UNIVERSITÀ DI PISA

Department of Computer Science

Master of Science in Computer Science

Design, prototyping, and validation of a Kubernetes operator for an IoT platform based on Red Hat OpenShift

Supervisors

Antonio Brogi

Jacopo Soldani

Matteo Bogo

Candidate

Aldo D'Aquino

Academic year 2019/2020

Abstract

Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components. The thesis discusses the design, prototyping and validation of a Kubernetes Operator automating the management of a microservice-based platform for managing industrial IoT installations, called "Iotsnap" and developed by Extra Red. The IotSnap platform consists of more than 35 microservices for the provisioning, data ingestion, business intelligence, and monitoring of an industrial IoT installation. The presented Kubernetes Operator automates the deployment and management of a portion of the IotSnap platform on the PaaS Red Hat OpenShift. In addition, the proposed Kubernetes Operator provides features for backup, monitoring, and for enforcing fault-resilience of the target portion of the IotSnap platform. Indeed, backup and monitoring complete the operator, automating two management operations that are currently being carried out manually. The operator has been tested for fault-resiliency by exploiting the fault-aware management protocols, through the modeling of the IotSnap microservices in the formal TOSCA specification and the use of the Barrel tool.

Dedication

To my family, my girlfriend, and friends, but also teachers and colleagues that supported and beared me during this beautiful journey.

Table of Contents

Chapter 1

Introduction

1.1 Context

Containers and their orchestration are taking on an increasingly fundamental role in the implementation and management of application stacks with microservices [61]. Kubernetes is gaining more and more momentum as a solution for orchestrating container in cloud-based environments, and in recent years its growth has led to increasingly complex and advanced features [56].

With increasing complexity in the orchestration of microservices, more and more applications require very careful management of the deployment and life cycle. Among the most delicate tasks we find, for example, the scaling complexity and the consistency between stateful applications, and all day-2 operations, e.g., images and configurations upgrades at runtime, backup, and monitoring. Kubernetes Operators have been proposed and released precisely to tackle these issues. Kubernetes Operators indeed provide a Kubernetes-native solution for deploying and managing applications or application stacks by defining custom resources that can be deployed on Kubernetes as native objects. For example, through the use of operators and custom resources, a database can become a Kubernetes object, and it can be created and configured with the same simplicity as a Pod or a Service.

The Kubernetes Operators are divided into five levels according to the features offered. The first level operators only deal with the application deployment, the patches and minor upgrades are managed at the second level, while a third level operator is also able

to manage the life cycle and backup of the application that is automating. The fourth level is achieved by providing monitoring features, while an operator can be considered fifth level if all the operations that can be done on the application it manages have been automated, totally excluding the need for manual intervention.

1.2 Objective of the Thesis

We carried out this thesis during a 600-hour internship at the Extra Red company in Pontedera. The aim of the internship was to design, prototype, and validate a Kubernetes Operator for automating the deployment and management of the microservices forming "IotSnap", a platform managing industrial IoT installations developed by Extra Red. The IotSnap platform is divided in five areas devoted to data ingestion, provisioning of the edge and IoT devices, monitoring of the IoT industrial installation, administration of devices and cloud infrastructure and interoperability with third party clouds and applications.

The operator must fit into an automation context of the IotSnap cloud platform, and its role is to handle the deployment and manage the IotSnap Root subsystem of IotSnap. IotSnap Root contains the IotSnap microservices devoted to software provisioning and platform administration. It consists of seven microservices, that include two databases, an object storage and four Java microservices providing APIs and implementing the business logic.

During the internship we designed, developed, and validated a prototype that fully exploits the operators' features and reaches the fifth level of the operators. Our prototype of Kubernetes Operator deploys and manages the identified portion of the platform, implements backup and monitoring, and features a full lifecycle automation, including configuration tuning and fault-recovery. The operator has also been validated using the formal framework given by management protocols [10].

1.3 Contributions of the Thesis

During the internship at Extra Red we successfully developed the requested operator, called IotSnap Root Operator. The operator has achieved the required objective: It

creates and manages all the Kubernetes and OpenShift objects necessary in an IotSnap Root installation. It performs the full deployment of the IotSnap Root subsystem, relieving current automation from this task. It takes care of relationships between components by tuning the configurations and their communication and verifies the success of the deployment step by step. In addition, it is capable of managing the component lifecycle, by recovering from faults and crashes, restoring default application settings, handling the scaling, and upgrading components configuration at runtime. The fault recovery capability is supported with a backup feature, which creates a copy of the application data that is compressed, encrypted, and stored in an S3 bucket. Moreover, the IotSnap Root Operator provide monitoring features. The monitoring components retrieve application metrics data from the deployed applications and store it inside an InfluxDB database for further analysis and automated alerts. Thanks to these capabilities, the operator has reached the fifth and last level described from the Operator Framework by exploring all the features provided by this automation approach that were suitable in this context.

The IotSnap Root Operator has been validated through the use of tests and with formal analysis techniques. We developed both unit tests to verify the correctness of the code and end-to-end tests to test the behavior of the operator at runtime, verifying the ability to deploy and manage the IotSnap Root components on a multi-host cluster, in addition to the emulated environment. The management protocols allowed us to perform a formal analysis to verify that the behavior of the operator respects the expected components management. We verified the operator's ability to restore the system both in case of unexpected behavior of the applications (i.e., the crashes) and the consequent faults that the crashes may entail.

1.4 Document Structure

The document is organized as follows.

- In Chapter 2 we first recall some background information on operating-system-level virtualization. We introduce how Kubernetes works and how OpenShift extends Kubernetes. Then, we present Kubernetes Operators and the Operator Framework. Finally, we illustrate the fault-aware management protocols and the Barrel tool that

exploits them.

- Chapter 3 provides the context and the requirements of the IotSnap Root Operator, and describes the design of the operator we created. Also, it discusses how we have exploited management protocols for planning the backup features.
- Chapter 4 illustrates the implementation and structure of the IotSnap Root Operator, showing in particular how we create a new custom resource through APIs and how we manage it through a controller.
- In Chapter 5 we describe the unit and end-to-end tests for the operator and how we have validated it using the fault-aware management protocols.
- Finally, Chapter 6 is devoted to conclusions, providing a critical look also to the limitations of the operator, its approach in solving the Extra Red problems, and the possible futures implementations and extensions to the IotSnap Root Operator.

Chapter 2

Background

2.1 Containerization

Operating-system-level virtualization, also known as containerization, refers to an operating system feature in which the kernel allows the existence of multiple isolated user-space instances, called containers.

A program running inside a container can only see the container's resources i.e., connected devices, also known as volumes; files and folders; network; container's OS and architecture; CPU and memory. This may suggest that containers are similar to virtual machines, but, unlike virtual machines, containerization allows applications to use the same Linux kernel as the system they are running on, instead of creating an entire virtual operating system. On Unix-like operating systems, this feature can be seen as an advanced implementation of the standard chroot mechanism, which changes the apparent root folder for the current running process and its children.

The operating-system-level virtualization has become popular in the latest years, but is an old concept: The chroot mechanism was released with the Seventh Edition Unix, in 1979 [63]. Then this feature was extended with FreeBSD Jails, introduced in FreeBSD v4.0 [39], released in March 2000 [29]. Over the next 9 years, FreeBSD Jails added features like CPU and memory limits, disk space and file number limits, process limits, and networking with multi-IP features [39]. In 2001, Linux-VServer [41] was proposed to create VPS (Virtual Private Server) and isolated web hosting spaces [42].

At the PyCon US 2013 Solomon Hykes presented Docker as the future of containers

[36]. Docker is a Linux-based platform for developing, shipping, and running applications through container-based virtualization. As for the systems presented before, it exploits the kernel of the operating system of a host to run multiple isolated user-space instances, that in Docker are called containers. Docker raised fast, rapidly becoming the de-facto standard for cloud containerization [56].

Only 7 month after the Docker launch, Docker and Red Hat announced major alliance, including Fedora/RHEL compatibility, and started using Docker as container standard within Red Hat OpenShift [17]. The following year Kubernetes was born, and in 2015 was adopted as base for a totally redesigned OpenShift 3.0 [23].

2.1.1 How Docker Containers Work

Each Docker container packages the applications to run, along with whatever software support they need (e.g., libraries, binaries, etc.). This is done by packaging everything in a so-called Docker Image, which can then be instantiated to obtain Docker containers. Docker images are made by layers. An image can be created from an empty one (a special image called "scratch") or on top of other existing images. For example, we can take the Ubuntu image and start copying files inside it, or execute commandas like `apt install`. Each operation run across an image creates a new layer. This allow to reuse common layers between images, saving up space. Existing Docker images are distributed through so-called Docker registries, like the official registry Docker Hub, the OpenShift registry Quay, or other private and public registries.

Docker containers are volatile, and the data produced by a container is by default lost when the container is stopped. This is why Docker introduces volumes, which are specially-designated directories (within one or more containers) whose purpose is to persist data, independently of the lifecycle of the containers mounting them. Docker never automatically deletes volumes when a container is removed, nor it removes volumes that are no longer referenced by any container. Kubernetes (and therefore OpenShift) allows to have distributed volumes, detaching the storage from the compute part.

Docker allows containers to intercommunicate. It permits creating virtual networks, which span from bridge networks (for single hosts), to complex overlay networks (for clusters of hosts). In Kubernetes for example, we have networks across microservices

distributed on many nodes, that allow applications to intercommunicate transparently.

2.2 Kubernetes

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications [2]. It groups containers that make up an application into logical units for easing their management and discovery.

Kubernetes was originally built as internal container orchestration tool at Google, under the name Borg, then it was open-sourced in June 2014 [47] as Kubernetes (meaning helmsan, in Greek [46]). Although Borg was originally written in C++, Kubernetes is implemented in Go, a statically typed, compiled programming language designed at Google [58]. As a consequence, Go is the language of most of the Kubernetes extensions, including operators.

2.2.1 Cluster Architecture

Even if Kubernetes can run on a single physical node, e.g., Kubernetes Minikube and OpenShift CloudReady Containers allow to run a single-node Kubernetes installation locally, it is usually intended to run on a cluster of multiple hosts.

A Kubernetes Cluster is the set of all the machines that compose a Kubernetes installation. Machines in a Cluster are divided in a Master and some Nodes (Figure 2.1).

Node

A host in a Kubernetes Cluster is called node, and can be either a VM or a physical machine. A node is associated with the following information [3].

- **Address**, composed by the *hostname*, *externalIP* of the machine and *internalIP* in the cluster.
- **Capacity and Allocatable**, that describe the CPU, Memory, and storage bounds of the node, as well as how many processes can be deployed on it.
- **Conditions**, like the *Ready* state, a flag *NetworkUnavailable* and some information about the *pressure* on the node: *MemoryPressure* alerts for a high level of Memory

occupied, *DiskPressure* for the storage and *PIDPressure* for too many processes allocated on this node. The conditions' flags are updated accordingly to the current status and the Capacity and Allocatable of the node.

- **Info** describes general information about the node, such as kernel version, Kubernetes version, Docker version (if used), and OS name.

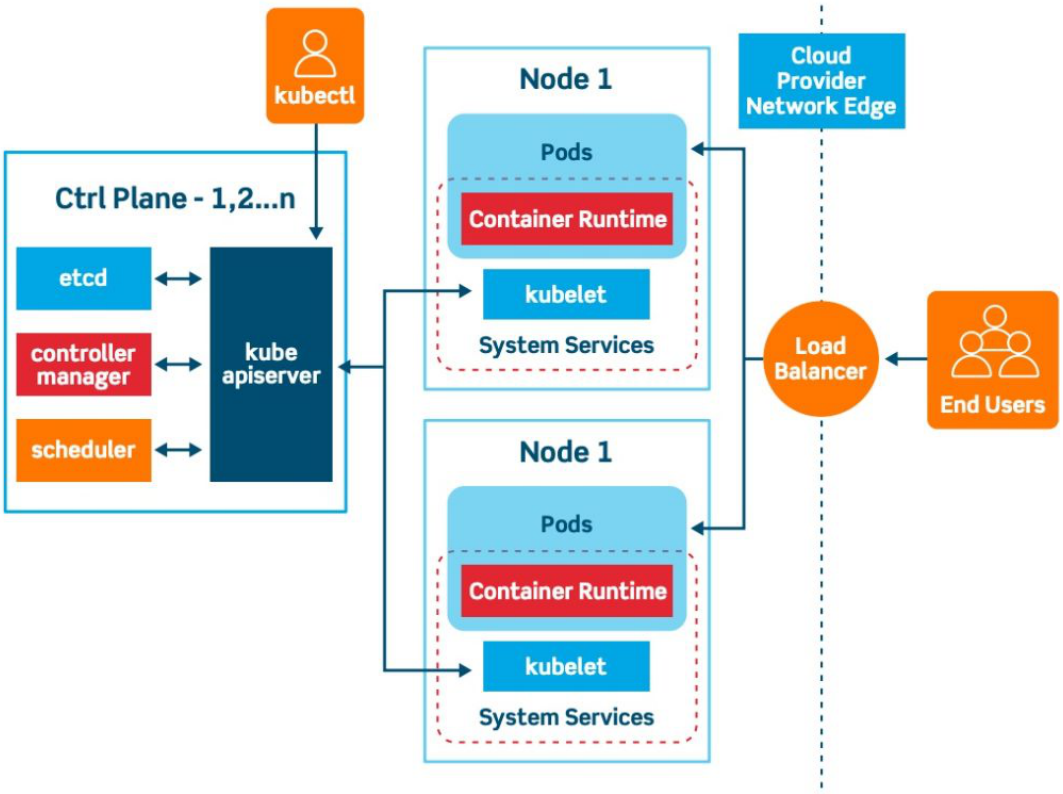


Figure 2.1: Kubernetes architecture diagram [59].

Master

Among the nodes in a Kubernetes cluster, one is elected to be the Master node, or master server. This server acts as a gateway and brain for the cluster by exposing APIs to interact with Kubernetes. The Master is the entrypoint for the Kubernetes Cluster, and is responsible for most of the centralized logic Kubernetes provides. It establish how to schedule tasks, i.e., how to split up and assign work to nodes. It also collect logs and

health status from the nodes. The Master node is usually dedicated to orchestrate the Cluster, while the nodes runs the containers.

In High-Availability Clusters, multiple Masters nodes are instantiated to achieve redundancy, with the aim of ensuring the cluster operation also in case of a Master node failure [4].

Namespace

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called Namespaces. Namespaces are intended for use in environments with many users spread across multiple teams, or projects. They are a way to divide cluster resources between multiple users via resource quota. Namespaces provide also a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. That allows teams to deploy resources without having to coordinating themselves with other teams to assign unique names to deployed resources.

2.2.2 Control Plane

The Kubernetes Control Plane consists of the various components that run on the master nodes (Figure 2.2).

etcd

etcd is crucial for Kubernetes to work across nodes, as it provides a lightweight and distributed key-value store that can span across multiple nodes. Kubernetes uses etcd to store configuration data that can be accessed by each of the nodes in the cluster. It is usually installed on the master node or, in production systems and High-Availability Clusters, on multiple masters for achieving redundancy and resiliency.

kube-apiserver

The Kubernetes API server is the main management point of the entire cluster: All the management tools, including the Kubernetes CLI kubectl, communicates with the Kubernetes installation through this APIs. kubectl is the default method of interacting

with the Kubernetes cluster from a local computer, allowing to administrate the cluster and deploy and manage the Kubernetes objects.

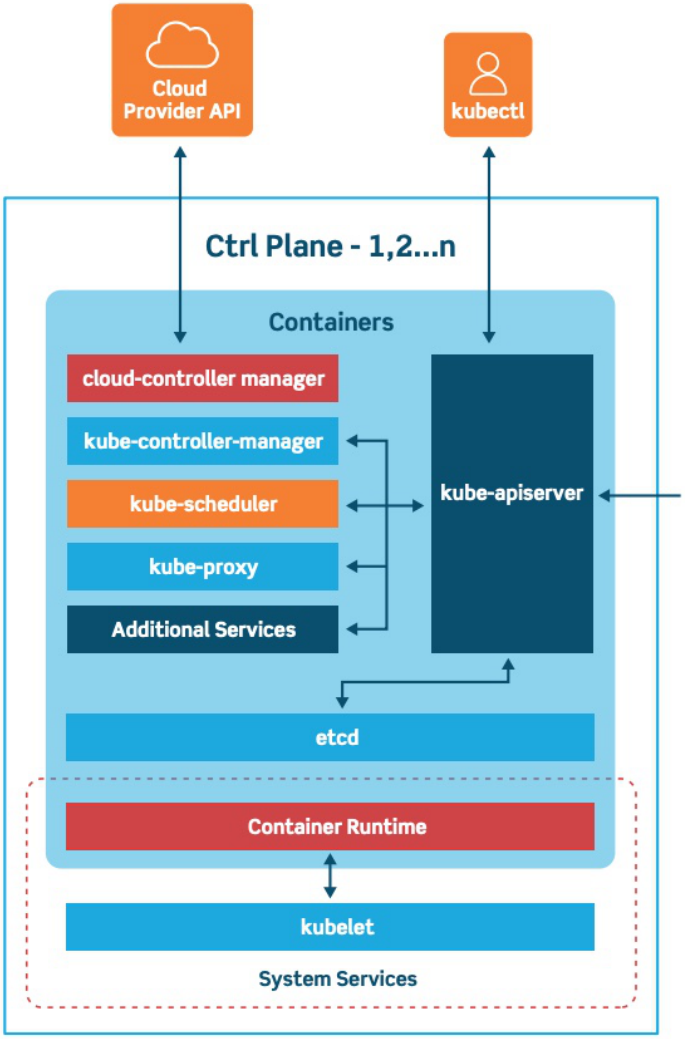


Figure 2.2: Kubernetes Control Plane diagram [59].

kube-scheduler

kube-scheduler is a service that assigns workloads to nodes by analyzing the current infrastructure environment. For this reason, it also tracks available capacity on each host to make sure that workloads does not exceed the resources available neither on a node or on the entire cluster [21].

kube-controller-manager

The controller manager is a general service with many responsibilities, which can hence be seen as a collection of controller components. Each of this controllers regulate the state of the cluster, manage workload life cycles, or perform routine tasks [59].

When a change is detected, the controller reads the new information and implements the procedure that fulfills the desired state. This can involve scaling an application up or down, adjusting endpoints, etc. For instance, a replication controller ensures that the number of replicas defined for an application matches the number currently deployed on the cluster.

cloud-controller-manager

Kubernetes can be deployed in many different environments, e.g., bare metal servers, Amazon Web Services, Google Kubernetes Engine, Azure, just to name a few. Each of this service has different APIs and implementation of the Kubernetes objects. Kubernetes must interact with this variety of infrastructure and cloud providers to map their non-homogeneous resources to its resources abstraction. The Cloud Controller Manager acts as the glue between Kubernetes and the underlying infrastructure. It tells Kubernetes how to interact with the target infrastructure different capabilities, features, and APIs. Since a Kubernetes installation may spread across multiple environments at the same time, there can be multiple CCM in a single cluster, one for each environment.

2.2.3 Node Server Components

While the Kubernetes Control Plan is deployed on the master server, some components must be installed also on the nodes (Figure 2.3) to allow them to take part of a Kubernetes installation.

Container Runtime

The container runtime is responsible for starting and managing containers. Docker is a typical container runtime, but cloud provider can offer provider-specific container runtimes to satisfy this component requirement.

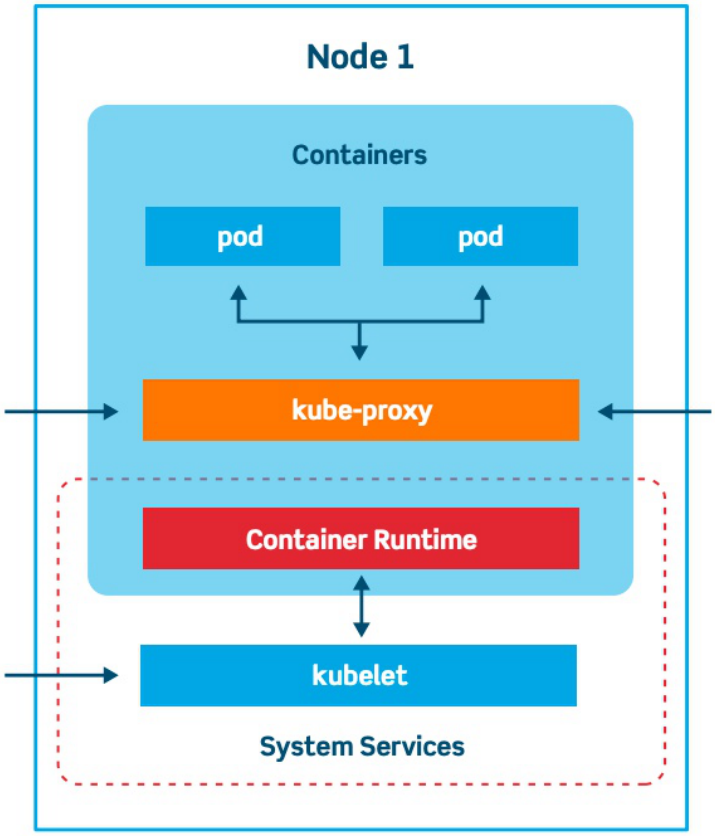


Figure 2.3: Kubernetes Node Server Components diagram [59].

kube-proxy

The Kubernetes Proxy is responsible to create networks between nodes and allows to join the Kubernetes cluster receiving traffic. It also forwards incoming requests to the correct container, also implementing some basic form of load balancing. Finally, it ensures the correct isolation of each network environment.

kubelet

Kubelet is responsible for exchanging information with the Control Plane services, as well as read and write configurations in the etcd store. It receives commands and work from the master components. Work is received in the form of a manifest which defines the resources to be deployed and the operating parameters. After receiving a bunch of work, kubelet assumes the responsibility of executing it and of maintaining the associated resources on the node server, with everything done by interacting with the container runtime.

