. Therefore, all standard Kubernetes interfaces and APIs will work as expected.

The screenshot displays an Azure DevOps pipeline run for job #20210523.6. The left pane shows the job's progress, with the 'Deploy' step highlighted. The right pane shows the terminal output for the 'Deploy to Kubernetes cluster' step, which includes the following details:

```

1 Starting: Deploy to Kubernetes cluster
2 =====
3 Task          : Deploy to Kubernetes
4 Description   : Use Kubernetes manifest files to deploy to clusters or even bake the manifest files to be used for deployments using Helm charts
5 Version       : 0.185.0
6 Author        : Microsoft Corporation
7 Help          : https://aka.ms/azpipas-k8s-manifest-tsg
8 =====
9
10              Kubect1 Client Version: v1.20.1-5-g76a04fc
11              Kubect1 Server Version: v1.20.0+75370d3
12 =====
13 /usr/local/bin/kubect1 apply -f /home/vsts/work/_temp/Deployment_web_1621771424182,/home/vsts/work/_temp/Deployment_leaderboard_1621771424182,/home/vsts/work/_temp/
14 deployment.apps/web configured
15 deployment.apps/leaderboard configured
16 service/leaderboard unchanged
17 service/web unchanged
18 route.route.openshift.io/web unchanged
19 /usr/local/bin/kubect1 rollout status Deployment/web --timeout 0s --insecure-skip-tls-verify --namespace stefan-bergstein-stage
20 Waiting for deployment "web" rollout to finish: 1 old replicas are pending termination...
21 Waiting for deployment "web" rollout to finish: 1 old replicas are pending termination...
22 deployment "web" successfully rolled out
23 /usr/local/bin/kubect1 rollout status Deployment/leaderboard --timeout 0s --insecure-skip-tls-verify --namespace stefan-bergstein-stage
24 Waiting for deployment "leaderboard" rollout to finish: 1 old replicas are pending termination...
25 Waiting for deployment "leaderboard" rollout to finish: 1 old replicas are pending termination...
26 deployment "leaderboard" successfully rolled out
27 /usr/local/bin/kubect1 get service/leaderboard -o json --insecure-skip-tls-verify --namespace stefan-bergstein-stage
28 {
29   "apiVersion": "v1",
30   "kind": "Service",
31   "metadata": {
32     "annotations": {
33       "azure-pipelines/jobName": "\Deploy",
34       "azure-pipelines/org": "https://dev.azure.com/stefanbergstein/",
35       "azure-pipelines/pipeline": "\stefan-bergstein.msilearn-tailspin-spacegame-web-kubernetes",
36       "azure-pipelines/pipelineId": "\",
37       "azure-pipelines/project": "Space Game - web - Kubernetes",
38       "azure-pipelines/run": "20210523.5",
39       "azure-pipelines/runurl": "https://dev.azure.com/stefanbergstein/Space Game - web - Kubernetes/_build/results?buildId=28",
40       "kubect1.kubernetes.io/last-applied-configuration": "{\"apiVersion\":\"v1\",\"kind\":\"Service\",\"metadata\":{\"annotations\":{},\"name\":\"leaderboard\"}}",
41     }
42   }
43 }

```

Figure 9.3: An Azure DevOps pipeline pushing content to OpenShift

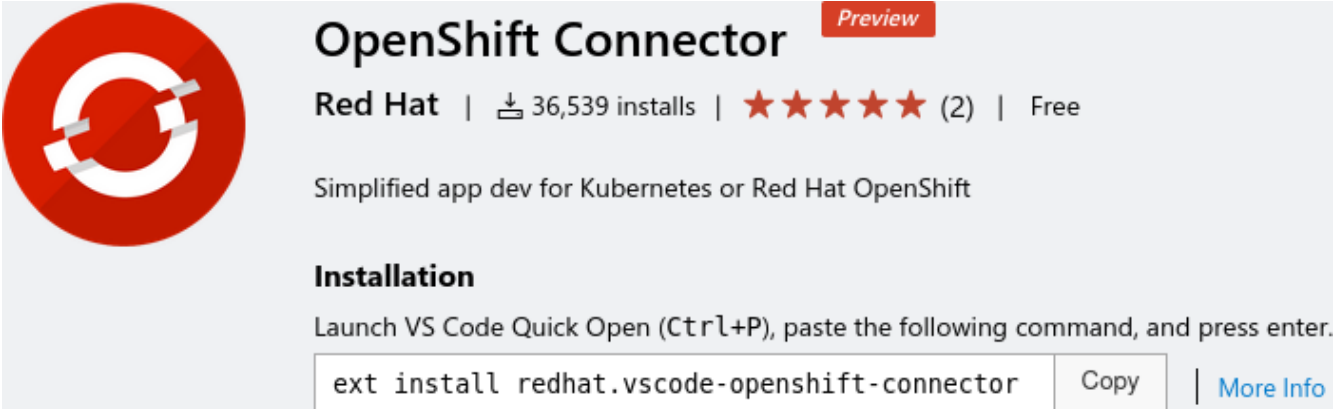
Further reading

The following community blog post offers a great tutorial on how to get started with Azure Red Hat OpenShift and Azure DevOps:

- [Blog post – May 2021](#)

Integration with Visual Studio Code

As part of the value of OpenShift's developer integration, there are plugins for many popular IDEs, including Visual Studio Code. This allows developers and administrators to quickly and easily access Kubernetes and OpenShift resources without having to leave their IDE.



