GETTING STARTED WITH <REDACTED> CLOUD

<REDACTED> 5.0.1

December 2020

Table of contents

Document revision history

This document applies to the release number stated on the title page. A summary of the changes to the document are described in the table below. If there are no changes to the document for a release, no entry is logged.

<REDACTED>

About this publication

This document describes the process necessary to deploy <REDACTED> Switches <REDACTED>, <REDACTED> communications servers, and <REDACTED> components into a Kubernetes environment.

Intended audience

This document is intended to be used by System Administrators and Application specialists at customers of <REDACTED>, as well as Technical Consultants and Solution Consultants representing <REDACTED>customers.

Related publications

For more information, see the following publications:

- <REDACTED>

Terminology

Terms used in the publication:

- **Docker –** Technology and tools to provide application components in discrete images that can be deployed as containers.
- **Elasticsearch** – Datastore with advanced search capabilities.
- **Envoy** – A C++ high-performance distributed proxy for large service-oriented architectures that handles traffic in the service mesh.
- **Grafana** – Dashboarding and visualization tool.
- **Istio** – Service mesh implementation.
- **Jaeger** – Activity tracing tool.
- **Kibana** – Log consolidation and visualization, using Elasticsearch as a storage and query mechanism for log / event messages.
- **Kubernetes** – Container orchestration platform.
- **Prometheus** – Time-series database used to capture and store metrics.
- **Seed** – The default data populated into the system.
- **Feature Preview:** Early availability of a new feature that has not yet fully undergone complete test case and/or system performance verification. A feature preview allows early testing and functionality verification for some projects and is not supported for production.

1 Overview

<REDACTED> Cloud is the cloud-based version of the <REDACTED>. It deploys on a Kubernetes cluster and runs as a set of containers and microservices.

1.1 Microservice architecture and containers

A microservice architecture runs a collection of containers with each container representing one or more microservices. Communication between containers is done through service calls. You can use a container orchestration infrastructure like Kubernetes to start and stop containers. Kubernetes provides elastic scalability which allows additional containers to be started in response to spikes in load and to stop containers when no longer needed.

In addition to the infrastructure for managing containers that Kubernetes provides, a service mesh such as Istio helps manage the services deployed within the containers. A service mesh provides features such as:

- Load balancing
- Staged rollouts (%-based traffic split supporting canary or blue/green deployments for upgrades/changes)
- Secure communications
- Health checks with circuit breaker functionality
- Monitoring

Some of the advantages for running <REDACTED> as a microservice-based architecture with containers include:

- **Elastic Scalability**

You run more instances of the services that need more CPUs and easily spread these instances across multiple servers to scale.

- **Improved mechanism for <REDACTED> cartridge changes**

Today, <REDACTED> changes typically involve deploying a new version of a cartridge (for example, to upgrade <REDACTED> or install a new version of a project cartridge). For systems with several cartridges, applying the change creates additional load on the system and can take time to apply. A cartridge upgrade can also involve changes to the database schema that must be managed by <REDACTED> while the system is running.

With a microservice-based architecture, changes that require new versions of cartridges require that a new version of a container image is built that includes that cartridge. You can deploy this new image in parallel with the running containers, and the service mesh can route a small percentage of traffic to the new version. If everything works, more load is switched to the new version and new instances are started to handle the addition load (while shutting down old versions). Eventually the new version handles all traffic.

Database schema changes that were included in a cartridge upgrade in a traditional <REDACTED> deployment are now in a separate Liquibase changeset. This creates more visibility when schema changes are happening and lets these changes be managed separately from others.
With a microservice based architecture, changes that involve new versions of cartridges are done by building a new version of a container image that includes that cartridge.

Database schema changes:

- **Improve upgrades**

The way that we can support cartridge upgrades with a canary style rollout can also be applied to upgrades of <REDACTED> itself. New container images are built with the new version of the software and deployed using a staged rollout where a controlled percentage of the traffic is routed to the new version and slowly increased as confidence grows in the new version. To perform a rollback, the spread of traffic is controlled so that all traffic flows back to the previous container versions. After all traffic is routed to the new version, the old containers are shutdown.

- **Improve observability**

Along with the use of Kubernetes and Istio, other technologies such as Jaeger, Prometheus and Grafana, and Elasticsearch and Kibana provide a centralized way to monitor and trace activity within the system.