2.2.4 Kubernetes Objects

Kubernetes defines a set of building blocks that provide mechanisms to deploy, maintain, and scale applications. These primitives can be extended through the Kubernetes API, that allows to implement Kubernetes Operators. The tools that benefit this Kubernetes extensibility can run as Kubernetes container, facilitating the deploying of the tool.

Spec and Status

Almost every Kubernetes object includes two nested object fields that govern the object's configuration, i.e., the object `spec` and the object `status`. The `spec` describes the desired state for an object, while its `status` describes the current state (monitored by Kubernetes and its components). Kubernetes continuously operates on objects with the goal of eventually having that the current `status` matches the desired state give in the `spec`.

Pod

A Pod is the basic deployment unit in Kubernetes. Everything in the container's world has a sea recall. A pod is a social group of whales [3]. This simile is not casual: The Kubernetes Pod is an abstraction for a group of containerized components. One or more tightly coupled containers are encapsulated in an object that guarantees they are co-located and co-scheduled on the same host machine and can share resources.

Containers in a Pod operate closely together, share a life cycle, and should always be scheduled on the same node. Kubernetes manage them as a single unit, i.e., they share their environment, volumes, and IP space. All the containers in a Pod can reference each other on localhost, and because of this, applications in a Pod must coordinate their usage of ports. The exposed port of a container are exposed also from the Pod. From outside the container, there is no way to communicate with a specific container of a Pod, we have instead communication with the Pod, that reaches a container accordingly to the port.

Like containers, Pods are considered to be relatively ephemeral entities, rather than durable. The Pod's unique ID (UID) and IP remain until the Pod termination. The termination may occur when the user send the termination command (with a grace period), when the grace period ends, when a new version is deployed and the old Pod is considered

dead after the transition, or when the list of commands inside the Pod terminates [3].

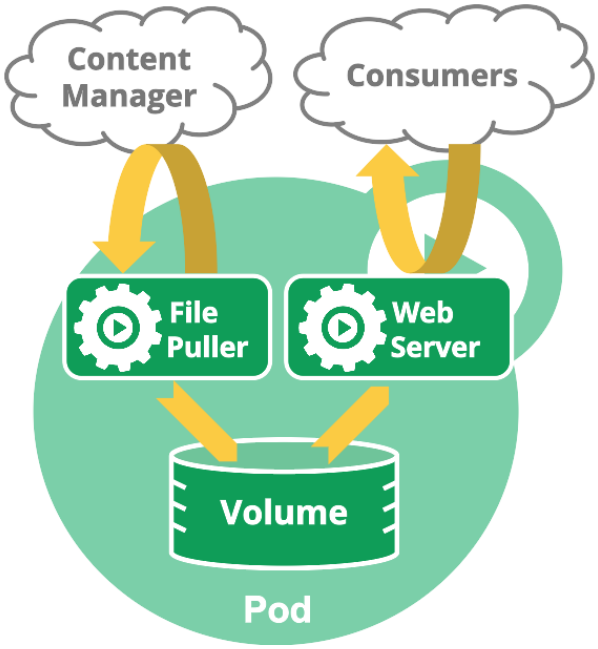


Figure 2.4: Diagram of a multi-container Pod that contains a file puller and a web server that uses a persistent volume for shared storage between the containers.

Usually, Pods consist of a main container that satisfies the general purpose of the workload and optionally some helper containers, called sidecars, that facilitate closely related tasks. These are programs that benefit from being run and managed in their own containers, but are tightly tied to the main application [21]. For example, a periodic task to be run in background, like the File Puller in Figure 2.4.

Running multiple containers in a Pod is hence quite different from running multiple programs in a single container, i.e., it allows to reach all the advantages of having multiple programs on localhost, while obtaining transparency, decoupling of software dependencies, efficiency, and ease of use. Horizontal scaling instead should not be implemented inside a Pod scaling containers but outside, scaling Pods [3].

Volume

Kubernetes Volumes are an abstraction for the storage. On-disk files in a Container are ephemeral, because they die together with the container and when a new one is deployed it starts with a clean state, from the original image. This may cause issues for non-

trivial applications, e.g., a storage system or a database. Moreover, sometimes containers running together in a Pod need to share files. The Volume abstraction solves both of these problems.

Volumes are however themselves fully persistent. When a Pod is removed, the associated Volumes are destroyed with it. This is avoided by Kubernetes Persistent Volumes, which are a mechanism for abstracting more robust storage that is not tied to the Pod lifecycle [3]. Persistent Volumes are separate storage resources for the cluster that can be attached to Pods via `persistentVolumeClaim`. If the Pod is removed, the Persistent Volume is released without being deleted. This allows, for example, to reattach the previous storage to a crashed Pod, restoring its data.

Replica Set

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods [3]. A ReplicaSet fulfills its purpose by creating and deleting Pods as needed to reach the desired number. When a ReplicaSet needs to create new Pods, it uses its Pod template, that is part of the ReplicaSet specification.

All the Pods deployed (or subsequently acquired) by a ReplicaSet are linked to it with their `ownerReferences` field. This allows the ReplicaSet to know how many and which Pods it is maintaining, to create a new Pod, or delete one of the existent accordingly. If there is a Pod that has no `ownerReference`, e.g., because it was created manually, if it matches a ReplicaSet's selector, it will be immediately acquired.

Deployment

A Deployment provides declarative updates for Pods and ReplicaSets [3]. Deployments are a high level object designed to ease the life cycle management of replicated pods. A Deployment describe the desired state, and the Deployment Controller changes the actual state to the desired state at a controlled rate. Deployments can be modified to change their target configuration and Kubernetes will adjust the replica sets, manage transitions between different application versions.

Service

A Service (Figure 2.5) is an abstract way to expose an application running on a set of Pods as a network service [3]. A Kubernetes Service can act also as a basic internal load balancer for pods. It groups together logical collections of pods that perform the same function to present them as a single entity, and route the incoming traffic between them. The set of pods that constitute a Service are defined by a label selector.

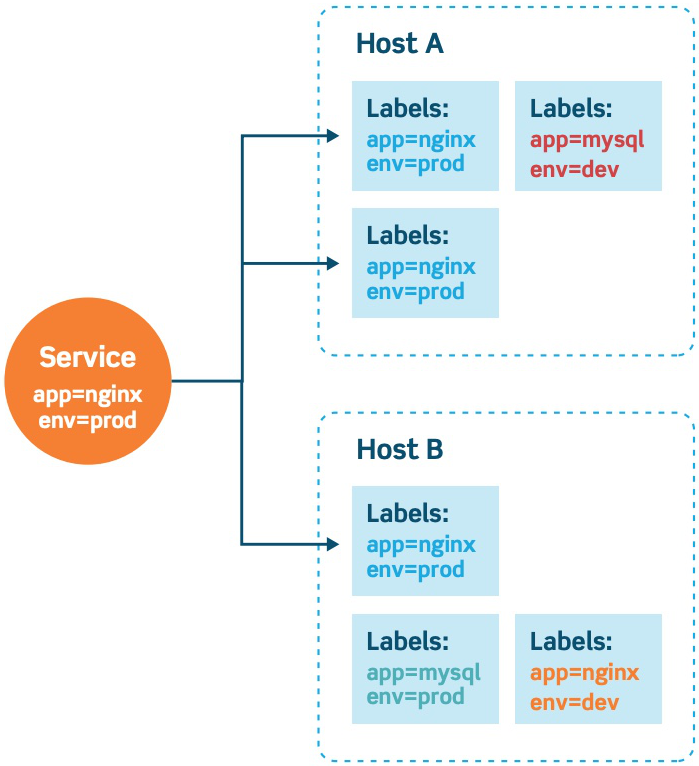


Figure 2.5: Service labels diagram [59].

It provide a stable endpoint for ephemeral object. The Service abstraction allows you to scale out or replace the backend units as necessary. Meanwhile, the Service's IP address remains stable regardless of changes to the pods it routes to [21]. By deploying a Service, you easily gain discoverability and can simplify your container designs.

The resources in a service receive a ClusterIP that exposes the service on an internal IP only [59]. This makes the service reachable only from within the cluster. A NodePort exposes the service on each node's IP at the specified port. This functionality allow to setup an external load balancer or to exposes services that should be reachable from the

outside world on their specific port when required. More often NodePorts are used for debugging purposes, to expose a service in a fast way, avoiding to configure Routes or Ingresses in the first development phases.

2.2.5 ConfigMap

A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume. ConfigMaps allow to decouple environment-specific configuration from the container images, so that the applications are easily portable.

2.3 OpenShift

OpenShift is a suite of containerization and orchestration software developed by Red Hat. Therein, OpenShift includes the OpenShift Container Platform (OCP), which enables setting an on-premises PaaS by extending Kubernetes as shown in Figure 2.6.

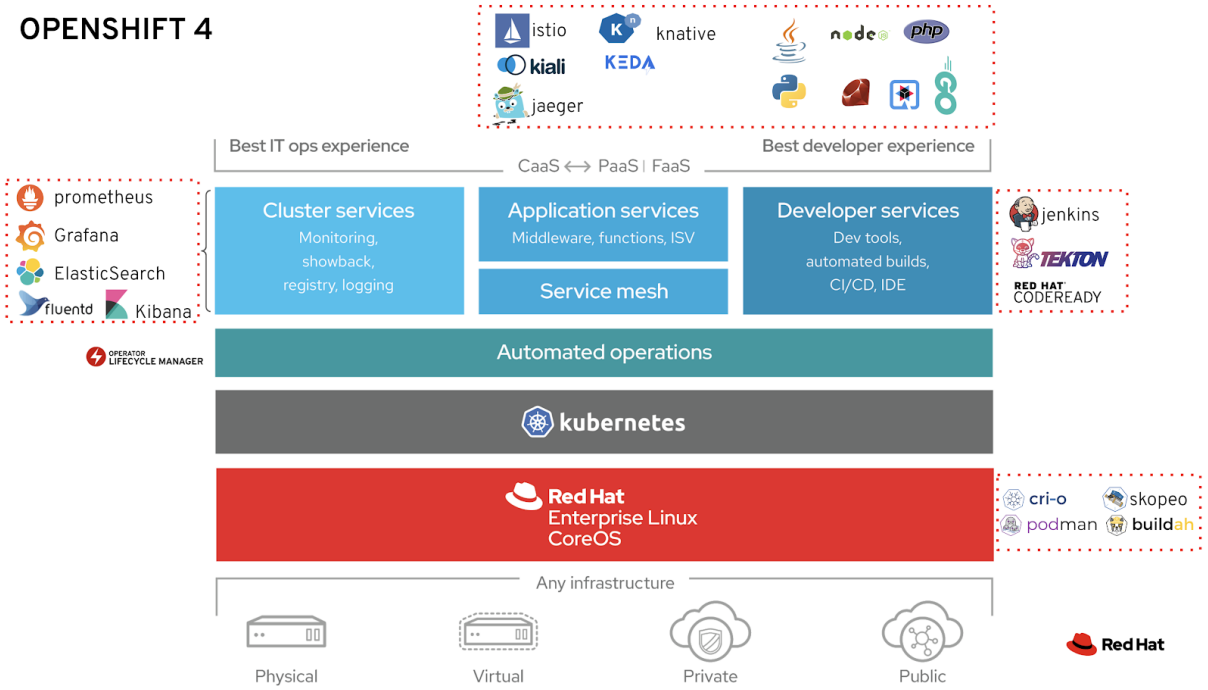


Figure 2.6: Red Hat OpenShift Container Platform v4 architecture. [16].

Both OpenShift and Kubernetes can run on multiple different infrastructures, e.g., bare metal, dedicated or virtual servers, private or public clouds.

While this is enabled in Kubernetes by exploiting the Docker engine, OpenShift exploits the Red Hat Enterprise Linux CoreOS (RHEL CoreOS). RHEL CoreOS is a lightweight Red Hat Enterprise Linux OS with the addition of Linux Container capabilities. RHEL CoreOS shares the same kernel as the Red Hat Enterprise Linux distribution, but is optimized for running containers and managing Kubernetes clusters at scale [16]. RHEL CoreOS includes cri-o, Skopeo, Podman, and Buildah. cri-o is the lightweight container runtime used in minikube, the Kubernetes distribution for the local development [24], as it only takes care of handling containers management. Skopeo is a tool for moving container images between different types of container registries, including in particular the DockerHub and the OpenShift's Quay registry [64]. Podman is a daemonless container engine for developing, managing, and running OCI (Open Container Initiative) Containers on Linux Systems, which provides the same Docker commands and exploits the same Linux functionalities, while also enabling to run containers in rootless mode [55]. Finally, Buildah is a tool that facilitates building Open Container Initiative (OCI) container images.

Over RHEL CoreOS layer is installed Kubernetes (including the orchestration capabilities, but not the Kubernetes cli). Even if this layer is in common with a standard Kubernetes installation, OpenShift adds its own resources. For instance, Templates are used in place of Kubernetes Helm charts (a new Kubernetes tool to use placeholders in Deployments); Routes in place of Ingresses. BuildConfigs, Source2Image, and ImageStreams are instead used to ease the build process, and DeploymentConfig is used to enrich Kubernetes deployments. Since OpenShift is an extension of Kubernetes, the Kubernetes object are still available and guarantees a full compatibility with Kubernetes resources, while OpenShift objects usage is encouraged for newer resources.

Most of the user-tangible OpenShift values stands on top of Kubernetes. OpenShift is a custom Kubernetes installation that extends Kubernetes with custom resources, with a special focus on Kubernetes Operators. In addition to releasing the operator-sdk to facilitate developers in the creation of operators, Red Hat has developed operators for the most popular services (e.g., for MongoDB databases) and most of the OpenShift components are managed and updated by Kubernetes Operators.

OpenShift also makes Kubernetes Operator installable in one click from the OpenShift control panel. Also, OpenShift integrates and keeps the OLM (Operator Lifecycle Manager). In Kubernetes this complex operation must be done manually. In addition to the public operator catalog (Operator Hub), OpenShift provides a private catalog with premium OpenShift certified operators, accessible only by OpenShift customers.

On top of its platform, OpenShift provides a handy UI to manage the cluster and a set of tool for supporting development. These tools include:

- CloudReady Containers (CRC, formerly minishift) which is the OpenShift equivalent of the local development cluster tool minikube;
- the `oc` command-line interface, which extends kubectl by adding namespace management and other handy features;
- the `odo` command-line interface, which helps developers in building images and running applications in multi-host clusters;
- IDE plugins to manage the cluster and to inspect and debug an application running on the cluster from its development environment.

2.4 Kubernetes Operators

Kubernetes describes operators as "software extensions to Kubernetes that make use of custom resources to manage applications and their components" [1]. An operator is essentially a custom controller. A controller is a core concept in Kubernetes and is implemented as a software loop that runs continuously on the Kubernetes master nodes comparing the desired state and the current state of an object, and reconciling such states whenever needed. Indeed, when the current state of an object is different from its desired state, the managing Kubernetes Operator issues instructions for the object so that it eventually reaches its desired state.

The operator is a piece of software running in a Pod on the cluster, interacting with the Kubernetes API server. It introduces new object types through Custom Resource Definitions, an extension mechanism in Kubernetes. An operator monitors such custom

resource types and is notified about their presence or modification. When an operator receives this notification it will start running a loop to ensure that all the required connections for the application service represented by these objects are actually available and configured in the way the user expressed in the object's specification. For example, the Grafana operator adds Grafana and GrafanaDashboard objects to Kubernetes. An user wishing to deploy Grafana just needs to create a Grafana object and to pass its desired configuration as parameters in a custom resource `spec`. The operator will deploy all the necessary objects and keep them running.

2.4.1 Operator Structure and Operator SDK

An operator is made up of two components i.e., API and controller. The API defines a custom resource. The main API file is called `types` and it includes the desired state for a component when deployed. The controller instead contains all the logic for deploying a component and for ensuring that it eventually reaches and maintain its desired state.

The operator-sdk developed by OpenShift allows to generate scaffolding code for an operator. Moreover, its command line includes tools to test, run, build, and deploy a developed operator, as well as to generate the Custom Resource Definition.

The generated Custom Resource Definition describes the new object Kind (i.e., the a new type of resources recognised by Kubernetes) managed by the operator. It is a yaml file which deployment results in adding a new resource Kind to the Kubernetes installation. As for standard Kubernetes resources, the custom one can be deployed with a yaml file. Instead of Pod or Service we will find the custom resource name in the field `Kind` of the yaml, and the `apis` field is not `core` or `apps` (as for standard Kubernetes resources) but our custom APIs.

The operator controller is usually written in Go, the programming language in which Kubernetes is written. During the development the operator is usually ran locally and it performs operations on the cluster from outside. The build process generates an image containing the compiled code. The operator is deployed through a deployment file, and with some role and role bindings we give it the permissions it needs to execute operations.

2.4.2 Operator Maturity Model

Operators have five capability levels (Figure 2.7) [20]:

1. Basic Install;
2. Seamless Upgrades;
3. Full Lifecycle;
4. Deep Insights;
5. Auto Pilot.

Levels are used to classify operators: The higher the level reached by an operator, the more complete it is. Most of the operators reach only the second or third level, that covers the most important and popular features.

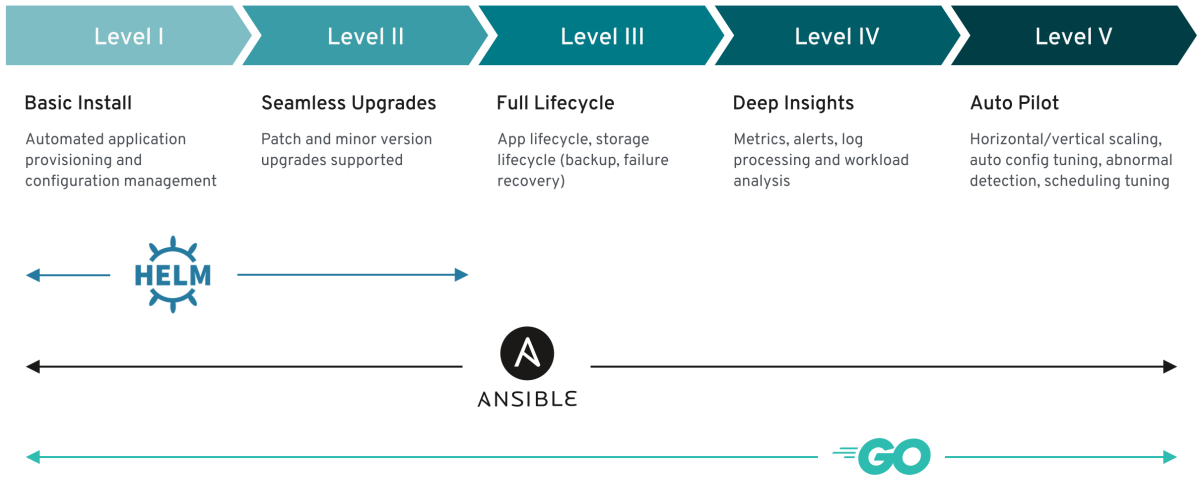


Figure 2.7: Operator capability levels and language supported.

The first level, i.e., basic install, concerns the automated application provisioning and configuration management. In this first stage the operator deploy all the resources (i.e., Pod, Service, Secret, etc.) needed to implement the custom resource.

The second level, i.e., seamless upgrade, implements patch and minor version upgrades. Major upgrades are optional and may not be seamless.

The third level concerns lifecycle management. The operator here takes care of keeping components up and running and it implements mechanisms to recover such components from potential failures. Rollbacks are also implemented in this level. In stateful applications, the operator also manages the storage lifecycle and its backup and restore.

The fourth level includes metrics, alerts, log processing, and aggregation, workload analysis and resource consumption. Since the custom resource can be made of many Pods and containers, all their logs are aggregated at custom resource level. Appropriate log levels and filters should be applied to each component in order to have a useful resource logging. When this level is reached, through the custom resource we can access to an aggregated resource consumes overview, since the custom resource aggregates many objects in a single one. Alerts are fired when a custom resource component is damaged or has an unusual behavior, by analyzing its logs or resources usage. Finally, an automated workload analysis helps in load balance the microservices architecture.

The fifth level is reached when the operator performs all the tasks necessary to automate a custom resource to the point that no manual intervention is required. Horizontal/vertical scaling, auto configuration tuning, abnormal detection and schedule tuning are only few examples. Each application has its own specific management automation needs and this level is reached when all such needs are automatically handled by the operator.