OpenShift Connector Preview

Red Hat | 📄 36,539 installs | ★★★★★ (2) | Free

Simplified app dev for Kubernetes or Red Hat OpenShift

Installation

Launch VS Code Quick Open (Ctrl+P), paste the following command, and press enter.

```
ext install redhat.vscode-openshift-connector
```

[Copy](#) | [More Info](#)

Figure 9.4: Installing OpenShift Connector

Once the connector is downloaded and installed in Visual Studio Code, you'll be prompted to log in to your OpenShift cluster. The following is a simple "hello world"-style project, with a connection to Azure Red Hat OpenShift:

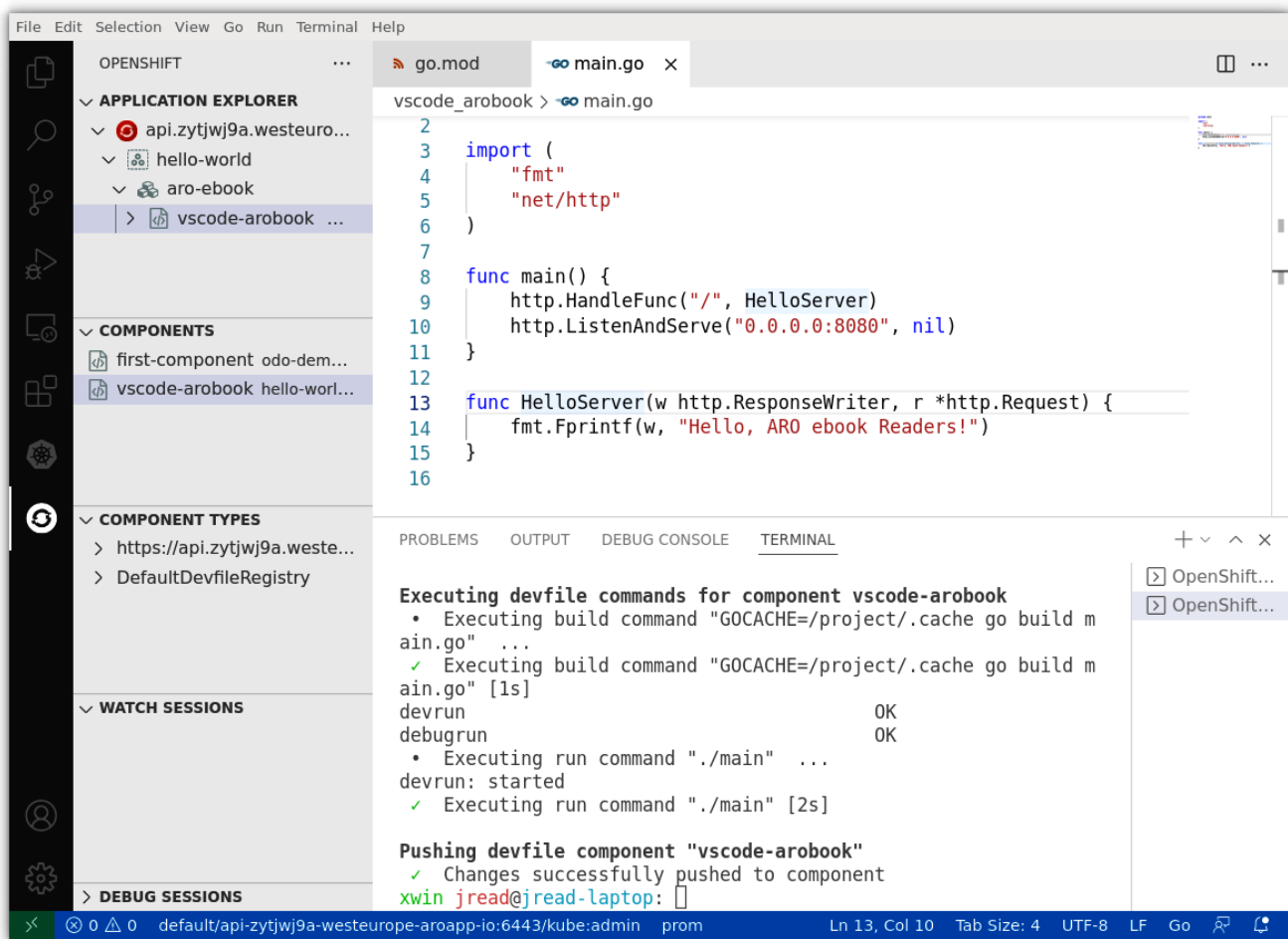


Figure 9.5: A Golang project that has just been deployed with the Visual Studio Code OpenShift connector

One benefit of using the OpenShift connector with Visual Studio Code is the availability of a graphical frontend to the popular OpenShift tool "OpenShift Do"—normally just called "odo." This allows developers to think in higher-level concepts than just raw Kubernetes components. Developers need not worry in so much detail about Deployments, ReplicaSets, and so on. You can see from the preceding screenshot that this is a simple project, with an exposed HTTP service.

Additionally, the connector also includes functionality to push code changes directly to OpenShift, without necessarily having to use source control first. This is really useful for rapid development, rather than having to push to source control every time, build a new container, and get it deployed.