- **Standard technology stack**

By using an industry-standard stack, customers can develop and deploy additional services that can be integrated seamlessly into the platform using standard tools and development methodologies.

1.2 <REDACTED> components and third-party components

A <REDACTED> Cloud deployment consists of several <REDACTED> and third-party components working together to provide a scalable, reliable, manageable, payments processing solution.

The following <REDACTED> components form part of the solution running within a Kubernetes cluster:

- <REDACTED>

<REDACTED> Cloud uses the following third-party components that must also be running in the Kubernetes cluster:

- Istio – The service mesh. Istio uses the high-performance proxy, Envoy. Envoy handles inbound and outbound traffic for services in the service mesh.
- Jaeger – Technology for capturing and viewing traces of service interactions
- Prometheus– Captures metrics of the running system in a time-series database
- Grafana – Provides the dashboards and visualization for the Prometheus metrics
- Elasticsearch – Provides a central location for storing log output
- Kibana – GUI to display log records stored in Elasticsearch
- FluentBit – Captures logs records from containers and ships them to Elasticsearch
- RabbitMQ – Used internally as a message bus for some <REDACTED> components

The following is a diagram showing how the components fit together:

<REDACTED image>

With containerized deployments under Kubernetes the concept of having to install the <REDACTED> components onto servers before running them has changed. Now, the process is to build images and have those images run in a Kubernetes cluster. Kubernetes handles scaling the number of instances and scheduling them for deployment on the set of nodes (servers) that form the cluster.

The third-party components that the <REDACTED> Cloud uses form the *infrastructure*. You can configure this infrastructure and deploy it in the cluster before running the <REDACTED> components, or you can use the <REDACTED>Cloud provided Ansible Playbook to install the infrastructure. This playbook provides an example for how to get these components up and running. It is not necessarily suitable as-is for a production system because it does not consider the customer-specific high availability requirements and other needs such as deploying redundant components that a production deployment might require.

After the infrastructure components are up and running, the Helm tool deploys the <REDACTED> components with the supplied Helm charts. Helm is a standard Kubernetes tool that takes Kubernetes configuration in *charts* and applies this configuration to a running Kubernetes cluster. It can determine what is currently running in Kubernetes and what changes must be made based on the configuration being applied. The <REDACTED> Cloud provided Ansible Playbook that installs the infrastructure components also uses Helm to install many of these infrastructure components.

From a deployment point of view, the above diagram shows which components are deployed by which mechanisms.

1.3 Dual-site

For dual site, you must have two independent installations with the same configuration. Each site will have a Kubernetes cluster, a <REDACTED> installation, and its own third-party applications installed such as Prometheus, ELK stack, and Jaeger. You must monitor each site for alarms and logs. The database replication is the only cross-site communication. Each site must be monitored independently using separate installs for the infrastructure components.

As of this writing, Oracle GoldenGate is the only supported database replication technology. Additionally, you can use Oracle RAC for database availability within the site. See the *<REDACTED>* for more information.

<REDACTED image>

<REDACTED content - now showing pieces of content as examples>

1.4 Docker registries

To deploy and run applications in a Kubernetes cluster, each node of the cluster must have access to a docker registry from where it can pull the built images. If you have a single node cluster you can use the built-in docker registry into which your images will automatically be built. If there are multiple nodes, the simplest approach is to run a docker registry in the Kubernetes cluster and push the built images into that registry. Also, to deploy infrastructure components into the cluster, you might need a private registry that has all necessary external images.

To help with the setup of a docker registry running as a container in Kubernetes, the <REDACTED> Cloud project includes a script to install (and one to uninstall) a docker registry and set up the appropriate authentication so that Kubernetes can pull images from the registry.

See “Set up docker registries” for more information about how to get a docker registry running in a Kubernetes cluster.

<REDACTED content - now showing pieces of content as examples>

1.5 Clean install

This section describes the steps to do if this is the first time you are installing (that is, not upgrading from a previous version).