2.4.3 Operator Programming Languages

As shown in Figure 2.7, operators can be programmed in Go, Ansible or Helm. Go operators are the suggested ones because they are developed with the same programming language as Kubernetes, hence easing their integration with the Kubernetes APIs. Ansible and Helm are instead useful to integrate already existing management automation scripts inside an operator.

Go provides a programmatic approach, while Ansible and especially Helm are more declarative-oriented. Moreover, Go operators run directly inside a Pod, while Ansible and Helm operators are made of a Go scaffolding code in the main container and an Ansible or Helm runner in a sidecar for the Ansible and Helm interpretation. For this reason, this two latter solutions add an overhead and are less performant than a totally compiled

operator.

2.5 Management Protocols

How to flexibly manage complex applications across heterogeneous cloud platforms is one of the main concerns in today's enterprise IT [40] [15]. The OASIS standard TOSCA [52] is one emerging solutions trying to address this problem from different perspectives. Building on top of TOSCA, management protocols [12] enable modelling, analysing, and automating the management of complex multi-component applications.

2.5.1 TOSCA

TOSCA (Topology and Orchestration Specification for Cloud Applications [52]) is an OASIS standard whose main goals are to enable the specification of portable cloud applications and the automation of their deployment and management. TOSCA follows a model-driven approach. It indeed provides a YAML-based modelling language for specifying portable cloud applications, and for automating their deployment and management. TOSCA permits describing the structure of a cloud application as a typed, directed topology graph [5], whose nodes represent application components, and whose arcs represent dependencies among such components. Each node of a topology can also be associated with the corresponding components requirements, the operations to manage it, the capabilities it features, and the policies applied to it. Inter-node dependencies associate the requirements of a node with the capabilities featured by other nodes. An application topology can then be declaratively processed [22] to automate the deployment and management of the specified applications [51].

TOSCA permits specifying a cloud application as a service template, that is in turn composed by a topology template, and by the types needed to build such a topology template. The topology template is a typed directed graph that describes the topological structure of a multi-component application [13]. Its nodes (called node templates) model the application components, while its edges (called relationship templates) model the relations occurring among such components.

Node templates and relationship templates are typed by means of node types and

relationship types, respectively. A node type defines the observable properties of a component, its possible requirements, the capabilities it may offer to satisfy other components' requirements, and the interfaces through which it offers its management operations. Requirements and capabilities are also typed, to permit specifying the properties characterising them. A relationship type instead describes the observable properties of a relationship occurring between two application components [14].

TOSCA applications are then packaged and distributed in CSARs (Cloud Service ARchives). A CSAR is a zip archive containing an application specification along with the concrete artifacts realising the deployment and management operations of its components.

2.5.2 Management Protocols

Being composed by multiple heterogeneous components, the management of applications must be suitably coordinated by taking into account inter-component dependencies and potential failures [61]. In other words, the deployment and configuration of the components forming a multi-component application must be suitably coordinated [32], by considering that a component may rely on other components to properly work [37], as well as that a component may fail and that this may cause cascading failures in other components [12]. As the number of components grows, or the need to reconfigure them becomes more frequent, application management becomes more and more time-consuming and error-prone [7].

The management behaviour of topology nodes can be specified by means of management protocols [9] [12], that are integrated in the OASIS standard TOSCA (Topology and Orchestration Specification for Cloud Applications [52]. Each node can be equipped with its own management protocol, which is a finite state machine whose states and transitions are enriched with conditions on the requirements and capabilities of such node. Conditions on states permit defining which requirements of a node must be satisfied in a state, as well as which capabilities the node actually provides in such state. Conditions on transitions instead define which additional requirements must be satisfied to actually execute a management operation in a state. The management behaviour of a composite application can be derived by composing the management protocols of its nodes according to the dependencies defined in its topology. Topology graphs connect the requirements of

a node with the capabilities that can satisfy them. A requirement is satisfied only when the corresponding capability is actually provided.

Faults must always be considered when managing complex composite applications [18]. Management protocols allow to model how nodes react to faults [51]. Through management protocols we can model the management behaviour of application components and they can be composed to analyse and automate the overall management of a multi-component application in a fault-resilient manner [12]. The aim is to define the consistency of component states and to constrain the executability of component operations to the satisfaction of their requirements.

The management behaviour of a TOSCA application is derived by composing the management protocols of its components, according to the application's topology [11]. The global state of an application is indeed defined as the set containing the current state of each of the application components. A global state is considered to be "valid" only if all requirements assumed to hold by a node are connected to capabilities that are actually provided by another node in such global state. An application can only transit from one valid global state to another, by executing a management operation on a component, provided that all requirements needed to execute such operation are satisfied in the starting global state [9].

The derived management behaviour can then be exploited to automate various useful analyses. For instance, given a plan orchestrating the management operations of a TOSCA application to achieve some management goal, one can readily check whether such plan is valid by verifying that it can only traverse valid global states. Such behaviour can be exploited to determine the effects of a plan too, e.g., which application configuration is reached by executing it, or whether it may generate faults while being executed. It also permits to automatically determine plans able to accomplish specific management goals, e.g., reaching a desired application configuration (i.e., the correct deployment) or restoring it after a fault occurred.

Fault-aware Management Protocols Definition

To describe the management behaviour of component described by a node we need to specify whether and how each management operation of the node depends on other man-

agement operations. This operations can belong to the same node or to different nodes that provide capabilities used to satisfy the requirements of the node in analysis. In fault-aware management protocols [10], the first kind of dependencies is described by relating the management operations of the node with its states. This determines the order of execution of the node's operations. The second kind of dependencies is specified by associating (possibly empty) sets of requirements with transitions and states. The requirements associated with a transition must be satisfied to perform the transition, while those associated with a state of the node must continue to be satisfied in order for the node to continue to work properly. As requirements are satisfied when the corresponding capability is provided, the requirements associated with transitions and states actually indicate which capabilities must be offered by other nodes to perform a transition or to continue to reside in a state.

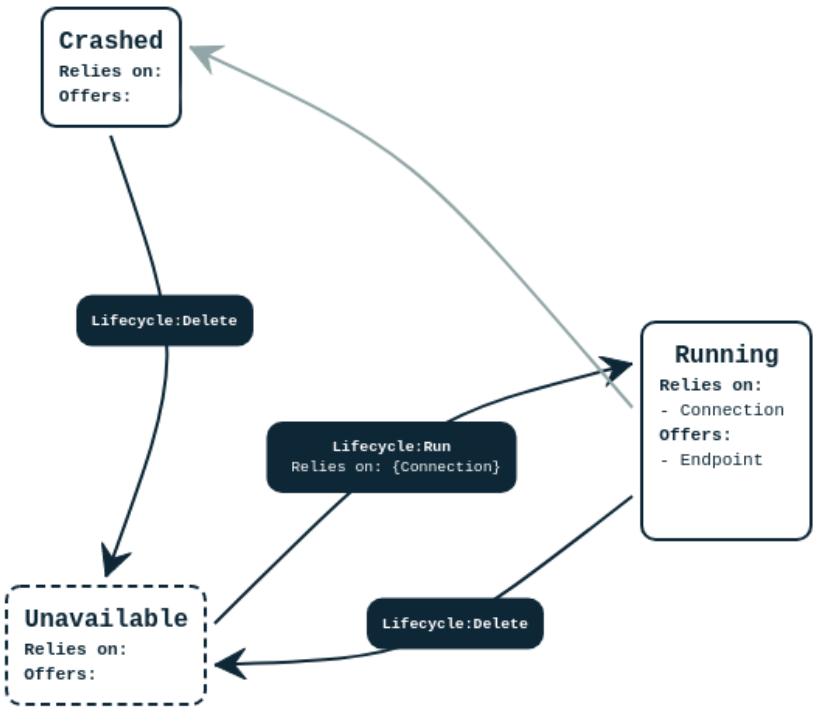


Figure 2.8: State diagram of a component modeled with the management protocols and represented in Barrel.

Fault-aware management protocols also allow to indicate how a node reacts when a fault occurs, i.e., when the node is in a state assuming some requirements to be satisfied and some other node stops providing the capabilities satisfying such requirements. This

is described by a transition relation that models the explicit fault handling of the node by specifying how the node changes its state when some of the requirements it assumes in the current state stop being satisfied.

Figure 2.8 shows the state diagram of a component modeled with the management protocols. We can see the states of the component (the rounded squares) and the transactions between them (the blue arrows). Both states and transactions have requirements (*Connection* in this example). The requirement fault is modeled with a fault transaction (the gray arrow).

Modelling Unexpected Behaviors

Till now we assumed that application components behave according to their specified management protocols. However, the actual behaviour of an application component may be different from that modelled in its fault-aware management protocol, e.g., because of non-deterministic bugs [31].

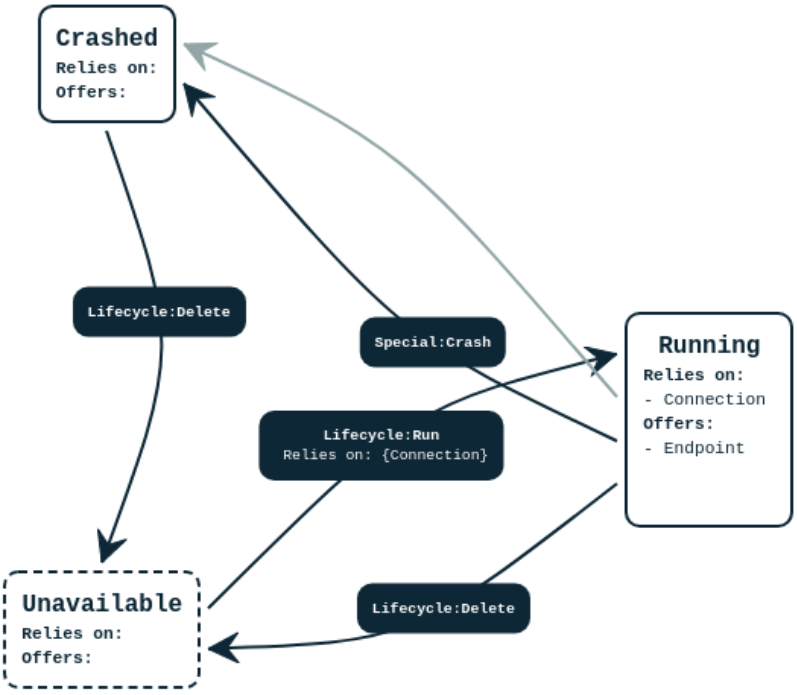


Figure 2.9: Unexpected behavior of a component modeled with the state diagram of the management protocols and represented in Barrel.

The unexpected behaviour of application components can be modelled by adding a "crash" operation to each node, and to automatically complete its fault-aware management protocol with "crash" transitions leading such node to its sink state. From this special state we can model transactions to move back the node (that is modelling a component) to a "safe state", i.e., a state different from the crash state. From this state we already have operations to return in the operational state in which the component were before the unexpected behavior occurs. Of course, to move forward to the operational state the node may need to wait for requirements to be satisfied. This can require to perform operations also on the nodes that provide the needed capabilities.

Figure 2.9 shows the same state diagram of Figure 2.8, with the addition of the unexpected behavior. This latter is modeled with a special transaction between the desired state *Running* and the fault state *Crashed*.

2.5.3 Winery

Winery is an Eclipse project part of the OpenTOSCA [6] ecosystem. It is a web-based environment to graphically model TOSCA topologies and plans managing these topologies. The environment includes a type and template management component to offer creation and modification of all elements defined in the TOSCA specification [25]. Eclipse Winery provides a graphical web-editor with which you can create and maintain all TOSCA entities. It validates and stores all TOSCA entities in the syntax defined in the standard. It is available through the Docker image called `opentosca/winery`. Though Winery we will model a TOSCA definition for the Root components we manage with our operator. This definition will be then exported as CSAR to be processed in Barrel [7].

2.5.4 Barrel

Barrel is a web-based application that permits editing and analysing fault-aware management protocols in composite applications specified in TOSCA. It implements the modelling and analysis approaches presented in [12]. We will use it to verify the fault tolerance of our multi-component Root architecture.

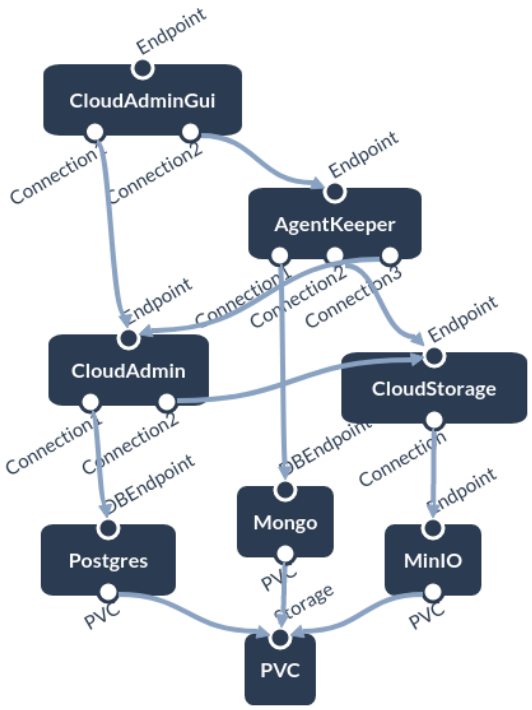
Starting from a TOSCA topology definition generated through Winery [38], Barrel

allows to associate capabilities to states and to define transitions between states. On each transition and state we can specify a the requirements that the application must handle to remain in and move to a certain state. If a transition requirement is not satisfied, the application cannot move to the destination state. We can define fault handlers to specify transitions to a different state if a state requirement is not satisfied anymore.

When the transition schema is completed we can simulate the application response to certain events. Moreover, we can check if it is possible to reach a destination state set given an initial state set, and which steps lead to this new configuration. Finally, by removing requirements and simulating component crashes we can verify the strongness of the designed architecture and its fault tolerance.

Visualizing the Application Topology

The first step is to import a CSAR (Cloud Service ARchive [50]) package containing the TOSCA application to be edited and analysed. Once the CSAR is loaded from the main panel we can visualize the rendered architecture.



Node	Node type	Capabilities	Requirements	Operations
PVC	PVC	Storage		
Postgres	Database	DBEndpoint	PVC	
Mongo	Database	DBEndpoint	PVC	
MinIO	MinIO	Endpoint	PVC	
CloudStorage	CloudStorage	Endpoint	Connection	
CloudAdmin	CloudAdmin	Endpoint	Connection1 Connection2	
AgentKeeper	AgentKeeper	Endpoint	Connection1 Connection2 Connection3	
CloudAdminGui	CloudAdmin	Endpoint	Connection1 Connection2	

Figure 2.10: On the left the IotSnap Root components diagram represented in Barrel. On the right a table showing the components defined in TOSCA.

Components are represented in terms of Nodes and each Node has a Node type. According to the type each Node offers some capabilities and need some requirements. Figure 2.10 shows an example of application topology visualized in Barrel, together with the Barrel's table associating each node with its Node type, capabilities, and requirements. In the application topology, rounded squares represent Nodes, blue bullets model the capabilities they offers, and white bullets model their requirements. Arrows model inter-component dependencies by connecting a component requirement with the capability of another component that satisfies that requirement. This application topology gives us a first hint about the dependencies schema and which the possible fault could be.

Editing the Application

A second panel, called Edit, permits editing the fault-aware management protocols of the nodes in the application topology. The Management protocol editor permits selecting the node type to be edited. All the nodes using this node type will implement the protocols we describse in this Section. Once a node type is selected, its fault-aware management protocol is displayed, and it can be edited with the toolbar right below it.

As we can see in Figure 2.11, from the Edit panel we can visualize all the states of a component. For example, in this case we are visualizing a component called AgentKeeper with three states i.e., Unavailable, Running, and Crashed. The toolbar below the graph allows to set the initial state, add or remove transitions and fault handlers, manage the default handling behavior and edit the states to set offered capabilities and needed requirements.

The initial state is set to Unavailable, and in the graph is represented as a dashed rounded square. On each state we can see the list of requirements (relies on) and capabilities (offers). In this example, the only state that is offering capabilities and has requirements is the Running state. This lists can be managed from the Edit button under States: For each state it shows the list of capabilities and requirement of the node, and by checking the checkboxes we can choose which of them we want in the state we are editing.

Transitions are represented by the blue arrows that connects states. Transitions can have requirements too. If we put a requirement on a transition we cannot follow that

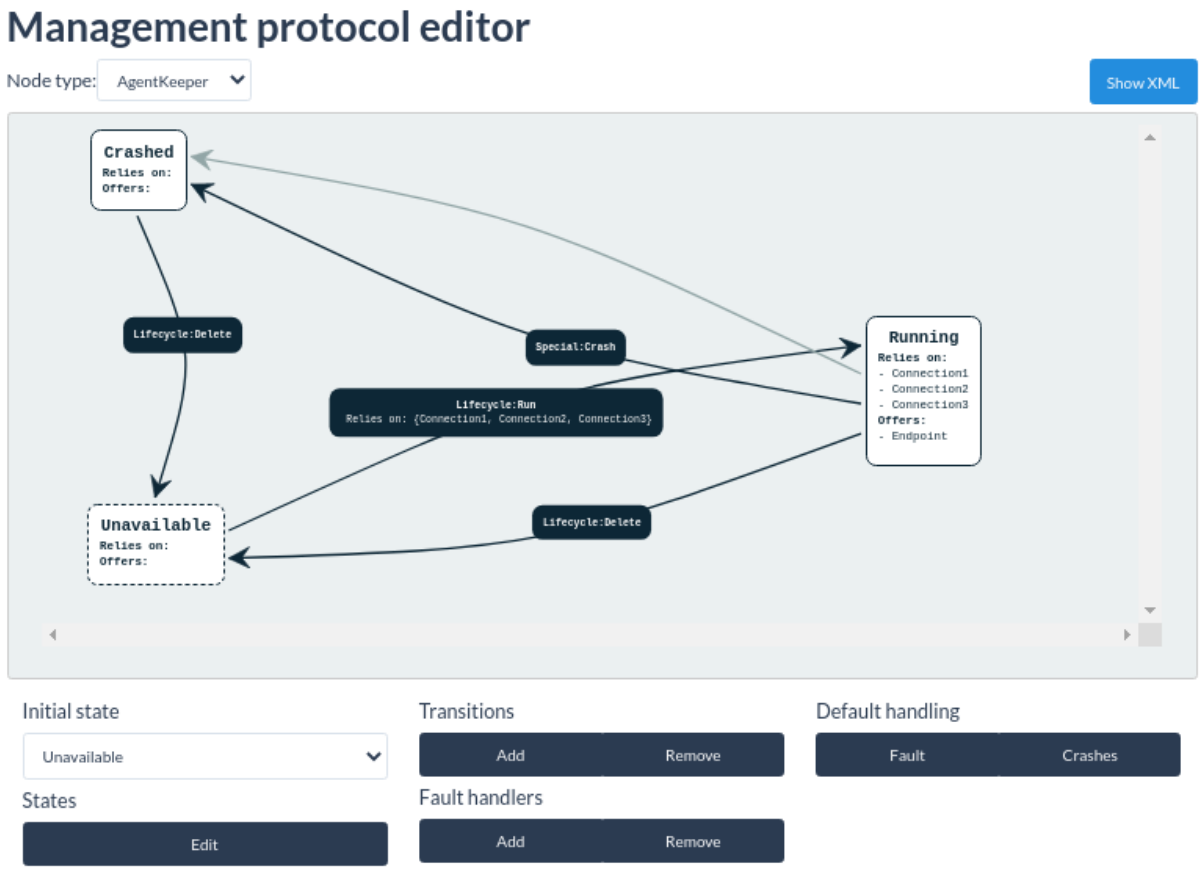


Figure 2.11: The Edit pane of the Barrel tool.

path until the conditions are met. The transitions requirements may be different from the target state requirements: For example, we may need a requirement only to change state, even if such requirement is not needed anymore in the state reached after performing the transition. Instead, while it is also possible to have a transitions with less requirements than its target state, it is pretty difficult to find a real use case: The node will reach the new state, then if the additional requirements are not satisfied it goes in a broken state and a fault handler has to be invoked. For that reason, a well designed topology does not have less requirements in transitions that in corresponding target states [10].

The Crashes default handler add a special transition Crashes from each component to a special Crashed state. It is useful to simulate the component crash without having to manually add a state to the topology. Barrel will crate it for us. Unfortunately the Crashes transitions are a bit complicated to be managed because the Crashes button add the transition from each state and there is no way to add a single transition. Transitions

in excess must be manually deleted with the Remove button under Transitions.

The Fault default handler behavior is similar to the Crashes, but it handles a fault due to a requirement lost. The Fault button add fault handler from all the state that have requirement, but we can also manage single handler from the Fault handlers section. While the Crashes is modeled as a transition of the group special instead of the lifecycle group, a fault handler is not even a transition. Indeed, we can see fault handlers in gray in the diagram, instead of in blue. This is because the fault handler cannot be invoked normally, but instead is the only operation we can do when there is a fault. It resolve a missing requirement. While is not possible to have a fault handler starting from a state with no requirements, we can have many fault handler on states with multiple requirements. For example, if we have two requirement, we can have up to three different fault handler: One that leads to a state where the first requirement is satisfied and the second not, another handler that leads to a state that requires only the second requirement, and finally a handler through a state with no requirements. Barrel will follow always the fault handler that leads to a state with with as many requirements as possible. In any case, it is always important to include a fault handler path to the state with no requirements at all, otherwise we may have a stall.