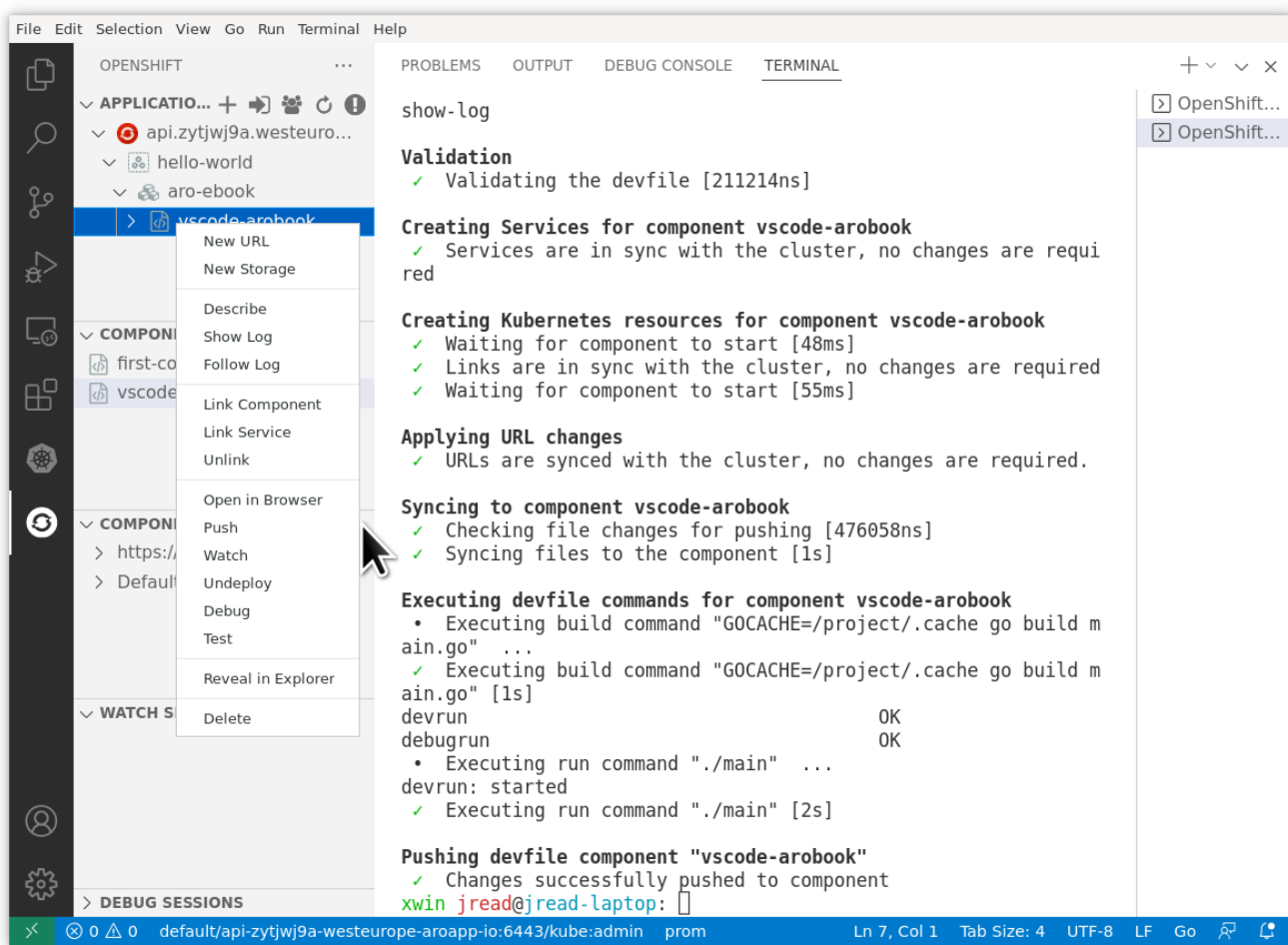


Figure 9.6: The "push" menu on a project, and a recent push in the terminal window

This connector simply speeds up the development and testing cycles for developers who prefer to use Visual Studio Code.



Figure 9.7: The basic Golang application running on OpenShift

Of course, developers are not restricted to just using Visual Studio Code—many developers work with Vim, Eclipse, and other editors—but Visual Studio Code is very popular for those developers who like a "lightweight IDE"-style experience.

Further reading

- [Demo video – using Visual Studio Code with OpenShift](#)
- [Visual Studio Code OpenShift Connector](#)

Integration with GitHub Actions

It is now possible to use GitHub Actions to deploy to any Red Hat OpenShift environment, including Azure Red Hat OpenShift. From any GitHub repository, navigate to **Actions** → **New Workflow** and select **OpenShift** from the list of available actions.