- The Vagrantfile in the `<redacted>\setup` directory contains the instructions to create and provision the Kubernetes VM. The VM is currently set to use 6 vCPUs and 20GB of memory. You might have to adjust this based on the specifications of the machine, but the system requires a minimum of 16GB memory for initialization.
- Initialize the Vagrant VM from the environment where your VirtualBox drivers are resident (typically a Windows system). This usually takes about 15 - 20 minutes to complete depending on network speed.

```
cd setup
vagrant up
```

- The Ansible variables must be prepared for the machine environment.
  - `config_dir` - Common location for Kubernetes setup and generated configuration
  - `external_host_ip_addr` - Externally accessible IP address of the Kubernetes control node
  - `ingress_host` - The host name for the host name in the cert. To avoid modifying the `/etc/hosts` file on local development environments, <REDACTED> has been using the ***nip.io*** service to do the DNS mapping for us.
  - `<REDACTED>_registry_hostname` - Host where registry for application images resides
  - `registry_port_<REDACTED>` - Port number of application image registry
  - `docker_<REDACTED>_registry_username` - User for the application image registry
  - docker_`<REDACTED> _registry_password` - Password for the application image registry

If you use the <REDACTED> Cloud provided Ansible playbook to install the infrastructure components (like Elasticsearch and Prometheus), then there is a separate playbook that you must run first. This playbook installs things such as Python modules for Kubernetes that the Ansible playbook uses for the infrastructure components. This additional playbook requires super user privileges so you can inspect the playbook and install the components yourself if you would prefer.

Note: For Docker to access secure private docker registries, you can install certificate chains into `/etc/docker/certs.d` or install a `daemon.json` file into `/etc/docker` that lists the registries to access insecurely. The `pre_requisites.yml` playbook will install a supplied `daemon.json` file if it is put into the filesystem of the Vagrant VM at `/home/vagrant/install/playbooks/files`.

<REDACTED content - now showing pieces of content as examples>

1.6 Define secrets

You must define Kubernetes Secrets for the following functions:

- TLS access into the cluster via the Istio Ingressgateway
- imagePullSecrets for access to Docker registries from within the Kubernetes cluster
- Secrets that define environment variables for the database (such as GoldenGate), keystore, and truststore passwords

2 Kubernetes Cluster

2.1 Prerequisites

<REDACTED>

2.1.1 Helm

The Helm package manager for Kubernetes version 3.1.2 or newer is required to deploy <REDACTED> cloud-native applications. Go to `setup/scripts/installHelm.sh` in the `<REDACTED>-cloud` deliverable to find the installation script for Helm**.**

2.2 Docker registries

To deploy and run applications in a Kubernetes cluster, each node of the cluster must have access to a docker registry from where it can pull the built images. If you have a single node cluster you can use the built-in docker registry into which your images will automatically be built. If there are multiple nodes, the simplest approach is to run a docker registry in the Kubernetes cluster and push the built images into that registry. Also, to deploy infrastructure components into the cluster, you might need a private registry that has all necessary external images.

To help with the setup of a docker registry running as a container in Kubernetes, the <REDACTED> Cloud project includes a script to install (and one to uninstall) a docker registry and set up the appropriate authentication so that Kubernetes can pull images from the registry.

See “Set up docker registries” for more information about how to get a docker registry running in a Kubernetes cluster.

<REDACTED>

2.3 Build images

Now, you must build <REDACTED> images. There are the following sets of <REDACTED> images:

- Base images
- Demo/Project images

When building images, you must know where they are going to reside and how your Kubernetes is going to access the images. When building images with Docker, Docker will build the images and store them locally. Kubernetes, running on that same host, can access the local Docker store but if your Kubernetes cluster has more than one node then you must confirm that all nodes can access a common Docker Registry where your images will reside. After you build your images locally, you must push the images from the locally built store into a remote Docker Registry and to configure your Kubernetes cluster to have access to this registry.

For all the nodes in your Kubernetes cluster to see the images, you must tag and push the images from your local Docker store to a remote Docker Registry. If you already have such a registry set up (for example you might be using Nexus) then you must configure your Kubernetes cluster to have access to this registry and to specify the `host:port` for the registry when tagging and pushing the images.

Note: An alternative to using an external Docker Registry is to run a Docker Registry within your Kubernetes Cluster and configure the nodes to all have access. The <REDACTED> Cloud provides an example script to install a Docker Registry into your Kubernetes Cluster (although it assumes that it is a cluster with a single master node). See the instructions in the “Set up docker registries” appendix for how to run the provided installation script.