The edited application topology with transitions and fault handlers can be exported as an updated CSAR or analysed.

Analyzing Applications

The Analyze pane permits interactively analyzing the fault-aware management behaviour of the composite application in the imported CSAR.

A Simulator section (Figure 2.12) contains a table that permits interactively simulating the behaviour of the composite application. More precisely, the Simulator allows to simulate sequences of operation/fault-handling transitions, hence permitting to determine their effects on the whole application. Given a situation, through this section we can see which operation the operator could perform. It can be used also to verify whether a sequence of steps is possible or not. For example, by observing the operator's steps we can try to reproduce the actions on the Barrel simulator to see if the sequence is correct and has the expected behavior, if other operations should be made before a certain one,

or even if the sequence in analysis cannot be performed at all.

	State	Offered capabilities	Assumed requirements	Available operations
PVC	Available	Storage		Lifecycle:Delete
Postgres	Running	DBEndpoint	PVC	Lifecycle:Delete Special:Crash
Mongo	Running	DBEndpoint	PVC	Lifecycle:Delete Special:Crash
MinIO	Running	Endpoint	PVC	Lifecycle:Delete Special:Crash
CloudStorage	Running	Endpoint	Connection	Lifecycle:Delete Special:Crash
CloudAdmin	Running	Endpoint	Connection1 Connection2	Lifecycle:Delete Special:Crash
AgentKeeper	Unavailable			Lifecycle:Run
CloudAdminGui	Unavailable			Lifecycle:Run

Figure 2.12: The Barrel simulator. It shows the current state of each node in the simulation, their offered capabilities and assumed requirements. On the right we have the available operations. The orange one cannot be performed for missing requirements.

The Analyze tab contains also a Planner (Figure 2.13. It permits specifying two different configurations of the imported composite application (source global state and target global state). The first one specifies for each node its starting state. All the nodes are set to their initial state by default. In the second configuration we describe the desired state of each node of our application. The planner calculate if the target state is reachable from the initial state. If possible it displays the sequence of operations and/or fault-handling transitions that leads from the source global state to the target global state.

We will use the planner to validate our operator, by comparing the operator's steps with the steps provider from the planner. Moreover, we will use it also to design the Backup features. We will model all the involved components and their behavior. The planner will tell us if there is a path to all the target states and which steps we need to implement in the operator to reach our aim in the implementation.

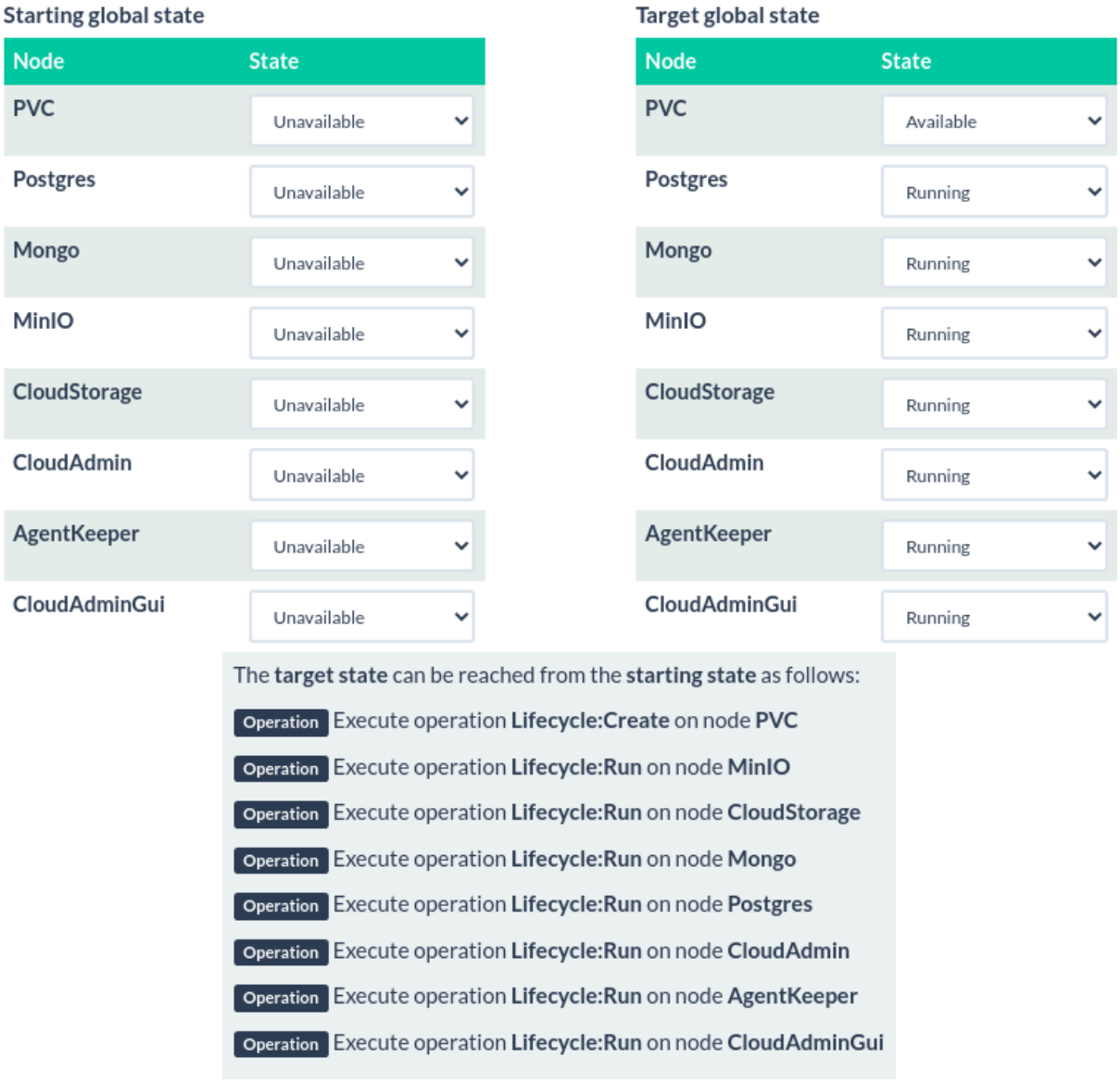


Figure 2.13: The planner section of the analyze pane in the Barrel tool.

Chapter 3

Design of the IotSnap Root Operator

In this chapter we first present the context and requirements of the Kubernetes Operator presented in the thesis, which are given by Extra Red's IotSnap project (Section 3.1). Then, we describe the operator's design choices and their rationale (Section 3.2). Finally, we show how management protocols and Barrel were exploited to design the backup capabilities of the Kubernetes Operator (Section 3.3).

3.1 Context and Requirements

IotSnap is a cloud architecture designed to receive and operate on data coming from sensors of industrial production machinery. The operator developed during the internship has the task of deploying and maintaining a core subset of IotSnap's microservices which is called IotSnap Root.

3.1.1 The IotSnap Project

IotSnap was born as a high-level business project based on an order from a customer that designs and supplies connected industrial machinery, leveraging industry 4.0 for its business. Their project with Extra Red has the ultimate goal of using AI for predictive maintenance [8]. By analyzing the data from the sensors of their machines, they wish to automatically determine the time left before a fault, so as to replace the piece before the machinery breaks, with the ultimate goal of reducing production discontinuity to bare-minimum.

The goal of IotSnap is to provide customers with a global solution for the implementation of new maintenance procedures and sharing of performance and production data. The project includes the creation of a software solution dedicated to the collection, storage, analysis, and visualization of data from sensors deployed on industrial machines, and based on a cloud infrastructure. In the IotSnap project, Extra Red division is responsible for developing and operating the software solution for the processing of data from the sensors.

Each IotSnap component has been designed in layers, so that it can be extended and tailored to customer. In addition, it is composed of two main stacks that are always present, TICK (Telegraf, InfluxDB, Chronograf, and Kapacitor) and Root, and a series of plug and play components and secondary stacks, which are deployed according to the customers' needs.

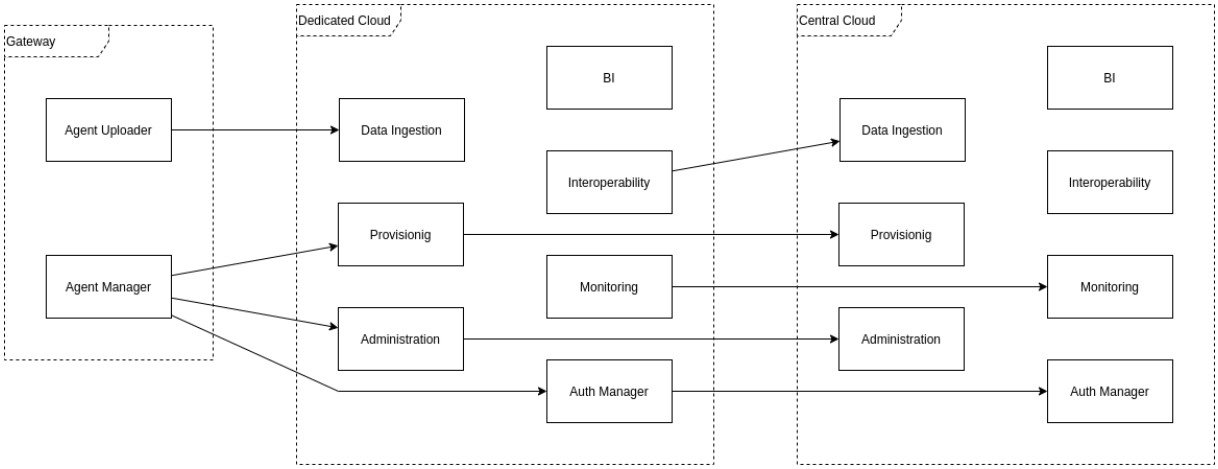


Figure 3.1: IotSnap high level architecture diagram.

As we can see in Figure 3.1, the solution developed by Extra Red is organized according to a hierarchical model. The information coming from the machinery are collected by a Gateway, i.e., an edge device installed inside the industry. The Gateway pushes the data to a hub dedicated to the customer, called *Dedicated Cloud*. Then, the information that the customer considers non-sensitive and the information necessary for remote monitoring of application environments and for provisioning are published on the *Central Cloud*. The Central Cloud is the Extra's cloud, connected to a group of customer's clouds. Often, in addition to the central cloud we also find a Root Cloud of an even higher level than

the central cloud. The purpose of the root cloud is to distribute software patches or apply bulk actions to all central and dedicated clouds. The interaction between central cloud and root cloud is the same as between dedicated and central cloud. The standard installation of IotSnap has three levels, but its design is made in such a way that they can be extended without limits in this waterfall structure. Dedicated and Central clouds have an identical architecture, while the Root Cloud is composed by only a subset of their microservices.

The cloud instances are intended to be allocated on Extra Cloud, an Extra Red service based on the Red Hat OpenShift technology platform, which offers IaaS and PaaS services. Extra Red takes care of resource provisioning, configuration and maintenance of the cloud infrastructure. In addition, since the Extra Red Cloud is based on OpenShift, and therefore on Kubernetes, each component will be encapsulated in a Docker container and the stack will consist of a set of Kubernetes and OpenShift resources.

The IotSnap system is a heavily distributed system consisting of a large number of applications divided into different clouds and hosts. Therefore the testing, build, preparation, and configuration of the cloud and deployment and configuration of the applications are highly automated processes. The IotSnap Root Operator goes in this direction, by automating the deployment and configuration of the Root portion of IotSnap.

The deployment is done through a series of Ansible scripts. A part of these acts on the administrative side of the cluster and take care of preparing the namespace, users, and permissions, as well as the setup of the secrets for pulling the images. A second set of scripts, executed by an automation pipeline, is responsible for assembling the YAMLs that describe the resources to be deployed. The Ansible code reads the resources to be deployed, retrieves the templates, overrides them with the customizations and sets the specific environment variables for the client and its associated cloud and finally aggregates the resources in a set of files to do deploy. A second pipeline reads the files and deploys them in a given order, ensuring that the system is deployed and reaches the desired configuration.

The IotSnap Root Operator has the purpose of automating this last phase. The goal is to totally delegate to it the deployment of the resources relevant to Root, in order to minimize the final script. As far as possible, we have to insert the template directly inside

the operator and let it customize the resources before deploying them, passing to it only the necessary parameters.

Finally, it is worth nothing that the IotSnap Root Operator is intended to be a proof of concept to understand advantages and limitations regarding the use of Kubernetes Operators for the automation of complex deployments based on common templates.

3.1.2 IotSnap Root Architecture

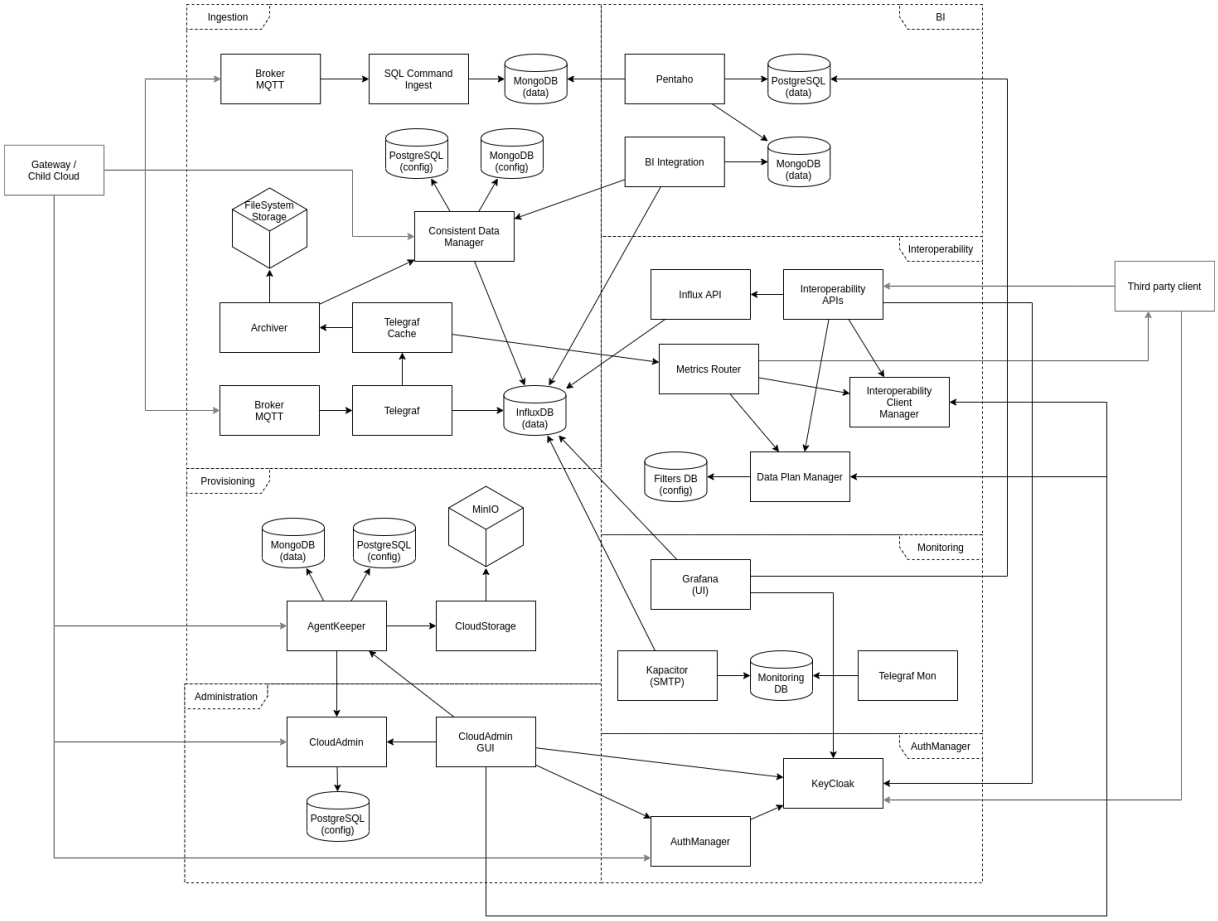


Figure 3.2: Architecture diagram of a cloud instance (dedicated or central), showing the IotSnap architecture components divided in areas.

In Figure 3.2 we can see the IotSnap components and their interaction. They can be divided in five areas:

1. **Ingestion**, the subsystem that allows to transfer the machine data from the gateway to the first level cloud;

2. **Provisioning**, the subsystem that allows to update the software components of IotSnap distributed across the gateway and the cloud;

3. **Interoperability**, the subsystem that transfers data from one cloud to other clouds (to the next level cloud, but also to third-party clouds);

4. **Monitoring**, to monitor both gateways and clouds;

5. **Administration**, to administer an entire cloud tree.

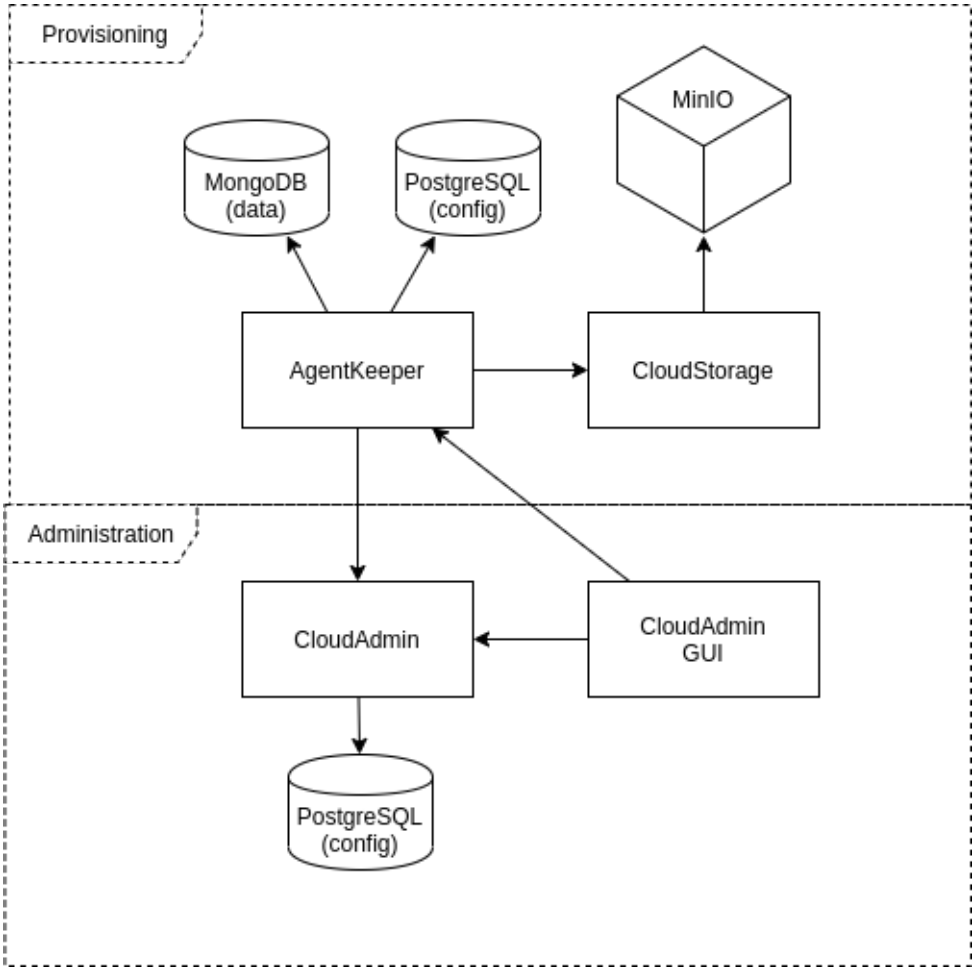


Figure 3.3: IotSnap Root architecture diagram.

The Root portion of IotSnap (Figure 3.3) is composed by the software components in the provisioning and administration subsystems. It is the core of the IotSnap system, and is called Root because contains all the microservices that are presents in all IotSnap

installations, and to which the other components are attached. Root is made by the following components:

- **MinIO**, as object storage;
- **CloudStorage**, that is a MinIO middleware, for storage management;
- **PostgreSQL** as main database used of the provisioning area;
- **CloudAdmin** that expose API to manage cloud and provisioning settings;
- **AgentKeeper**, that manages client configuration and software distribution;
- **MongoDB** as no-SQL database used by AgentKeeper;
- **CloudAdminGUI**, the GUI to interface with CloudAdmin and the corresponding administration microservices in the other areas.

CloudStorage, CloudAdmin, AgentKeeper, and CloudAdminGUI are exposed on internet.