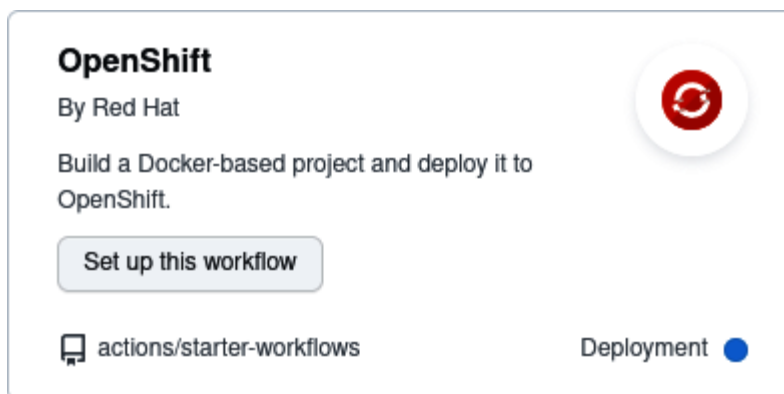


Figure 9.8: Deploying to OpenShift from GitHub

The default template comes with an excellent example set of actions that just needs a few environment variables set to connect to an OpenShift cluster:

```
name: OpenShift

env:
  # ✎ EDIT your repository secrets to log into your OpenShift cluster and set up the context.
  # See https://github.com/redhat-actions/oc-login#readme for how to retrieve these values.
  # To get a permanent token, refer to https://github.com/redhat-actions/oc-login/wiki/Using-a-
  Service-Account-for-GitHub-Actions
  OPENSIFT_SERVER: ${ secrets.OPENSIFT_SERVER }}
  OPENSIFT_TOKEN: ${ secrets.OPENSIFT_TOKEN }}
  # ✎ EDIT to set the kube context's namespace after login. Leave blank to use your user's default
  namespace.
  OPENSIFT_NAMESPACE: ""
  ...
```

An excellent blog post describing how to do so is provided here, along with a demo video:

- [Blog post](#)
- [GitHub Actions with OpenShift demo video](#)

Summary

This chapter covered many of the common integrations that we've seen customers use with Azure Red Hat OpenShift. As stated in the chapter introduction, the standard OpenShift APIs allow integration with many more services not listed in this chapter. A huge number of services can also be easily integrated using the operators listed in OperatorHub.

In the next chapter, we'll share some best practices and recommendations on how to onboard application teams and gain some traction in adopting the service within your organization.

Chapter 10

Onboarding workloads and teams

Once you have run through the *pre-provisioning* steps, provisioned Azure Red Hat OpenShift, and run through the necessary post-provisioning steps, all that is left is to support and onboard developers and applications onto the cluster. How you go about this ultimately depends on the culture of your organization—some developer and application teams are keen to "dive into the deep end" and get started, whereas other teams might appreciate more guidance.

This chapter serves as a set of checklists that provide a good foundation to make sure your developers and applications teams are likely to get up and running more quickly when using Azure Red Hat OpenShift.

Do consider extending these checklists as you adapt them to your organizational structure and culture.

Checklist: Pre-onboarding

Before onboarding a new workload, it is important to be able to convey to the application team, or the owners of that workload, that Azure Red Hat OpenShift has been deployed and designed in such a way that it fits your organization's best practices already:

- Azure Red Hat OpenShift has been deployed into an Azure landing zone or another Azure environment that has been approved for the organization's applications.
- The cluster has all the "Day-2" setup required, such as storage classes and authentication (described in *Chapter 6, Post-provisioning - Day 2*).
- The cluster is already fully configured and supported by your platform team and is backed by the integrated support offered by Red Hat and Microsoft.
- The cluster has been updated to the latest available stable version and is well patched. It will continue to be well patched going forward.

- The Azure Red Hat OpenShift cluster has connectivity to the essential resources required:
 - Connectivity to the on-premises environment, via Azure ExpressRoute or a VPN
 - Connectivity to an enterprise artifact repository (such as a Java Maven repository)
 - Connectivity to the public internet, for downloading dependencies during an application build

Onboarding pilot workloads

A common pattern for rolling out Azure Red Hat OpenShift to an organization is to onboard at least two pilot workloads:

- **The minimal meaningful workload** – Ideally as an SRE team, the first pilot workload is a small, well-known application with very simple technical requirements. This is slightly above a simple "Hello World" application in that the application normally should have a real business owner within the organization. However, with the first workload, the emphasis is on checking that the Azure Red Hat OpenShift cluster can meet the business needs on a cluster level—Azure connectivity, Azure Active Directory login, and to simply identify "low-hanging fruit"—common issues that every application is likely to experience.
- **A meaningful reference workload** – Once a minimal, simple workload has been deployed, it really helps further adoption if the second workload is sufficiently complex and meaningful (important to the business, or even business-critical). This workload will help identify and resolve issues such as connectivity to important databases, performance testing, and logging/metrics at scale. Importantly, this meaningful reference workload can then be used as an **internal reference** within your organization. This will help future development and application teams gain confidence in the Azure Red Hat OpenShift platform.

There are, of course, other patterns of onboarding the first few workloads that might work for your organization. It may be necessary to target applications in a data center that is being decommissioned shortly or target applications that are running on an old Java application server.

Checklist: Onboarding meeting with additional teams

When onboarding a new development or application team onto the cluster, a good practice is to host an onboarding meeting:

- Create a namespace, or a series of namespaces, for the team to use.
- Perform a brief demo of Red Hat OpenShift to the team, pointing out key functionality such as Deploy from GitHub (or equivalent).
- Check that the cluster has enough spare capacity to deploy the target workload (CPU, RAM, storage, and so on).
- Check that members of the team can log in to the cluster—connectivity to Azure Active Directory is a good way to make this easier.
- Provide contact details for the SRE team (or similar) who are managing the cluster.
- Provide a list of OpenShift getting started links:
 - <http://learn.openshift.com>
 - <https://github.com/openshift-labs/starter-guides>
 - <http://docs.openshift.com>

Checklist: Regular health check calls

After a development or application team has begun deploying to the cluster, it often makes sense to have regular health check meetings. These could form part of a daily standup, or simply a 30-minute weekly cadence might be sufficient. Here is a list of things you may want to check for:

- How has the team got on generally— have any applications been deployed?
- Have there been any recent issues in using the cluster (Red Hat OpenShift knowledge or connectivity issues)?
- Have there been any outages in Azure or with the cluster that have negatively affected the experience?
- Reminder about the SRE team availability and contact details for answering questions.