AgentKeeper is the most important component of the provisioning area. It is responsible to distribute software updates to the in-industry gateways. Since security is important in industry, and gateways can potentially contain confidential information or trade secrets, the software is exchanged with the pull mechanism instead of push. The gateway periodically contacts the cloud to find out about the availability of software updates, instead of the cloud being notifying the gateway when a new version is available. In this way, the gateway can also be protected from connections from the outside, as it is always the gateway who starts the communication. To comply with this foresight, the interoperability module also awaits data push on the cloud instead of pulling the edge device.

The AgentKeeper databases stores information about the customer, i.e., its name, the current and previous deployment, the deployment status, and if it has been downgraded; software information, i.e, the software group and version, release notes, extensions, if is a manger software, and if it is live; and provisioning activity information, i.e., info about updates of the software and the manager itself, including start and end date of upgrades and errors.

CloudAdmin deals with the following cloud administration tasks:

- it manages client registrations in the cloud and subscription to the higher level cloud if it exists, through REST services;
- it exposes REST services to the CloudAdmin GUI component in order to get the list of all clients and the detail of a client given a name;
- it exposes REST services to the AgentKeeper instance of the same level cloud, in order to to retrieve the coordinates of the parent cloud;
- it exposes REST services to the Influx Uploader component of the same level cloud (outside the Root scope) so that it can retrieve the coordinates of the parent cloud Influx Uploader to which to forward the data.

CloudStorage is a MinIO middleware that allows AgentKeeper to store and retrieve software update files. CloudAdmin GUI offers a web interface to manage the cloud, by interacting with the CloudAdmin APIs. MongoDB and PostgreSQL are databases used by CloudAdmin and AgentKeeper to store their data.

3.1.3 Gateway-Cloud Interaction

Gateways are machines installed within the industrial plant, which collect data from the sensors of the machinery and send such data to the cloud. They are the edge devices of the industrial IoT infrastructure.

As shown in Figure 3.4, some of the gateway-cloud interactions are between AgentKeeper and CloudAdmin on the cloud side and a gateway component called AgentManager, responsible for administrative communications with the cloud and software and configurations updates and management. Periodically, the AgentManager contacts the AgentKeeper to check if its version is the latest available and, if this is the case, it gets updated to the latest version. It proceeds to download from the AgentKeeper the software and configurations needed by the Gateway components and it proceeds with the deployment of the applications on the Gateway and subsequently to their start-up. It periodically contacts the AgentKeeper to find out about any software updates. It monitors managed applications and exposes start, stop and restart functions for the Gateway.

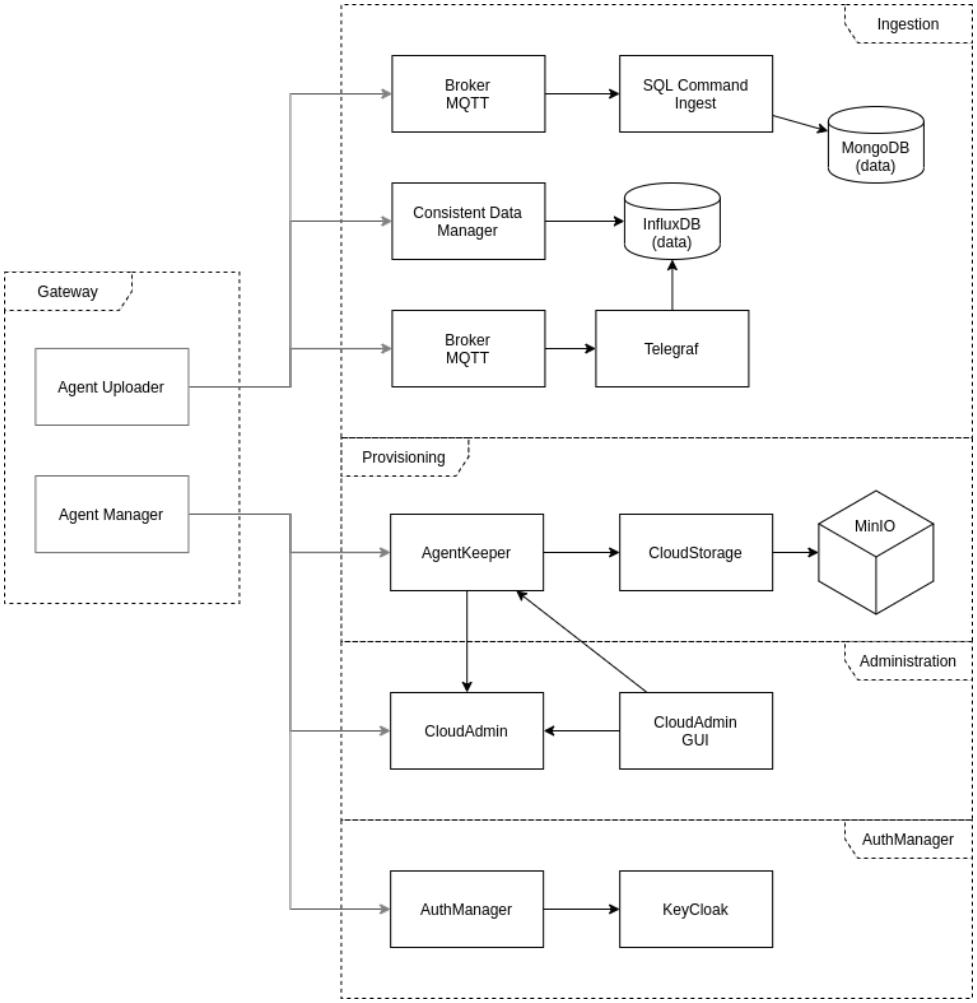


Figure 3.4: Gateway - cloud interaction diagram.

AgentKeeper in turn exposes services to the AgentManager for checking new versions of the AgentManager themselves and of the applications and configurations it manages. It also offers services for downloading such software and configurations.

On the cloud side, AgentKeeper also interact with MongoDB and PostgreSQL to retrieve and store information about the gateway and with CloudAdmin to retrieve parent cloud address. The sequence diagram in Figure 3.5 shows the AgentKeeper interaction with PostgreSQL and MongoDB in response to AgentManager requests. AgentManager first ask for the deployment descriptor version, that is stored on PostgreSQL. If AgentKeeper returns an updated version, AgentManager requests a new deployment document version, that is stored on MongoDB instead. Finally, it download the software updates, stored on MinIO and retrieved through CloudStorage.

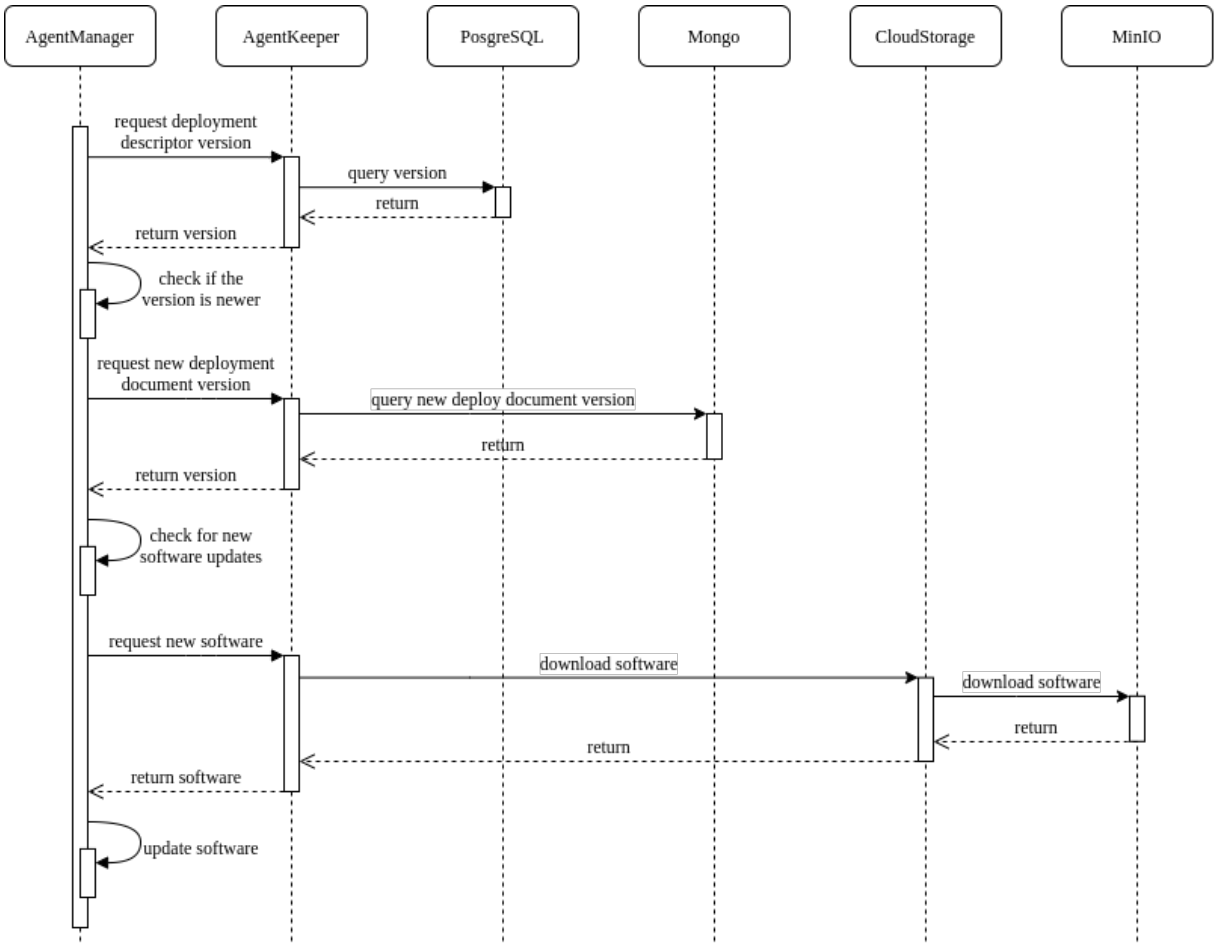


Figure 3.5: Sequence diagram showing the AgentKeeper interaction with PostgreSQL, MongoDB, and CloudStorage (cloud components) in response of AgentManager (gateway software) requests.

CloudAdmin manages the subscriptions. The subscription process consists of two steps. First there is a client request to subscribe to the higher level cloud (Figure 3.6); then the client requests to the higher level cloud for its activation status, e.g., to retrieve data about the AgentKeeper url, activation status, topics name to which send information, and so on (Figure 3.7). This information is stored on the PostgreSQL database.

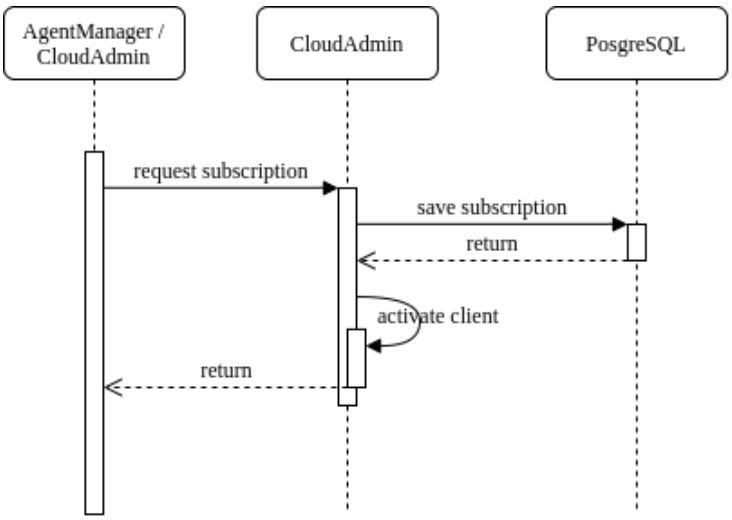


Figure 3.6: Sequence diagram showing the CloudAdmin interaction with PostgreSQL (cloud software) in response to a subscription request from an AgentManager (gateway software) or a child cloud CloudAdmin instance.

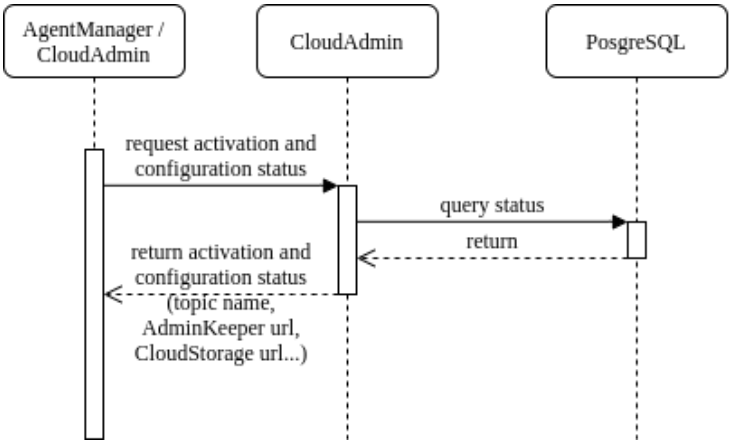


Figure 3.7: Sequence diagram showing the CloudAdmin interaction with PostgreSQL (cloud software) in response to a status and activation request from an AgentManager (gateway software) or a child cloud CloudAdmin instance.

3.1.4 Dedicated Cloud - Parent Cloud Interaction

IotSnap is organized according a hierarchical model, as shown by the example in Figure 3.8. At the top of the tree there is the Extra Red Cloud. It only contains the IotSnap microservices but only the ones for administration and cloud status monitoring. Its aim is to distribute updates through AgentKeeper and to monitor the health state of all the child clouds.

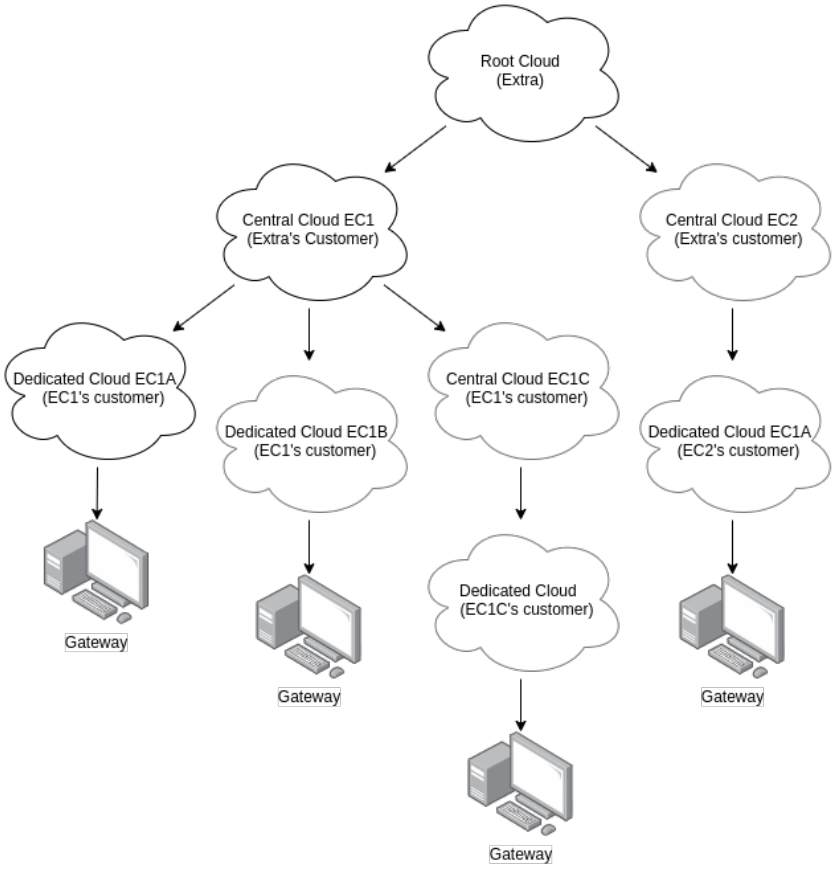


Figure 3.8: IotSnap hierarchical model.

Under the Root cloud we found the Central Clouds. There will be a Central Cloud for each Extra Red's customer, to control, monitor, and retrieve data from all the sub-customers or industrial plant. A central cloud distributes upgrades to child clouds and receives sensors and cloud data from them. Central clouds can have as child both other central clouds and dedicated clouds.

A Dedicated Cloud is the lower cloud in the tree. As we have seen, it distribute updates to the gateways and receives sensors and monitoring data from them. This information

are cleaned, aggregated, and filtered, and then sent to the parent cloud. Updates to be distributed comes from the parent cloud.

Except for the root cloud, that is composed by only a small subset of microservices, each cloud level is equal. Depending on customers' needs, single microservices or set of them may be enabled or disabled, and other ad-hoc services may be deployed, but the cloud architecture and the way components interact each other is the same in all levels. Also, microservices of each level are exactly equal. They use the same protocol with parent and child clouds independently from the cloud nature i.e., gateway, dedicated, central, or root.

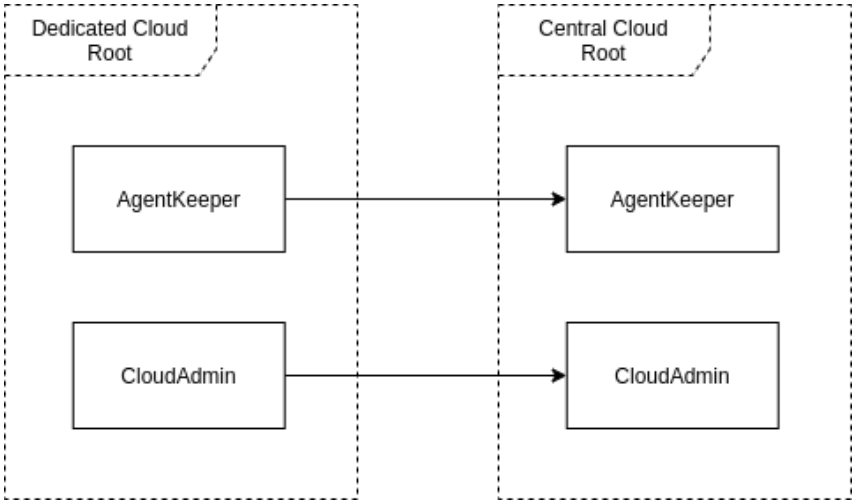


Figure 3.9: Diagram showing the interaction between the root instances in the dedicated and central cloud.

Among the microservices in IotSnap Root, CloudAdmin and AgentKeeper are responsible for communications between clouds (Figure 3.9). The dedicated cloud's CloudAdmin instance talks to the central cloud CloudAdmin, and this is also the way in which lower-level cloud subscribes to a higher-level cloud (also called parent cloud). The request takes place in the same way in which the gateway subscribes to the dedicated cloud associated with it. AgentKeeper communicates with the higher level cloud too. As with CloudAdmin, the principle is the same as that between gateway and dedicated cloud. Just as AgentManager contacts AgentKeeper on the dedicated cloud, the latter in turn contacts the AgentKeeper instance of the upper level for information on software updates and configurations. In this way it is possible to release updates not only for the customer's

custom components, but also for the IotSnap core. Updates flow from the central cloud to all the clouds that are subscribed.

The fact that information flows from one level to another with this mechanism using the same protocols for both cloud-gateway and between clouds communication greatly simplifies the deployment of the infrastructure. From an architectural point of view, the dedicated cloud and the central cloud are identical. Only configuration changes. Configurations are currently managed by Ansible through templates. Configurations are done via ConfigMap or environment variables.

3.1.5 Performance Requirements

As far as the operator is concerned, there are no stringent requirements to be respected. Extra Red has left total carte blanche on both its design and implementation. For Extra Red the purpose of prototype is to fully evaluate the possibilities of an operator for a very specific case and a bit far from the common use case. They are interested in this new type of technology and I want to understand if it is also applicable in a wider automation context. The requirements therefore include covering as many operator levels as possible. Another requirement is to apply design patterns aimed at reusing the code, in order to be able to use it for the development of other operators in this context.

More stringent requirements concern the components to be orchestrated instead, as the operator must perform a deployment and management that guarantee a certain level of reliability and performance. Among the Root components we have CloudAdmin and AgentKeeper, that are two of the most important and central components of the cloud. AgentKeeper takes care of distributing software updates to gateways, therefore it will have heavy I/O needs on the network. In addition, the software to be distributed will be temporarily saved within the component: Therefore in the future it will be appropriate to evaluate the advantages and disadvantages in the use of RAM or disk, depending on the cloud on which you will deploy. The current implementation of the component saves data to disk, so it will need enough space and a fast disk. CloudAdmin deals with the dialogue with the gateway, it does not have strong performance requirements, but it must be a reliable component, therefore it should be scalable to obtain redundancy and fault tolerance.

The aim of the operator was initially to ease the deploy, by aggregating many templates into a single custom resource. Then, its maintenance, starting from keeping the service up and running, to restoring the configurations and the live update of some variables. Backups and monitoring are *nice-to-have* features.

3.2 Operator Design

3.2.1 Grain of the Operator

The first big choice was whether to create an operator for each service or a single operator for the whole root stack. According to Operator Framework's best practices [45] an "Operator should manage a single type of application, essentially following the UNIX principle: Do one thing and do it well", and also "if an application consists of multiple different tiers or components, multiple Operators should be written for each of them". The main task that the IotSnap Root Operator must solve is to simplify the deployment of the Root components of IotSnap, hence suggesting that the development of a single operator for managing all of them could be the way to go.

To better understand whether this was the right way to go, we analyzed the existing operators. Notably, almost all the operators present in the OperatorHub are automating the management of a single application. However, they were all cases very far from ours: The operators were referring to a product, and they assisted in the complex and distributed deployment of this product where there were particular problems. The most used operators are meant for databases, where there are serious problems related to redundancy and distributed replicas. In these cases an operator helps to manage the synchronization between distributed replicas with well designed and tested master/slave mechanisms.

A much similar case to ours is that of the Grafana operator [62], developed by Integreatly, a Red Hat team for "Integration solutions". Among the contributors we have found some that have also contributed to the development of the Operator SDK. The Grafana operator manages not only Grafana deployments, but also dashboard, data sources and plugins. Following this example, the most similar to our case we found, and reassured by the fact that Red Hat was behind its development, we opted for a single operator.

3.2.2 External Operators

Postgres, MongoDB, and MinIO operators are already available on the OperatorHub. Therefore, we had to choose whether to keep these three services within the IotSnap Root Operator or outsource them and use their operators.

Using existing operators offers many advantages, because it simplifies the operator to be developed, delegating the management of almost half of the Root services to them. A dedicated operator, certified and used by hundreds of thousands of users [49], offers many guarantees in terms of performance, security, and reliability. In addition, it saves the work of developing and updating the needed features for that component, in full compliance with the principles of software reuse.

However, one of the main objectives of this operator is to facilitate the deployment of IotSnap, and dividing Root into four different operators does not help in this task. The official best practices of the operator framework expressly state that "an operator shouldn't deploy or manage other operators" [45]. Therefore, it is excluded to delegate the entire stack deployment to our operator if we use the official operators of MongoDB, PostgreSQL, and MinIO.

A turning point in the decision was the fact that the MinIO operator does not support the version of MinIO used by CloudStorage. Therefore, using the official operator would have required substantial changes to CloudStorage to update its compatibility, and Extra Red felt that the benefits offered would not justify the cost of those changes.

Moreover, the IotSnap PostgreSQL instance must be set up using some initdb scripts that are strongly coupled with the Root microservices. An external operator would make deployment more complex and more error prone.

Finally, the use of MongoDB is pretty limited, and can be considered a minimal database supporting a service. No large amounts of data are saved on this database, there are no heavy I/O operations, and there is no need to scale the service.

These considerations led us to the choice of creating a custom Kubernetes Operator for managing services, rather than updating, adapting, and coordinating already existing, ad-hoc operators.

3.2.3 Dependencies and Deployment Order

In order not to weigh down the cluster in the deployment phase, we decided not to start all the services together but to start them staggered. At the moment, the only constraints on the deployment order are that by the time CloudAdmin and AgentKeeper are deployed the PostgreSQL service must have been created, but without the need for Postgres to be ready and listening.

Nevertheless, getting each component deployed only after the components it depends on have been created adds resiliency for the future. If a service is not ready, those who depend on it will restart until it is ready, without causing problems. Anyway, it is important to keep the possibility of checking the ready state of the components open [65], in order to be able in the future to block the deployment process if some component needs more time to start, or whenever we would have a microservice with stronger constraints on the readiness of its dependencies.

For the above, we decided to deploy components following the topological sorting of the graph. The architecture diagram in Figure 3.3 shows the dependencies occurring among the Root microservices. PostgreSQL, MongoDB, and MinIO are the only services that have no dependencies. Then, CloudStorage can be deployed as soon as MinIO is ready. When both CloudStorage and Postgres are available we deploy CloudAdmin. AgentKeeper comes as soon as CloudAdmin and MongoDB have been deployed. Finally, CloudAdminGUI, right after AgentKeeper.

3.2.4 Persistent Storage

Initially, we had chosen to have a single Persistent Volume Claim for all services that requires persistent storage. The choice was due to purely commercial and administrative reasons: It would have allowed to sell a single block of storage rather than many small fractioned blocks, one per service. Having more PVC indeed requires to be able to estimate the space necessary for each microservice, which is not easy in the development and first deployments of the platform. Furthermore, it entails having a sufficient margin of space in each PVC to allow the correct operation of the services without running into lack of space. This leads to allocating much more space than necessary, increasing the risk of

over provisioning if compared to the solution with single PVC.

This decision subsequently ran up against implementation problems. The real-world cluster with OpenShift Container Platform version 4.3 is based on Azure, and therefore uses azure-file as StorageClass for ReadWriteMany. azure-file is in fact one of the few OpenShift storage to support ReadWriteMany. On the cluster there is also a standard StorageClass, based on azure-disk that allows only the ReadWriteOnce. PostgreSQL has known bugs with azure-file, and more in general has many compatibility issues with ReadWriteMany storage. This issues involve permissions [34] and missing support to hard links in Azure Files [48][35], and while there was a workaround [33] in OpenShift Container Platform v3.11, this is not working with newer cluster versions. Moreover, in our tests Azure Files was considerably slower compared with Azure Disk, that instead is faster, easier, and cheaper.

As a consequence, we agilely changed our initial design choice and opted for a dedicated PVC for each service. Despite the administrative and billing complications, the choice is better on the technical side, in line with the best practices, and more efficient.

3.2.5 ConfigMaps

We decided to move the ConfigMaps needed to configure our applications out of the operator scope. Usually, they are deployed by the operator, sometimes using parameters passed through custom resource to fill the placeholders. However, an Ansible automation assembling the ConfigMaps was already available. The scenarios and deployment platforms are varied and do not always include the use of the operator. In this context, leaving the ConfigMap deployment to the operator would require the automation rewrite, which in the case of deployment with operator should insert the parameters in the custom resource instead of replacing them in the template. With the aim of reusing the already existing automation, we assumed that the ConfigMaps will be already present at the time of deployment.

The operator has the purpose of managing them and preserving their state in which they are located at the time of deployment. Therefore, its task will be to ensure that they remain up and that they are not changed at runtime, possibly restoring them.

3.2.6 Routes and Ingresses

While Kubernetes is supporting Ingresses for routing and load balancing tasks, OpenShift supports Routes for the same purpose. Routes are born to ease the service exposure and the load balancing features, that on Kubernetes were delegated to external resources and third party proxies. Only lately, Kubernetes has started developing Ingresses, the Kubernetes respective of Routes. Today Ingresses are still in beta. Even if Ingresses will become more and more popular on Kubernetes, they are still offering less features than Routes: Among these there are wildcard domains and generated pattern-based hostnames [60] that Extra Red is exploiting in it architectures. Since Ingresses are still in beta and are not offering all the features needed by Extra Red, we have chosen to use Routes instead of Ingresses.

3.2.7 Performances

As we saw in the requirements section, there are no requirements regarding the operator itself. The requests from Extra Red are to cover as many aspects as possible and implement a design pattern that allows the reuse of code. The requirements that we want to discuss here are those concerning the performance of the IotSnap Root components

According to the requirement analysis we have to deal with a heavy I/O on AgentKeeper. The current implementation of the application saves data to disk, so it will need enough space and a fast disk. The choice to have a single ReadWriteOnce PVC per component is in line with this goal, as ReadWriteOnce volumes are generally faster than ReadWriteMany because they are not distributed. We choose to keep the PVC lifecycle outside the operator, for that reason the storage type is not under our control. Anyway, the PVC settings allows to choose the StorageClass, that is sufficient for this goal. For example, on the test cluster we assigned a premium storage class to improve performances.

CloudAdmin requirements were about reliability, in particular redundancy and fault tolerance. To satisfy this requirements we need to scale the component. The persistent storage absence simplifies the work. The use of dedicated labels in the deployment in conjunction with an appropriate selector on the service will be sufficient to balance the traffic between application instances and to ensure a good tolerance to failures. A ReplicaSet

will keep up and running the wanted number of replicas.

3.2.8 Design of the Operator Levels

The operator we designed reaches the fifth and final stage of the Operator Framework, covering all the operator capability levels shown in Figure 2.7.

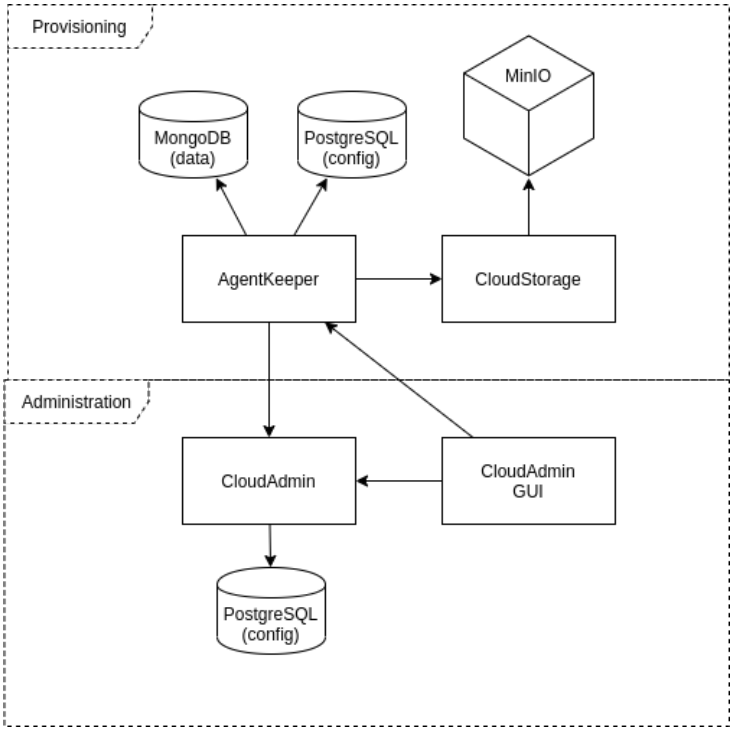


Figure 3.10: IotSnap Root architecture diagram.

The first level, basic install, is covered by the deployment of all the necessary resources in an installation of IotSnap Root, that you can see in Figure 3.10. The Root operator parses the stack specifications from a custom resource file and deploys all the components in the provisioning and administration areas of IotSnap. The deployment is application specific, by awaiting for dependencies and double-checking for the correctness of the deployment after each step. For each component the operator creates a Deployment, specifying the Pod and the ReplicaSet, the corresponding Service, a Route if the component has to be exposed on Internet and a Secret for Databases passwords. Also, it automatically handles the labels and environment variable assembly. We discuss the implementation of this stage in Section 4.3.

No particular feature is instead to be implemented for the second level (seamless upgrade), since Extra Red wants to version all its applications together with the operator: A specific version of the operator must correspond to specific versions of the applications, and the company will certify this set of versions. The operator will be updated according to the package purchased by the customer. The operator update will bring applications updates as well as the rules to update services at runtime from one version of the operator to another. This choice does not allow for smooth and constant updates, but guarantees a stable and certified deployment of the services purchased.

The third level concerns lifecycle management and backup. Regarding the lifecycle, the operator takes care of keeping all components up and running. We designed the component recovery for both the involuntary elimination (even after a crash) and the unwanted modification of the state of the components. To reach this level the operator should also take care of updating the components when the user rollout an update of the custom resource with a new configuration. The implementation details are explained in the Section 4.4 of this document. For what concerns backup (Section 3.3) instead, we decided to save application data to an S3 storage for later recovery. We achieve this feature by exploiting a sidecar container (Figure 3.11) that reads data from the PVC attached to the Pod, encrypt and compress the data, and pushes the result to an external S3 storage, which is currently to be implemented by a MinIO instance (as the latter provides S3-compliant APIs).

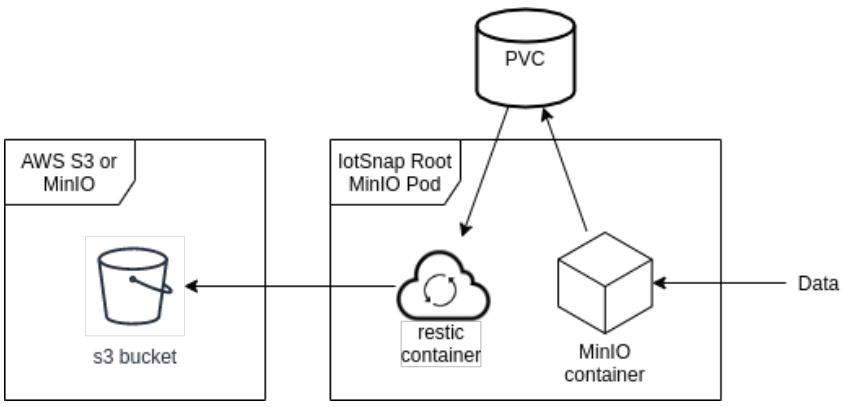


Figure 3.11: Backup service diagram.

In the fourth level we have disposed monitoring mechanisms for deployed components. As shown in Figure 3.12, the monitoring is performed by a Telegraf service that collects not only system but also application metrics from components and inserts them into an InfluxDB database. Other components in the Monitoring area of IotSnap, external to the Root ecosystem, will read data from this instance of InfluxDB and provide control dashboards and an alerting service. Furthermore the operator will allow to see the overall resources used by the IotSnap Root components. You can explore these implementation details in the Section 4.6.

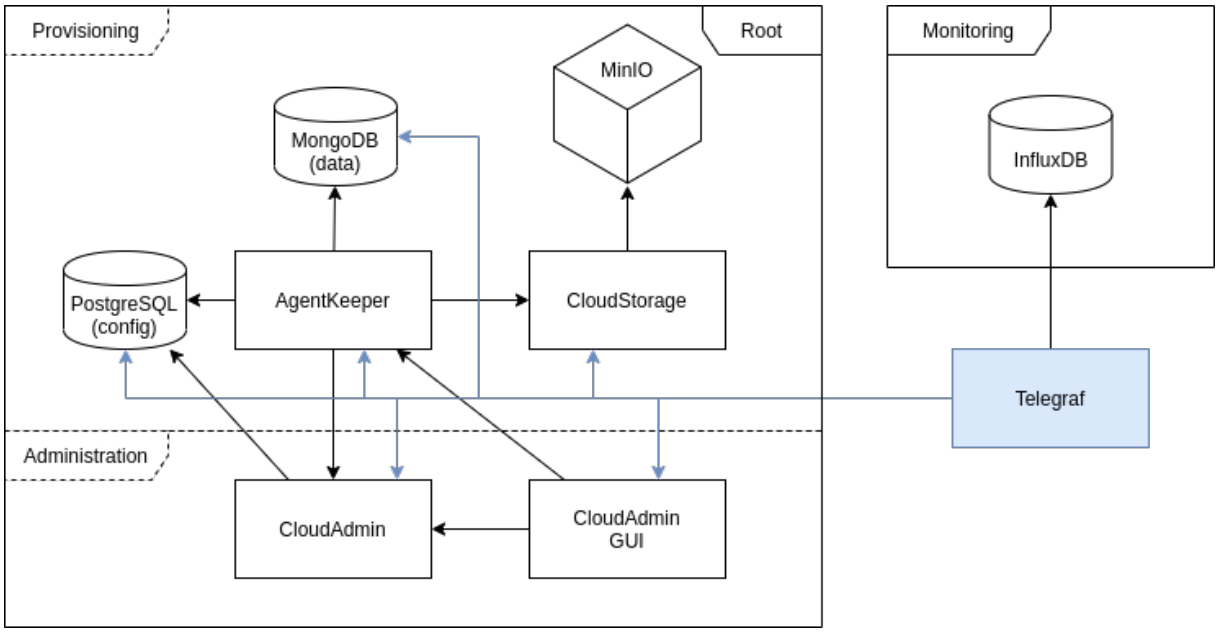


Figure 3.12: Monitoring service architecture diagram.

The fifth phase is reached when the operator performs all the tasks necessary to automate a custom resource to the point that no manual intervention is required. At the time of writing, it is not yet clear what all the necessary operations will be on the IotSnap platform, as it is still under development. Furthermore, the operator is part of a complex automation pipeline system, and the automation of some processes, as far as Root resources are concerned, is outside the scope of the operator. We focused on the automation of the tasks for which an external automation was not already planned or for which we considered it important to move the scope on the operator. These include the acquisition of control over the ConfigMaps and the tuning of the configurations. We talk

about ConfigMaps in Section 4.4, as this operation extends the recovery tasks to these resources too. The configurations tuning will be integrated in the deployment context instead, so we talk about it in Section 4.3.

3.3 Exploiting Protocols for Planning Backup

To design and implement the backup feature of the IotSnap Root components we exploited the formal model provided by management protocols and supported by the Barrel tool (Section 2.5). We first designed a model representing the backup's components and the operations they were supposed to perform. After validating the model, we exploited such model to complete the design of the backup architecture, including also Kubernetes objects and their interaction.

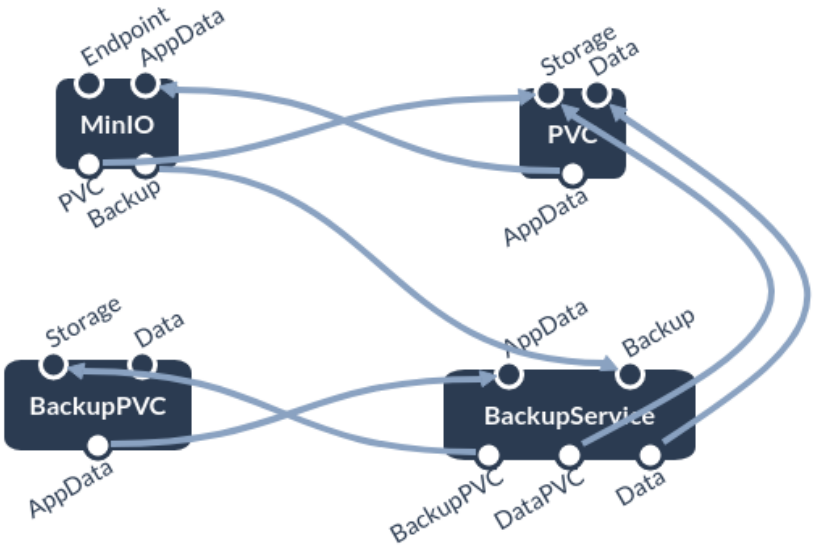


Figure 3.13: Application topology diagram generated with barrel. Rounded squares represent the application components and arrows the dependencies between requirements, the blue bullets on the components, and capabilities, the withe bullets.

We modeled in TOSCA three components of IotSnap Root: A storage service (for both backup and application data storage), an application service (MinIO, to be backed up), and the backup service itself. For each of those components, we modeled its states and the transitions between those states (Figures 3.13, 3.14, 3.15, and 3.16). Each state and transition was associated with the requirements needed to stay in that state and to

move from one state to another, respectively. Each state was also associated with the capabilities offered by the component in that state. We then exploited Barrel to check the reachability of different states of the three components in order to make sure that all operations were executable as expected. We also verified that for each fault that can possibly occur there was a suitable sequence of recovery steps. Finally, we exploited the model of the components to implement the components into Kubernetes objects. While doing so, we double-checked every implementation step by writing tests to ensure that the implementation was matching the modeled expected behaviour.

Overall, while Barrel has some limitations, being a research prototype, we found that that the performed model-based design and analysis reduced the implementation time needed to get the backup feature. It is worth mentioning that in this project we exploited the formal model provided by management protocols and supported by Barrel also for validating the implementation of the entire deployment, as we will discuss in Section 5.3.

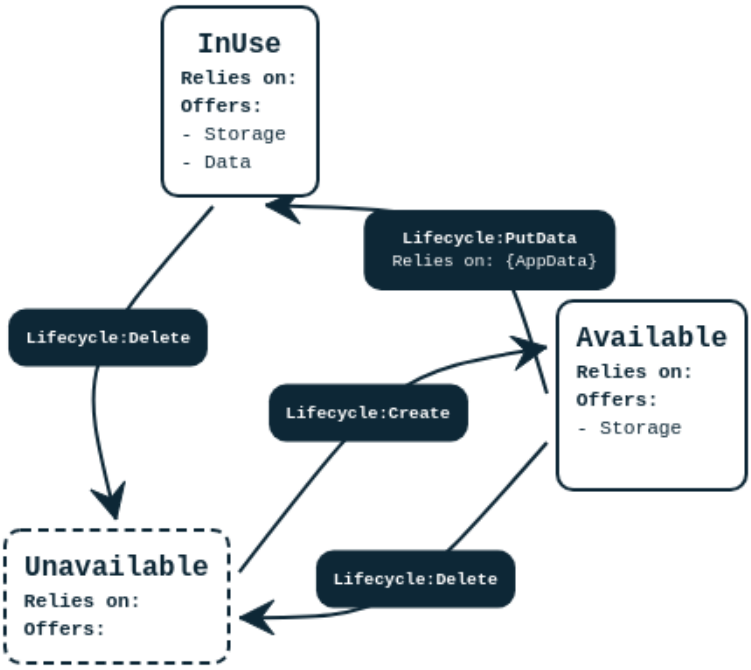


Figure 3.14: Diagram showing the PVC transition schema. PVC represents the storage, for both the MinIO application and the backup service.