Consider what else you might have used already when having regular health check calls for similar projects. Add things you think might work to this checklist.

Antipattern: Developer playgrounds/sandboxes

A common "antipattern" to encourage developer adoption with an organization is to provide a sandbox type of environment, commonly called a "developer playground," and expect developers and applications to start adopting Azure Red Hat OpenShift. Without any additional follow-up support or resources, Azure Red Hat OpenShift can be an intimidating environment, with a lot of complexity that can be too much to absorb. In most cases, adopting a "sandbox" pattern can lead to wasted cloud compute spend and empty clusters.

However, if your organization does have a history of "sandboxing" with development teams, here are some tips to try and make this as successful as possible:

- Provide at least a short demo of what Azure Red Hat OpenShift can do.
- Provide some documentation and at least the getting started links outlined previously.
- Organize a "hackathon" event, challenging developers to build something with OpenShift in a small competition format.
- Provide an offer of extended support, an introduction to the SRE team, and a more guided onboarding experience as an option.

Following a guided adoption pattern checklist, as outlined previously, is more likely to see the platform be adopted.

Azure Red Hat OpenShift Developer Workshop

The Azure Red Hat OpenShift Developer Workshop (<https://aroworkshop.io>) is a useful pre-created workshop that you can use within your organization to provide a guided, hands-on experience of Azure Red Hat OpenShift. The workshop is split into two lab exercises, both designed to take a developer or an operator who is new to OpenShift through a guided tutorial. Both labs highlight key OpenShift functionality and showcase how it can be used. Lab 1 is an introduction to modern microservices design, and lab 2 goes into the internals of the service.

A useful activity in raising awareness of Azure Red Hat OpenShift within your organization is to use the aroworkshop.io content on your own internal Azure Red Hat OpenShift cluster. In a pre-workshop discussion, you can explain how Azure Red Hat OpenShift has been deployed within your organization's existing Azure Red Hat OpenShift subscription, and that use of the service within the workshop could be a similar experience to using Azure Red Hat OpenShift for hosting their production applications.

Summary

This chapter provided some helpful checklists and guidance to successfully onboard workloads and teams onto Azure Red Hat OpenShift. A common anti-pattern was described—"sandbox" clusters with no support. It is difficult for any guide to explain a complete list of resources and checklist items that would apply to every organization, so it is important to adapt the contents of this chapter to suit your own needs.

The cloud offers an enticing array of services; however, many remain overlooked, or poorly adopted, because organizations focus heavily on the technology and how it is deployed. While understanding those topics is important, they are only a small part of the equation when it comes to taking that service through to production and efficient adoption. This chapter has tried to highlight how training, regular check-ins, and similar activities are necessary to make your adoption of Azure Red Hat OpenShift more successful.

Chapter 11

Conclusion

Your development and operations teams can spend most of their working hours dealing with provisioning, setting up, maintaining, and overseeing your clusters and CI/CD pipeline. When they do, they're not able to dedicate their valuable time to what they do best—keeping your applications at the cutting edge.

As we have learned from this guide, Azure Red Hat OpenShift lets you deploy fully managed Red Hat OpenShift clusters without worrying about building or managing the infrastructure required to run them. We've seen that running Kubernetes alone comes with a few caveats, mainly in relation to the extra hands-on attention required with tasks that could be automated with Azure Red Hat OpenShift.

When you're deciding which cluster management strategy to choose for your organization, consider the pros and cons that you'll be getting with a Kubernetes type of platform versus Azure Red Hat OpenShift, which is built on the Kubernetes framework and offers you a bundle of additional out-of-the-box benefits.

To learn more about Azure Red Hat OpenShift, visit the product page or check out our documentation section. You can also go through a hands-on workshop and register to watch a webinar at your convenience. Most importantly, we hope that you reach out to Microsoft and Red Hat to support you in your evaluation of Azure Red Hat OpenShift.

Authors and versions

James Read <james@redhat.com>

Principal Solution Architect at Red Hat,
covering Microsoft – updated and revised for AROv4

Ahmed Sabbour <asabbour@microsoft.com>

Senior Product Marketing Manager at Microsoft,
covering Azure Red Hat OpenShift – initial version for AROv3

Authors from previous editions

Oren Kashi <okashi@redhat.com>

Senior Principal Technical Product Marketing Manager at Red Hat

Thanks to Brooke Jackson, Nermina Miller, Jose Moreno, Ahmed Sabbour, Aditya Datar, Vince Power, Alex Patterson, and others who kindly offered their time and feedback in reviewing this guide.

Chapter 12

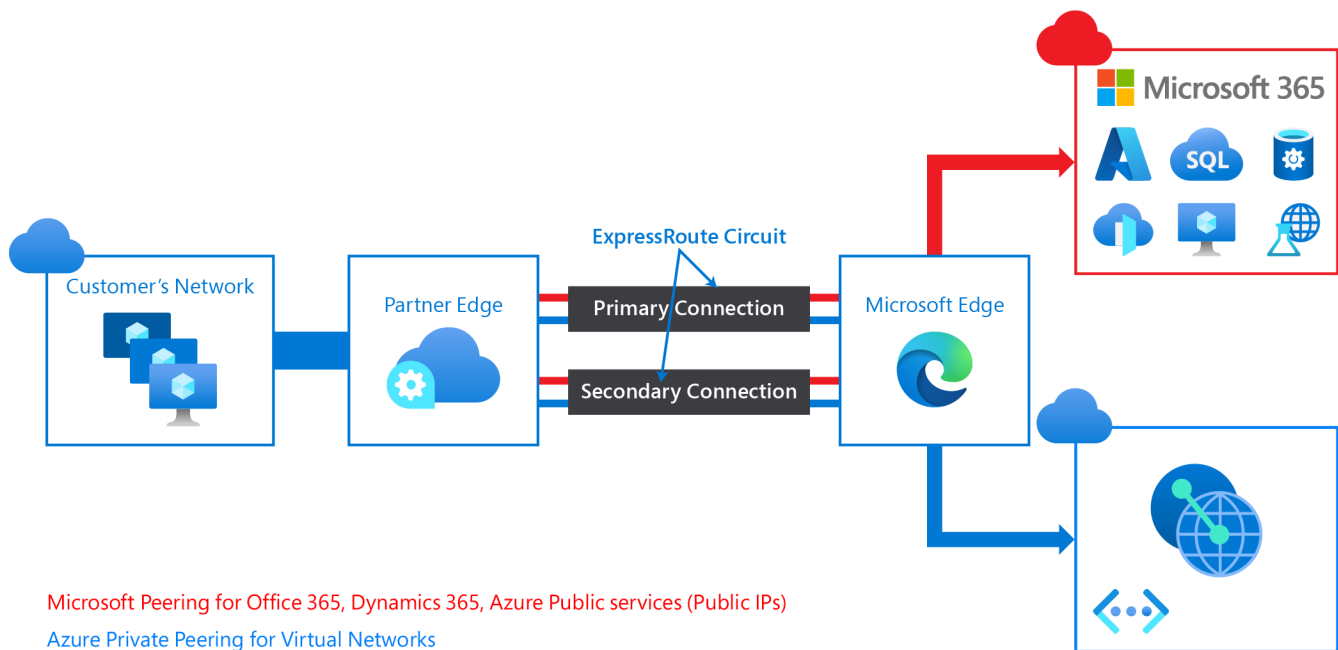
Glossary

This glossary provides a helpful quick reference for some of the terms used in this guide and in the Azure Red Hat OpenShift ecosystem. The headings are ordered alphabetically.

Azure ExpressRoute

ExpressRoute lets you extend your on-premises networks into the Microsoft cloud over a private connection with the help of a connectivity provider. With ExpressRoute, you can establish connections to Microsoft cloud services, such as Microsoft Azure and Microsoft 365.

The following is an ExpressRoute network diagram, taken from the Microsoft Azure documentation:



To read more about ExpressRoute, see the [What is ExpressRoute?](#) page in the Microsoft Azure documentation.

To understand how ExpressRoute works with Azure, see *Chapter 4, Pre-provisioning – enterprise architecture questions*.

Azure Landing Zones

Azure landing zones are a popular deployment pattern for organizations planning a large-scale deployment using Azure, with considerations for scale, security governance, networking, and identity factored in.

[Azure landing zone introduction](#)

Currently in development at the time of writing, there is a community GitHub project that makes some recommendations about how to deploy Azure Red Hat OpenShift into an Azure landing zone architecture: <https://github.com/Azure/Enterprise-Scale/tree/main/workloads/ARO>

Builds and Image Streams

A build is the process of transforming input parameters into a resulting object. Most often, the process is used to transform input parameters or source code into a runnable image. A BuildConfig object is the definition of the entire build process.

Azure Red Hat OpenShift uses Kubernetes by creating Docker-formatted containers from build images and pushing them to a container image registry.

Build objects share common characteristics: inputs for a build, the need to complete a build process, logging the build process, publishing resources from successful builds, and publishing the final status of the build. Builds take advantage of resource restrictions, specifying limitations on resources such as CPU usage, memory usage, and build or pod execution time.

The Azure Red Hat OpenShift build system provides extensible support for build strategies that are based on selectable types specified in the build API. There are three primary build strategies available:

- **Docker build** – using a Dockerfile, which can be uploaded or pulled from a source repository
- **Source-to-Image (S2I) build** – takes a source repository (such as Git) and determines how to build the application from well-known language build files (for example, Maven .pom files for Java projects)
- **Custom build**

By default, Docker builds and S2I builds are supported.

The resulting object of a build depends on the builder used to create it. For Docker and S2I builds, the resulting objects are runnable images. For custom builds, the resulting objects are whatever the builder image author has specified.

Container

The basic units of Azure Red Hat OpenShift applications are called containers. Linux container technologies are lightweight mechanisms for isolating running processes so that they are limited to interacting with only their designated resources.

Many application instances can be running in containers on a single host without visibility into each other's processes, files, network, and so on. Typically, each container provides a single service (often called a "microservice"), such as a web server or a database, though containers can be used for arbitrary workloads.

Container Images

Containers in Azure Red Hat OpenShift are based on Docker-formatted container images. An image is a binary that includes all the requirements for running a single container, as well as metadata describing its needs and capabilities.