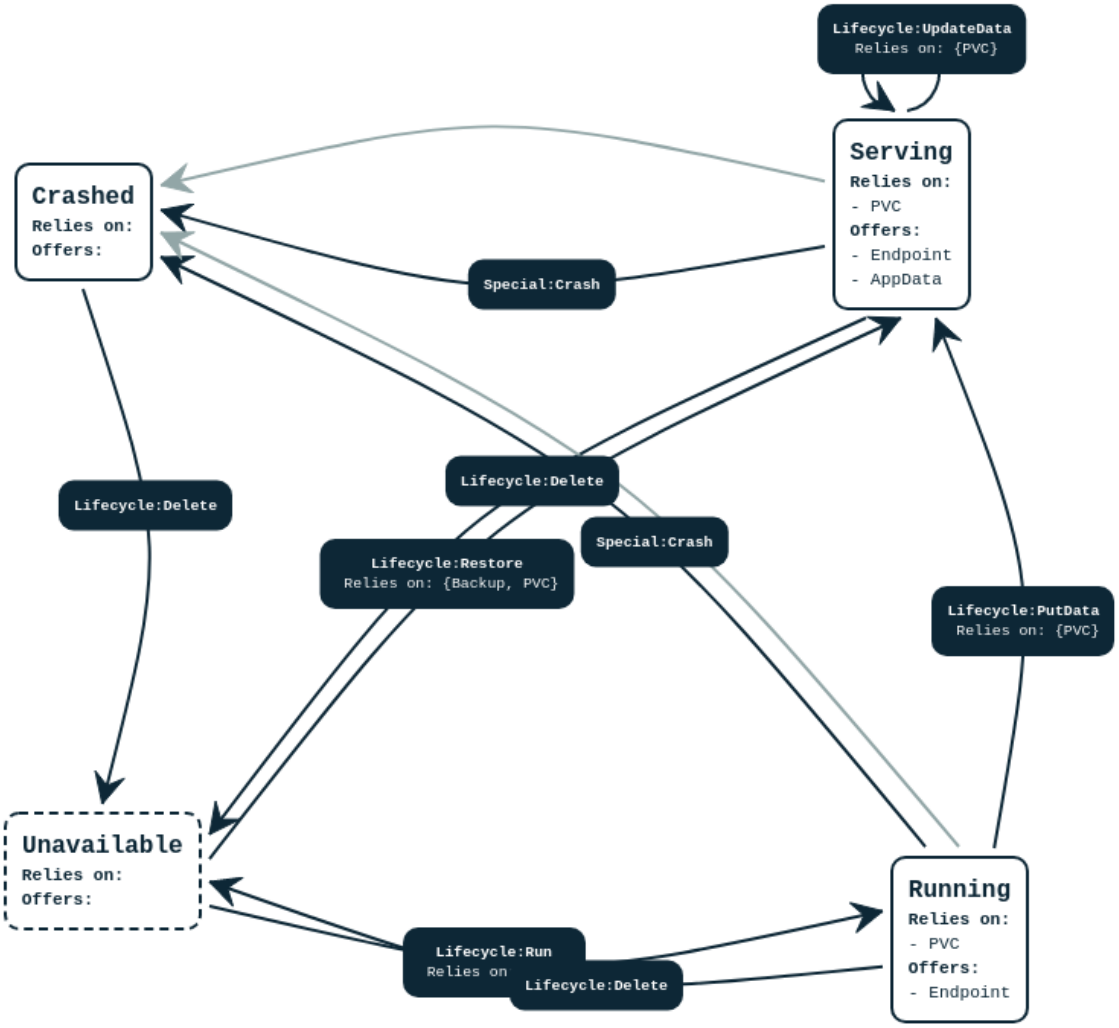


Figure 3.15: Diagram showing the MinIO transition schema. MinIO is the application we want to backup.

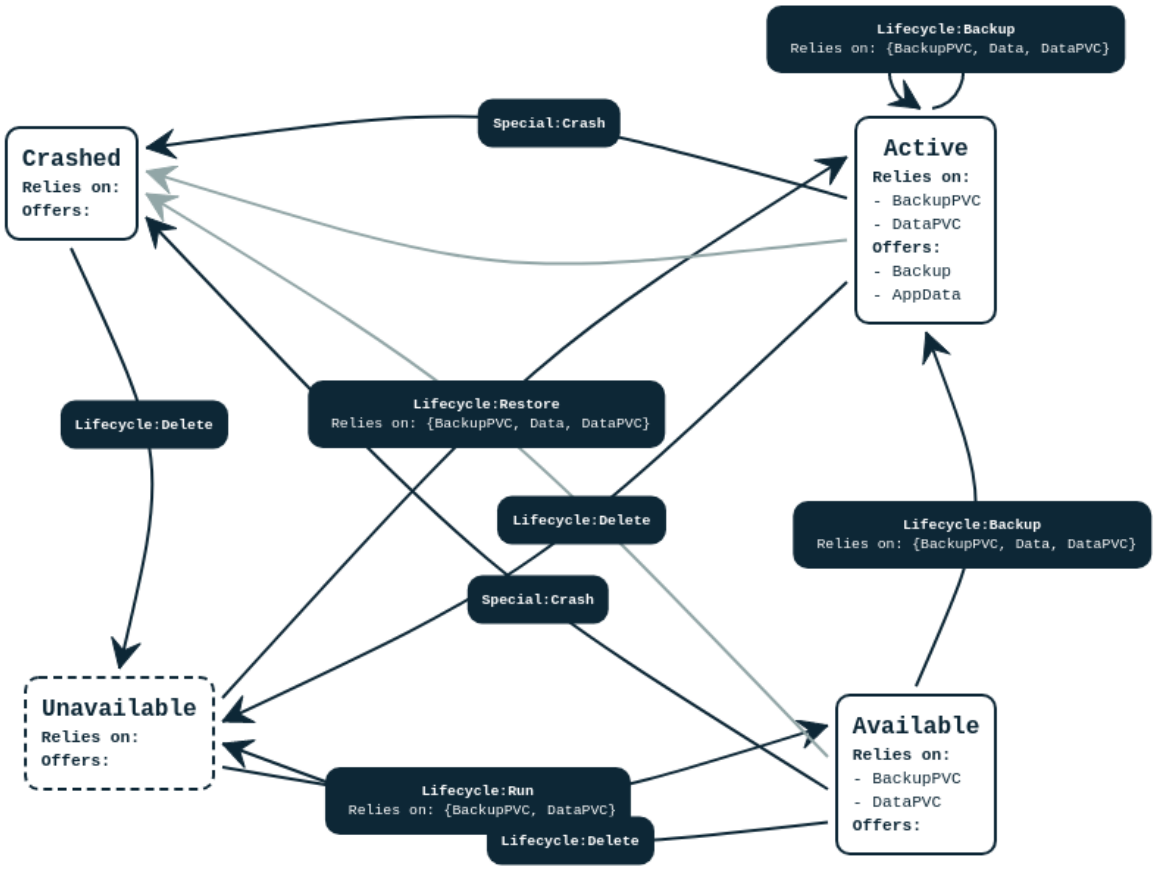


Figure 3.16: Diagram showing the BackupService states and the transitions between them. Each state can have requirements (relies on) and capabilities (offers). Transitions may have requirements that must be satisfied to move from a state to another. Requirements on transition and states may not coincide: For example the Backup transition has a Data requirement that is needed only to move from the state Available to the state Active, or to loop on Active, but is not required to stay in the Active state.

Chapter 4

Implementation of the IotSnap Root Operator

4.1 Technologies used

Here we discuss the programming language and framework we have chosen to implement the IotSnap Root Operator.

4.1.1 SDK Choice

Choosing the SDK is not obvious when writing a Kubernetes Operator. There are two official SDKs, which development continues in parallel. The first is Kubebuilder, developed by Kubernetes SIG and born in March 2018 [30], the second is Operator SDK, born in February of the same year by CoreOS [19], acquired by Red Hat just a couple of week before [53].

Google and the SIG group of Kubernetes are pushing on Kubebuilder. However, Operator SDK is developed by CoreOS [57], the group behind the development of many components of and for Kubernetes, including etcd database, the core element of Kubernetes [54]. In addition, Operator SDK is part of the Operator Framework, an ecosystem of services oriented to supporting operators. Among these services we also find the OperatorHub, a distribution tool for all operators, even those developed with Kubebuilder, minor frameworks or without the use of frameworks.

Here we report the main technical differences between the two frameworks.

- While Kubebuilder supports only Go, Operator SDK supports also Ansible and Helm operators.
- Operator SDK integrates with the Operator Lifecycle Manager (OLM), that manages operators, providing for example live upgrades.
- Operator SDK includes an end-to-end testing framework that simplifies testing the operator against an actual cluster. Kubebuilder includes an envtest package that allows operator developers to run simple tests with a standalone etcd and apiserver.
- Kubebuilder scaffolds a Makefile while Operator SDK is currently using built-in subcommands, like build, test, and so on. Anyway, the Operator SDK team will likely be migrating to a Makefile-based approach in the future [43].
- Kubebuilder uses Kustomize to build deployment manifests while Operator SDK uses static files with placeholders.
- Kubebuilder has an improved support for admission and webhooks, that the Operator SDK has not.

The fact that Extra Red is an OpenShift partner and that the operator will have to run on an OpenShift cluster led us to choose the Operator SDK solution. This should facilitate the development and execution of the operator and ensure technical support by Red Hat.

4.1.2 Programming Language

The Operator Framework supports three languages for writing an operator i.e., Helm, Ansible, and Go. While we immediately discarded Helm because it only allows developing operators whose maximum level is 2 (Figure 4.1), we initially took Ansible into consideration as there was already an automation in Ansible that played part of the operator's role. In particular, the first part of the internship consisted in developing some simple Ansible and Go operators to evaluate their advantages and disadvantages. After careful analysis we decided to proceed with the operator in Go. Go is the recommended language for

operator development and provides a programmatic approach that we considered much more comfortable than Ansible. Finally, aiming in future to create a set of operators for the entire platform, Extra Red considered Go the most forward-looking choice.

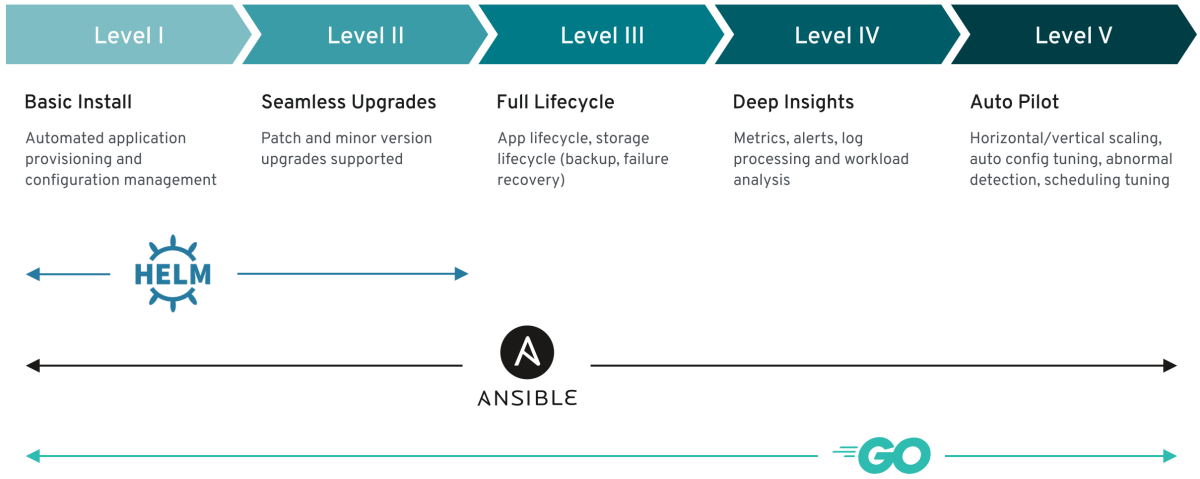


Figure 4.1: Operator capability levels and language supported.

4.2 Organization of the Project

The operator consists of about 1700 rows of code for the controller and the APIs, to which is added the scaffolding code generated by the operator-sdk. The folder structure is organized as in Figure 4.2.

```
build/   deploy/   go.sum    pkg/         test/      version/
cmd/     go.mod    images/   README.md    tools.go
```

Figure 4.2: Files and directory in the root directory of the IotSnap Root Operator project.

`go.mod` and `go.sum` contains metadata of the Go module and the dependencies list. `tools.go` is an empty file on which you can place any runtime dependencies as imports to force Go to download and install them.

build, version, and cmd

The `build` directory (Figure 4.3) contains the Dockerfile to build the operator's image and the scripts to be executed inside the container.

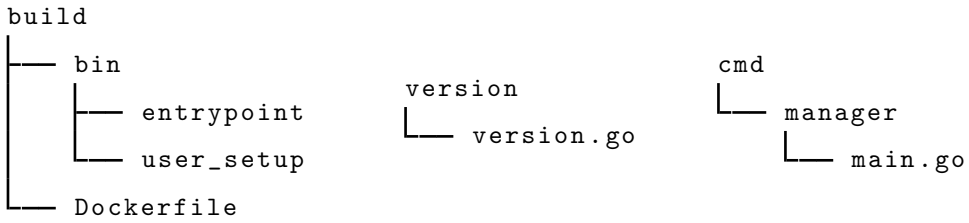


Figure 4.3: Expanded tree of the `build`, `version`, and `cmd` directories.

`version.go` contains a single variable storing the application version.

`cmd/manager/main.go` (Figure 4.3) is the operator's main instead. It is a scaffolding file though, usually you do not modify it. Anyway, in this case we had to modify it to include Routes, that are OpenShift objects and hence not imported. When using only Kubernetes native objects there is no need to change this file. To add new resources you need to import them and then call the `AddToScheme` function on them, as in Listing 4.1.

```
1  import routev1 "github.com/openshift/api/route/v1"
2
3  func main() {
4      // ...
5
6      if err := routev1.AddToScheme(mgr.GetScheme()); err != nil {
7          log.Error(err, "")
8          os.Exit(1)
9      }
10
11     // ...
12 }
```

Listing 4.1: Usage of the `AddToScheme` function to add new resources definitions to the operator.

deploy

```
deploy
├── cluster_role.yaml
├── crds
│   ├── iotsnap.red.extrasys.it_iotsnaproots_crd.yaml
│   └── iotsnap.red.extrasys.it_v1_iotsnaproot_cr.yaml.yaml
├── operator.yaml
├── resources
│   ├── configmaps
│   │   ├── agentkeeper-config.yaml
│   │   ├── backupserver-config.yaml
│   │   ├── cloudadmin-config.yaml
│   │   ├── cloudadmingui-config.yaml
│   │   ├── cloudstorage-config.yaml
│   │   ├── influxdbmon-config.yaml
│   │   ├── minio-config.yaml
│   │   └── telegrafmon-config.yaml
│   ├── kustomization.yaml
│   └── pvc
│       ├── agentkeeper-pvc.yaml
│       ├── backupserver-pvc.yaml
│       ├── influxdbmon-pvc.yaml
│       ├── minio-pvc.yaml
│       ├── mongo-pvc.yaml
│       └── postgres-pvc.yaml
├── role_binding.yaml
├── role.yaml
└── service_account.yaml
```

Figure 4.4: Expanded tree of the ***deploy*** *directory.*

`deploy` (Figure 4.4) contains all the yaml files needed to deploy the operator.

`crds/iotsnap.red.extrasys.it_iotsnaproots_crd.yaml` is the Custom Resources Definition. It defines a new Kubernetes resources called *IotSnapRoot* of the API group *iotsnaproots*, that allows to deploy the Root stack.

`crd/iotsnap.red.extrasys.it_v1_iotsnaproot_cr.yaml` is a custom resource specification file. It contains all the details of an example custom resource iotsnap-root. If we deploy this yaml while the operator is running on the cluster, it will recognise the custom resource and will start to deploy and manage its component. Resources can be deployed with `oc apply -f file.yaml` and deleted with `oc delete -f file.yaml`. When the custom resource is deleted, all the associated resources (Pods, Deployments, ReplicaSets, Services, Routes, and so on) will be deleted. We will explore the content of this file in Section 4.3.1.

`resources` contains all the external resource, i.e., ConfigMaps and PVCs for the Root microservices. They are only meant for debugging purposing, while the actual resources will be assembled and deployed by the Ansible automation according to the customer plan. The resources in this directory represent an example cutomer with a cloud configured for the operator development. The `kustomization.yaml` file in this directory aggregates all this resources for bulk actions. For example, we can deploy all the external resources with `oc apply -k deply/resources` instead of having to call `oc apply -f file` on each yaml file manually.

`operator.yaml`, `role_binding.yaml`, `role.yaml`, and `service_account.yaml` are used for the operator deployment. The first file contains all the definition to deploy the operator in the cluster. The other three files give the operator the permissions needed to operate on the cluster.

Finally, `cluster_role.yaml` is used to give the right permissions to an user on a namespace. It is needed to allow users to view, deploy, and manage IotSnapRoot resources. Since they are not Kubernetes core resources there is no definition on what an user can do or not with them. By default only global cluster admin can manage custom resources. With this ClusterRoles we give to users the permissions to see the resources and to admin and editors the permissions to create and manage them. This permissions are split in two ClusterRole resources, inside the same file.

```
1 kind: ClusterRole
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   name: iotsnap-root-edit-admin
```

```
5   labels:
6     rbac.authorization.k8s.io/aggregate-to-admin: "true"
7     rbac.authorization.k8s.io/aggregate-to-edit: "true"
8 rules:
9   - apiGroups:
10      - iotsnap.red.extrasys.it
11    resources:
12      - "*"
13    verbs:
14      - create
15      - delete
16      - get
17      - list
18      - patch
19      - update
20      - watch
```

Listing 4.2: ClusterRole that gives to user of the group `admin` *and* `edit` *the permissions to add and edit IotSnapRoot resources.*

The labels of the ClusterRole in Listing 4.2 define the entities to which the rules apply. We find a single rule that gives the permissions to perform the actions listed under "verbs" on all the resources of the API group iotsnap.red.extrasys.it.

```
1 kind: ClusterRole
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   name: iotsnap-root-view
5   labels:
6     rbac.authorization.k8s.io/aggregate-to-view: "true"
7     rbac.authorization.k8s.io/aggregate-to-cluster-reader: "true"
8 rules:
9   - apiGroups:
10      - iotsnap.red.extrasys.it
11    resources:
12      - "*"
13    verbs:
14      - get
```

```
15        - list
16        - watch
```

Listing 4.3: ClusterRole that gives to user that have a `view` *role in this namespace the permissions to read and watch IotSnapRoot resources.*

The second ClusterRole in the file (Listing 4.3) has the same structure and gives permissions to viewers. They are less then the admin and edit permissions, as we can see from the smaller verbs list, and are only action to get and view resources, not to edit or create them.

We created the `cluster_role.yaml` and the files in the `resources` folder from scratch, while the other file were generated by the operator-sdk. The Custom Resource Definition is automatically updated starting from the API definition we provide, and must not be manually modified. Instead, the custom resource contains only the skeleton and must be kept up to date manually with example values, according to the APIs we write. The files needed for the operator deployment are skeletons too, but requires fewer and wiser edits with respect to the custom resource. We need to change the operator name and image, add permissions for further API groups we need (e.g., OpenShift's Routes) and remove unnecessary permissions to respect the least privilege security principle.

pkg

`pkg` is the core of the operator. It is divided in APIs and controller (Figure 4.5).

Inside the APIs directory we found `addtoscheme_iotsnap_v1.go` and `apis.go` that act as glue between the main and the APIs, `iotsnap/group.go` that are just metadata for a correct versioning, and the `v1` directory that contains the APIs itself. The most important file is `root_types.go`, inside the `v1` directory. It contains the actual APIs definition. `doc.go` contains metadata and defines the package version. `register.go` is again a bridge file between the main and the types file. Finally, `zz_generated.deepcopy.go` is a file generated by the operator-sdk starting from the APIs specifications in `root_types.go`. It verifies the correctness of the custom resource when they are deployed. If some parameter is missing or does not match the type the custom resource is rejected from this code. We will deepen the `root_types.go` file when discussing the APIs in the

Section 4.3.1 of this document.

```
pkg
├── apis
│   ├── addtoscheme_iotsnap_v1.go
│   ├── apis.go
│   └── iotsnap
│       ├── group.go
│       └── v1
│           ├── doc.go
│           ├── register.go
│           ├── root_types.go
│           └── zz_generated.deepcopy.go
└── controller
    ├── add_root.go
    ├── controller.go
    └── root
        ├── agentkeeper.go
        ├── backup.go
        ├── cloudadmin.go
        ├── cloudadmingui.go
        ├── cloudstorage.go
        ├── component.go
        ├── influxdbmon.go
        ├── minio.go
        ├── mongo.go
        ├── objects.go
        ├── postgres.go
        ├── reconcile.go
        ├── root_controller.go
        └── telegrafmon.go
```

Figure 4.5: Expanded tree of the `pkg` directory.