You can think of it as a packaging technology. Containers only have access to resources defined in the image unless you give the container additional access when creating it. By deploying the same image in multiple containers across multiple hosts and load balancing between them, Azure Red Hat OpenShift can provide redundancy and horizontal scaling for a service packaged into an image.

Container Registry

Azure Red Hat OpenShift provides an integrated container image registry called **OpenShift Container Registry (OCR)** that adds the ability to automatically provision new image repositories on demand. This provides users with a built-in location for their application builds to push the resulting images.

Whenever a new image is pushed to OCR, the registry notifies Azure Red Hat OpenShift about the new image, passing along all the information about it, such as the namespace, name, and image metadata. Different pieces of Azure Red Hat OpenShift react to new images, creating new builds and deployments.

Azure Red Hat OpenShift can also utilize any server implementing the container image registry API as a source of images, including Docker Hub and Azure Container Registry.

Deployments and Deployment Configurations

Building on replication controllers, Azure Red Hat OpenShift adds expanded support for the software development and deployment life cycle with the concept of deployments. In the simplest case, a deployment just creates a new ReplicationController and lets it start up pods. However, Azure Red Hat OpenShift deployments also provide the ability to transition from an existing deployment of an image to a new one and also define hooks to be run before or after creating the ReplicationController.

The Azure Red Hat OpenShift DeploymentConfig object defines the following details of a deployment:

1. The elements of a ReplicationController definition
2. Triggers for creating a new deployment automatically
3. The strategy for transitioning between deployments
4. Life cycle hooks

Each time a deployment is triggered, whether manually or automatically, a deployer pod manages the deployment (including scaling down the old ReplicationController, scaling up the new one, and running hooks). The deployment pod remains for an indefinite amount of time after it completes the deployment in order to retain its logs of the deployment. When a deployment is superseded by another, the previous ReplicationController is retained to enable easy rollback if needed.

For detailed instructions on how to create and interact with deployments, refer to [Deployments and DeploymentConfigs](#).

Jobs

A job is similar to a ReplicationController, in that its purpose is to create pods for specific reasons. The difference is that replication controllers are designed for pods that will be continuously running, whereas jobs are for one-time pods. A job tracks any successful completions and when the specified amount of completions have been reached, the job itself is completed.

See the *Jobs* topic for more information on how to use jobs.

Pods and Services

Azure Red Hat OpenShift uses the Kubernetes concept of a Pod, which is one or more containers deployed together on one host and is the smallest compute unit that can be defined, deployed, and managed.

Pods are the rough equivalent of a machine instance (physical or virtual) to a container. Each pod is allocated its own internal IP address, therefore owning its entire port space, and containers within pods can share their local storage and networking.

Pods have a life cycle; they are defined, then they are assigned to run on a node, then they run until their container(s) exit or they are removed for some other reason. Pods, depending on the policy and exit code, may be removed after exiting, or may be retained in order to enable access to the logs of their containers.

Azure Red Hat OpenShift treats pods as largely immutable; changes cannot be made to a pod definition while it is running. Azure Red Hat OpenShift implements changes by terminating an existing pod and recreating it with a modified configuration, base image(s), or both. The runtime components of a pod are treated as expendable and are recreated from the defined container image. Therefore, pods should usually be managed by higher-level controllers, rather than directly by users.

Projects and Users

A project is a Kubernetes namespace with additional annotations and is the central vehicle by which access to resources for regular users is managed. A project allows a community of users to organize and manage their content in isolation from other communities. Users must be given access to projects by administrators, or if allowed to create projects, automatically have access to their own projects. Projects can have a separate name, `displayName`, and description.

The mandatory name is a unique identifier for the project and is most visible when using the CLI tools or API. The maximum name length is 63 characters. The optional `displayName` is how the project is displayed in the web console (defaults to `name`). The optional description can be a more detailed description of the project and is also visible in the web console.

Developers and administrators can interact with projects using the CLI or the web console.

ReplicaSets

Similar to a `ReplicationController`, a `ReplicaSet` ensures that a specified number of pod replicas are running at any given time. The difference between a `ReplicaSet` and a `ReplicationController` is that a `ReplicaSet` supports set-based selector requirements whereas a `ReplicationController` only supports equality-based selector requirements.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend-1
  labels:
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - image: openshift/hello-openshift
          name : helloworld
          ports:
            - containerPort: 8080
              protocol: TCP
          restartPolicy: Always
```

In the code above, you can see:

- A label query over a set of resources. The results of `matchLabels` and `matchExpressions` are logically conjoined.
- An equality-based selector to specify resources with labels that match the selector.
- A set-based selector to filter keys. This selects all resources with key equal to `tier` and value equal to `frontend`.

ReplicationController

A [ReplicationController](#) ensures that a specified number of replicas of a pod are running at all times. If pods exit or are deleted, the ReplicationController acts to instantiate more, up to the defined number. Likewise, if there are more running than desired, it deletes as many as necessary to match the defined amount.

A ReplicationController configuration consists of:

- The number of replicas desired (which can be adjusted at runtime).
- A pod definition to use when creating a replicated pod.
- A selector for identifying managed pods.
- A selector is a set of labels assigned to the pods that are managed by the ReplicationController. These labels are included in the pod definition that the ReplicationController instantiates. The ReplicationController uses the selector to determine how many instances of the pod are already running in order to adjust as needed.

The ReplicationController does not perform auto-scaling based on load or traffic, as it does not track either. Rather, this would require its replica count to be adjusted by an external auto-scaler.

A ReplicationController is a core Kubernetes object called ReplicationController. The following is an example ReplicationController definition:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend-1
spec:
  replicas: 1
  selector:
    name: frontend
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - image: openshift/hello-openshift
          name: helloworld
          ports:
            - containerPort: 8080
              protocol: TCP
          restartPolicy: Always
```

In the code above, you can see:

- The number of copies of the pod to run.
- The label selector of the pod to run.
- A template for the pod the controller creates.
- Labels on the pod should include those from the label selector.
- The maximum name length after expanding any parameters is 63 characters.

Routes and Ingresses

Azure Red Hat OpenShift supports both routes and ingresses. Both are used to expose a service using a DNS name, such as www.example.com, so that external clients can reach it.

Routes were originally conceived of in Red Hat OpenShift version 3. Since its release, Red Hat worked with the Kubernetes community to standardize this functionality in what is now called **Ingress**.

The Kubernetes Ingress resource in OpenShift Container Platform implements the Ingress Controller with a shared router service that runs as a pod inside the cluster. The most common way to manage Ingress traffic is with the Ingress Controller. You can scale and replicate this pod like any other regular pod. This router service is based on HAProxy, which is an open-source load balancer solution.

The OpenShift Container Platform route provides Ingress traffic to services in the cluster. Routes provide advanced features that might not be supported by standard Kubernetes Ingress Controllers, such as TLS re-encryption, TLS passthrough, and split traffic for blue-green deployments.

Ingress traffic accesses services in the cluster through a route. Routes and Ingresses are the main resources for handling Ingress traffic. An Ingress provides features similar to a route, such as accepting external requests and delegating them based on the route. However, with an Ingress you can only allow certain types of connections: HTTP/2, HTTPS and **server name identification (SNI)**, and TLS with a certificate. In OpenShift Container Platform, routes are generated to meet the conditions specified by the Ingress resource.

Source-to-Image (S2I)

S2I is a toolkit and workflow for building reproducible container images from source code. S2I produces ready-to-run images by injecting source code into a container image and letting the container prepare that source code for execution. By creating self-assembling builder images, you can version and control your build environments exactly like you use container images to version your runtime environments.

For a dynamic language like Ruby, the build-time and runtime environments are typically the same. Starting with a builder image that describes this environment with Ruby, Bundler, Rake, Apache, GCC, and other packages needed to set up and run a Ruby application installed, S2I performs the following steps:

1. Start a container from the builder image with the application source injected into a known directory.
2. The container process transforms that source code into the appropriate runnable setup—in this case, by installing dependencies with Bundler and moving the source code into a directory where Apache has been preconfigured to look for the Ruby `config.ru` file.
3. Commit the new container and set the image entry point to be a script (provided by the builder image) that will start Apache to host the Ruby application.

For compiled languages like C, Go, or Java, the dependencies necessary for compilation might dramatically outweigh the size of the actual runtime artifacts. To keep runtime images slim, S2I enables multiple-step build processes, where a binary artifact such as an executable or Java WAR file is created in the first builder image, extracted, and injected into a second runtime image that simply places the executable in the correct location for execution.

For example, to create a reproducible build pipeline for Tomcat (the popular Java web server) and Maven, take the following steps:

1. Create a builder image containing OpenJDK and Tomcat that expects to have a WAR file injected.
2. Create a second image that layers Maven and any other standard dependencies on top of the first image, and expects to have a Maven project injected.
3. Invoke S2I using the Java application source and the Maven image to create the desired application WAR.
4. Invoke S2I a second time using the WAR file from the previous step and the initial Tomcat image to create the runtime image.

By placing our build logic inside of images, and by combining the images into multiple steps, we can keep our runtime environment close to our build environment (same JDK, same Tomcat JARs) without requiring build tools to be deployed to production.

The goals and benefits of using S2I as your build strategy are as follows:

- **Reproducibility:** Allow build environments to be tightly versioned by encapsulating them within a container image and defining a simple interface (injected source code) for callers. Reproducible builds are a key requirement for enabling security updates and continuous integration in containerized infrastructure, and builder images help ensure repeatability as well as the ability to swap runtimes.
- **Flexibility:** Any existing build system that can run on Linux can be run inside of a container, and each individual builder can also be part of a larger pipeline. In addition, the scripts that process the application source code can be injected into the builder image, allowing authors to adapt existing images to enable source handling.
- **Speed:** Instead of building multiple layers in a single Dockerfile, S2I encourages authors to represent an application in a single image layer. This saves time during creation and deployment and allows for better control over the output of the final image.
- **Security:** Builds using Dockerfiles are run without many of the normal operational controls of containers, usually running as root and having access to the container network. S2I can be used to control what permissions and privileges are available to the builder image since the build is launched in a single container. In concert with platforms like OpenShift, S2I can enable admins to tightly control what privileges developers have at build time.

Templates

A template describes a set of objects that can be parameterized and processed to produce a list of objects for creation by Azure Red Hat OpenShift. A template can be processed to create anything you have permission to create within a project, for example, services, build configurations, and deployment configurations. A template may also define a set of labels to apply to every object defined in the template.

You can create a list of objects from a template using the CLI or, if a template has been uploaded to your project or the global template library, using the web console. For a curated set of templates, see the OpenShift image streams and templates library.