As for the APIs, in the `controller` directory we find some glue files i.e., `add_root.go` and `controller.go`. The main directory, `root`, contains the controller itself. We organized this directory creating a file for each Root component. This are `agentkeeper`,

`cloudadmin`, `cloudadmingui`, `cloudstorage`, `minio`, `mongo`, and `postgres`. `backup` defines the backup component and `influxdbmon` and `telegrafmon` are the monitoring components. Each component file has a similar structure. We will see an example in the Section 4.3.2, while exploring the controller implementation. `component.go` contains all the code to generate a component representation, starting from the definitions in the components files. `objects.go` contains functions to generate Kubernetes object given a component. This object will be deployed through the Kubernetes APIs. `reconcile.go` contains utility for the custom resource reconciliation. Those are functions to deploy, update, and delete the Kubernetes objects. Finally, `root_controller` is the actual controller. It contains some scaffolding code and the main workflow to reconcile the custom resource.

test

```
test
├── e2e
│   ├── deployment.go
│   ├── main_test.go
│   ├── recovery.go
│   ├── root_test.go
│   ├── update.go
│   └── utils.go
├── unit
│   ├── root_controller_test.go
│   └── utils.go
└── utils
    └── test_utils.go
```

Figure 4.6: Expanded tree of the `test` directory.

The `test` directory contains the operator's tests. There are both end-to-and tests and unit tests (Figure 4.6). This latter runs on the code and tests operator's response to events. The first tests are meant to be on a real cluster instead and tests the operator behavior with the actual application by automate testing cluster operations. `utils` contains some

common utilities like example resources to be deployed, expected responses, functions to verify the correctness of a result and so on. We talk about unit tests in the Section 5.1 of this document and about end-to-end testing in Section 5.2.

4.3 Deployment

As shown in Section 4.2, the code is divided in two main components: APIs and controller. APIs describe the custom resource IotSnapRoot

The IotSnap Root subsystem is modeled as custom resource, described by the operator's APIs that defines the `spec` of this new resource. The custom resource contains information about the stack and the single microservices. Stack information are, for example, repositories endpoint for source images, global configurations and label. The microservice `spec` defines specific environment variables and labels for the microservice, custom placeholder values, ConfigMaps and volumes references, and other service-specific information.

The controller's code is the place for the business logic, as it is aimed to manage the IotSnap Root resources. It deploys and manages the lifecycle of the IotSnap Root microservices. Each of them is made by:

- a Deployment, comprehending Pods and the associated ReplicaSet;
- a Service to expose the application on the network;
- a Route if the application needs to be exposed on Internet;
- a Secret for the databases;
- a provided Persistent Volume Claim for the microservices with persistent storage;
- a ConfigMap for microservices that requires configurations.

Moreover, PostgreSQL and MongoDB databases must be initialized with an `initdb` script at first rum that prepares the users and tables needed by the other microservices. We run `initdb` scripts in init-containers, a one-shot container execution that prepares the Pod for the main container, containing the application.

4.3.1 APIs and Custom Resource

```
1  package v1
2
3  import metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
4
5  // =======================
6  // == EDIT HERE ==========
7  // IotSnapRootSpec is the spec definition of the IotSnapRoot CR.
8  type IotSnapRootSpec struct {
9  }
10
11 // IotSnapRootStatus defines the observed state of an IotSnapRoot
       instance,
12 // and contains the status of each component.
13 type IotSnapRootStatus struct {
14 }
15
16 // == STOP EDIT ==========
17 // =======================
18
19 // +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.
       Object
20
21 // IotSnapRoot is the Schema for the iotsnaproots API.
22 // +kubebuilder:subresource:status
23 // +kubebuilder:resource:path=iotsnaproots,scope=Namespaced
24 type IotSnapRoot struct {
25     metav1.TypeMeta   `json:",inline"`
26     metav1.ObjectMeta `json:"metadata,omitempty"`
27
28     Spec   IotSnapRootSpec   `json:"spec,omitempty"`
29     Status IotSnapRootStatus `json:"status,omitempty"`
30 }
31
32 // +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.
       Object
33
```

```
34 // IotSnapRootList contains a list of IotSnapRoot instances.
35 type IotSnapRootList struct {
36     metav1.TypeMeta `json:",inline"`
37     metav1.ListMeta `json:"metadata,omitempty"`
38     Items           []IotSnapRoot `json:"items"`
39 }
40
41 func init() {
42     SchemeBuilder.Register(&IotSnapRoot{}, &IotSnapRootList{})
43 }
```

Listing 4.4: `root_types.go` file content. It is the main API file and contains the definition of the IotSnapRoot custom resource.

The `root_types.go` file (Listing 4.4) contains some scaffolding code that is the skeleton of the Custom Resource definition and a section that define the resource itself (the one between the "EDIT HERE" and "STOP EDIT" comments). We will edit this part. We are interested in particular in the `IotSnapRootSpec` and `IotSnapRootStatus` structs.

The `+kubebuilder:resource:path` instruction (row 23) tells the name to use for the Custom Resource to the kubebuilder code generator integrated in the operator-sdk. In this case our resources will be `iotsnaproots`, that means that we can for example retrieve our custom resources with `oc get iotsnaproots`, likewise we do with Pods and other Kubernetes resources.

```
1 apiVersion: iotsnap.red.extrasys.it/v1
2 kind: IotSnapRoot
3 metadata:
4   name: iotsnap-root
5 spec:
```

Listing 4.5: First rows of the `crd/iotsnap.red.extrasys.it_v1_iotsnaproot_cr.yaml` file, i.e., the file to deploy an IotSnapRoot resource. This first part of the file is generated by the `+kubebuilder` instructions in Listing 4.4.

The resource Kind to be specified in the custom resource yaml file will be the same of the struct under this `+kubebuilder` instructions, i.e., IotSnapRoot (row 24). The custom resource example in Listing 4.5 shows how the Kind (row 2) is the one specified in

`root_types.go` and the API version corresponds to the package name of the Root types.

The `spec` section, as for the other Kubernetes resources, defines the deployed object. In our case define how the IotSnapRoot object must be implemented. It corresponds to the `IotSnapRootSpec` structure of the `root_types.go` file.

Spec

```
1  // IotSnapRootSpec is the spec definition of the IotSnapRoot CR.
2  type IotSnapRootSpec struct {
3      PersistentDataClaim     string            `json:"persistent-data-claim,omitempty
           "`
4      InstallationDataVersion string            `json:"installation-data-version,
           omitempty"`
5      Registry                string            `json:"registry,omitempty"`
6      Labels                  map[string]string `json:"labels,omitempty"`
7      Env                     map[string]string `json:"env,omitempty"`
8      Backup                  bool              `json:"backup,omitempty"`
9      Monitoring              bool              `json:"monitoring,omitempty"`
10     BackupServer            []ComponentSpec   `json:"backupserver,omitempty"`
11     Telegrafmon             []ComponentSpec   `json:"telegrafmon,omitempty"`
12     Influxdbmon             []ComponentSpec   `json:"influxdbmon,omitempty"`
13     Postgres                []ComponentSpec   `json:"postgres,omitempty"`
14     Mongo                   []ComponentSpec   `json:"mongo,omitempty"`
15     Minio                   []ComponentSpec   `json:"minio,omitempty"`
16     Cloudstorage            []ComponentSpec   `json:"cloudstorage,omitempty"`
17     Cloudadmin              []ComponentSpec   `json:"cloudadmin,omitempty"`
18     Agentkeeper             []ComponentSpec   `json:"agentkeeper,omitempty"`
19     Cloudadmingui           []ComponentSpec   `json:"cloudadmingui,omitempty"`
20 }
```

Listing 4.6: The `IotSnapRootSpec` structure in `root_types.go`.

The `IotSnapRootSpec` struct (Listing 4.6) define the custom resource `spec`. It correspond exactly to the fields that we expect to find under the `spec` field of the custom resource yaml file.

As we can see the structure is made of three columns. The first is the variable name, i.e., the name we will use in the Go controller to refer to this field. The second column is the type of the variable. The third column is optional in Go. It contains information about how to serialize or parse this structure from or to a json. After the `json:` part, we found the name of the property in the json file and "omitempty" if the field is optional. It is called omit empty since in the serialization will be omitted if it is empty or has the default value. Since YAML is a superset of JSON, this is used to serialize this structure

in a Custom Resource Definition yaml file and to parse a custom resource yaml file.

Each Root component is represented by a `ComponentSpec`. They are arrays, since by design we can have multiple distinct instances (not replicas) of the same component, that may have different ConfigMaps, env, labels and so on. This is for testing purposes, to implement A/B testing or to set up a develop environment for a component. If no instances are passed, a single instance with default values is used, that is the production configuration.

Together with the components `spec` (rows 13-19 the IotSnap Root components and rows 10-12 backup and monitoring components), the `Spec` structure contains the following variables.

- `PersistentDataClaim` (row 3), a string representing the name of the global ReadWriteMany PVC to be used. If not specified, the operator uses a ReadWriteOnce PVC for each component. The claim name must correspond to an existent PVC.

- `InstallationDataVersion` (row 4), the tag the operator should use for the installation data images. If not provided, the default tag will be used, that is the one for which the operator is certified.

- The `Registry` URI (row 5) for both the installation data and application images. If not provided, the company's Quay instance is used.

- `Labels` to be applied to all the objects (row 6). Each object (Service, Deployment, Pod, container, Route, Secret) has its own labels, that are component references to be referred from another object. For example, a Service can find its Pod(s) using the component-specific labels. The operator add its own labels to its object that provide information about the namespace, the operator name and version, the custom resource name, the resource name, and so on. Then additional component-specific labels can be added through the `ComponentSpec`. Finally, this global `Labels` variable allows to add labels to all the components. For example, they may include information about the customer and its cloud coordinates.

- `Env` (row 7) to be applied to all the containers. Each containers has already its specific env, and other component-specific env variables can be specified also through

the `ComponentSpec`. Anyway, most of the Root containers use the same set of env variables for the customers information and cloud specifications. This global env will ease this process, ensuring consistency across components.

- `Backup` and `Monitoring` (rows 8 and 9) are boolean values that activate or deactivate the backup and monitoring features. They are false by default, i.e., disabled.

ComponentSpec

```
1  // ComponentSpec defines the desired state of components,
2  // and corresponds to the component section of the CR's spec.
3  type ComponentSpec struct {
4      PersistentDataClaim     string            `json:"persistent-data-claim,omitempty
           "`
5      Name                    string            `json:"name,omitempty"`
6      Replicas                int32             `json:"replicas,omitempty"`
7      ConfigMap               string            `json:"configmap,omitempty"`
8      InstallationDataVersion string            `json:"installation-data-version,
           omitempty"`
9      PersistentDataPath      string            `json:"persistent-data-path,omitempty"`
10     Labels                  map[string]string `json:"labels,omitempty"`
11     Env                     map[string]string `json:"env,omitempty"`
12 }
```

Listing 4.7: The `ComponentSpec` structure in `root_types.go`.

`ComponentSpec` defines an IotSnap Root component. As we can see in Listing 4.7, it contains the following fields.

- `PersistentDataClaim` (row 4) is a string representing the name of the component's PVC. It must correspond to an existing PVC. If not provided the operator assume that a PVC called `componentname-pvc` (e.g., `postgres-pvc`) exists. This is how PVCs deployed by the Ansible automation are called.

- The component `Name` (row 5). By default is the application name, e.g., `postgres`. If multiple instances are provided, they will have a progressive number, i.e., `postgres`, `postgres2`, `postgres3`, and so on. It can be overridden with this field.

- The number of `Replicas` we want for a component (row 6). A single replica by default.

- The `ConfigMap` name (row 7). As for PVCs, ConfigMaps are provided by the automation. By default, the CM will be named `componentname-config` (e.g., `postgres-config`), but you can override it for development and testing purposes, to use a different one with other configurations.

- `InstallationDataVersion` (row 8), the tag the operator should use for the installation data image of this component. If not provided, the default tag will be used, that is the one for which the operator is certified. If the component does not have any installation data the field is ignored. This field overrides the global `InstallationDataVersion` if both are provided.

- `PersistentDataPath` (row 9) specifies the sub path of the PVC that the component will use to store its data. It is not necessary for ReadWriteOnce PVCs, that are dedicated to the component. Instead, when using a global ReadWriteMany PVC for all the components we must differentiate the path in which components will write. If not provided it is `componentname` (e.g., `postgres`), even when we have a dedicated PVC.

- `Labels` to be applied to all the component objects (row 10). Each object (Service, Deployment, Pod, container, Route, Secret) has its own labels that we use to reference this object from another one. For example, the PostgreSQL Service can find its Pod(s) using the component-specific labels, that says that this object belongs to the PostgreSQL stack. We can add additional component-specific labels through this field. Additionally, the component will have the operator's label and the global labels provided form the custom resource `spec` (see Section 4.3.1). In case of conflict the component labels override the global label.

- `Env` to be applied to the component container (row 11). Each containers has already its specific env, for example the databases have a Secret reference. Moreover, global env will be applied. This field can be used for development and testing purposes, to test custom setup of the application, e.g., by overriding the secret value or injecting additional environment variables, or you can use it to override global env.

The yaml code in Listing 4.8 is the `spec` of an example custom resource. We can see

all the global fields of the `IotSnapRootSpec` section (rows 3-14). Then we will find the components `spec`. In this example we show only CloudAdmin GUI (rows 16-36), that uses all the fields in the `ComponentSpec`, but in the same way you can add specifications for all the other components. We have two CloudAdmin GUI instances (rows 17-26 and rows 27-36). For each instance we can specify all the fields or only those we want to override from the default specification. When deploying multiple instances the user must specify at least one field for each instance. If the user passes no instances, a single default instance is deployed.

```
1  spec:
2    # all the optional fields have their default value set
3    persistent-data-claim: persistent-data    # optional
4    registry: quay.extrasys.it/red-eriot      # optional
5    labels:                                   # optional
6      instance_name: root-test
7      cloud_name: root-test-cloud
8      release: 1.0.0
9      stage: alpha
10   installation-data-version: 1.0.8-dev      # optional
11   env:                                      # optional
12     CENTRAL_CLOUD_ADDRESS: "whatever"
13   backup: false                             # optional
14   monitoring: false                         # optional
15
16   cloudadmingui:  # optional
17     - name: cloudadmingui                          # optional
18       replicas: 1                                  # optional
19       configmap: cloudadmingui-config              # optional
20       installation-data-version: 1.0.8-dev         # optional
21       persistent-data-claim: cloudadmingui-pvc     # optional
22       persistent-data-path: cloudadmingui          # optional
23       labels:                                      # optional
24         key: value
25       env:                                         # optional
26         key: value
27     - name: cloudadmingui2                         # optional
```

```
28        replicas: 1                                    # optional
29        configmap: cloudadmingui2-config               # optional
30        installation-data-version: 1.0.8-dev           # optional
31        persistent-data-claim: cloudadmingui2-pvc      # optional
32        persistent-data-path: cloudadmingui2           # optional
33        labels:                                        # optional
34          key: value
35        env:                                           # optional
36          key: value
```

Listing 4.8: `spec` section of the `crd/iotsnap.red.extrasys.it_v1_iotsnaproot_cr.yaml` file, i.e., the file to deploy an IotSnapRoot resource. All the keys are associated with their default values.

Status

```
1  // ComponentStatus defines the saved state of IotSnapRoot components.
2  type ComponentStatus struct {
3      ConfigMapData *map[string]string `json:"configmap,omitempty"`
4  }
5
6  // IotSnapRootStatus defines the observed state of an IotSnapRoot instance,
7  // and contains the status of each component.
8  type IotSnapRootStatus struct {
9      BackupServer  []ComponentStatus `json:"backupserver,omitempty"`
10     Telegrafmon   []ComponentStatus `json:"telegrafmon,omitempty"`
11     Influxdbmon   []ComponentStatus `json:"influxdbmon,omitempty"`
12     Minio         []ComponentStatus `json:"minio,omitempty"`
13     Cloudstorage  []ComponentStatus `json:"cloudstorage,omitempty"`
14     Cloudadmin    []ComponentStatus `json:"cloudadmin,omitempty"`
15     Agentkeeper   []ComponentStatus `json:"agentkeeper,omitempty"`
16     Cloudadmingui []ComponentStatus `json:"cloudadmingui,omitempty"`
17 }
```

Listing 4.9: The `IotSnapRootStatus` (rows 6-17) and `ComponentStatus` (rows 1-4) structures in `root_types.go`.

The `IotSnapRootStatus` structure (rows 6-17 of the Listing 4.9) describe the observed status of the IotSnapRoot custom resource.

As we will see in section 4.4, the Status is used to store components configuration to restore them in case of changes. This information are stored in the `ComponentStatus`

struct. This object includes only a field, `ConfigMapData`, but in the future it could be extended with other components information.

`IotSnapRootStatus` contains a list of `ComponentStatus` elements for each component that has a ConfigMap. Each element in the list corresponds to a component instance. In the custom resource provided before for example we have two CloudAdmin GUI instances, so we will have two elements in the status list. Since by default we have a single instance for each component, we will also have a single element in this arrays.

4.3.2 Controller

The `root_controller.go` file is the main file of the controller. It starts with some scaffolding functions. Among this we find the `add` function (Listing 4.10) that registers the controller.

```
1  // add adds a new Controller to mgr with r as the reconcile.
       Reconciler
2  func add(mgr manager.Manager, r reconcile.Reconciler) error {
3      // Create a new controller
4      c, err := controller.New("iotsnaproot-controller", mgr,
           controller.Options{Reconciler: r})
5      if err != nil {
6          return err
7      }
8
9      // Watch for changes to IotSnapRoot resources
10     err = c.Watch(&source.Kind{Type: &iotsnapv1.IotSnapRoot{}}, &
           handler.EnqueueRequestForObject{})
11     if err != nil {
12         return err
13     }
14
15     // Watch for changes to owned resources (deployed or passed)
16     resourcesToWatch := []runtime.Object{
17         // deployed
18         &corev1.Secret{},
19         &appsv1.Deployment{},
```

```
20          &corev1.Pod{},
21          &corev1.Service{},
22          &routev1.Route{},
23          // conquered
24          &corev1.ConfigMap{},
25      }
26      for _, resource := range resourcesToWatch {
27          err = c.Watch(&source.Kind{Type: resource}, &handler.
                EnqueueRequestForOwner{
28              IsController: true,
29              OwnerType:    &iotsnapv1.IotSnapRoot{},
30          })
31          if err != nil {
32              return err
33          }
34      }
35
36      return nil
37  }
```

Listing 4.10: The `add` *function in* `root_controller.go`*, that is the main file of the controller.*

In `add` we register the controller and then we start watching some resources. This resources include by default the IotSnapRoot resource kind (rows 9-13). Then, we added all the resource kinds we deploy (rows 15-34): Secret, Deployment, Pod, Service, and Route. Finally, we watch for ConfigMaps (row 24), even though we do not deploy them directly. We conquer them when the corresponding component goes up, by setting the operator's owner reference on them. Then we watch them to prevent changes on configurations.

```
1  // ReconcileIotSnapRoot contains the client and the scheme to
       reconcile an IotSnapRoot object.
2  type ReconcileIotSnapRoot struct {
3      // This client, initialized using mgr.Client() above, is a split
           client
4      // that reads objects from the cache and writes to the apiserver
5      client client.Client
```

```
6       scheme *runtime.Scheme
7   }
```

Listing 4.11: The `ReconcileIotSnapRoot` structure in `root_controller.go`.

After the `add` function we find the `ReconcileIotSnapRoot` structure (Listing 4.11). It contains the client and the scheme needed to reconcile an IotSnapRoot object. We will call all the reconcile functions on instances of this struct.

Reconcile Function

Last but most important, the `root_controller.go` file contains the Reconcile function (Listing 4.12). It "fix" an IotSnap object making it status coincide with the wanted state. It deploys missing object and restore the existing ones to the wanted state, until everything does not match exactly the status we expect according to the provided `spec`.

It starts from retrieving the custom resource to be reconciled, i.e., the one that fired a reconcile request (row 10-21). Kubernetes calls the Reconcile function when one of the watched resources changes. It occurs when a custom resource of type IotSnapRoot is deployed, deleted or updated, or when one of our deployed or conquered objects changes, i.e., it is deployed, reach a state (active, error, and so on) or is modified (e.g., someone accidentally changes its labels as well as a Pod crashes).

At this point it reconciles all the components one by one. In this example we show AgentKeeper (rows 25-65), but in the file we find a similar code for each component. If some resource requires some actions, the operator's controller requires the needed changes and requeues a new reconcile request. If no object requires changes we reach the end of this function in which we return no error and no reconcile. Kubernetes will stop to call this function until a new change occurs.

```
1  // Reconcile reads that state of the cluster for a IotSnapRoot object
       and makes changes based on the state read
2  // and what is in the IotSnapRoot.Spec
3  // The Controller will requeue the Request to be processed again if
       the returned error is non-nil or
4  // Result.Requeue is true, otherwise upon completion it will remove
       the work from the queue.
```

```
5  func (r *ReconcileIotSnapRoot) Reconcile(request reconcile.Request) (
       reconcile.Result, error) {
6      reqLogger := log.WithValues("Request.Namespace", request.
           Namespace, "Request.Name", request.Name)
7      reqLogger.Info("Reconciling IotSnapRoot")
8
9      // Fetch the IotSnapRoot instance
10     instance := &iotsnapv1.IotSnapRoot{}
11     err := r.client.Get(context.TODO(), request.NamespacedName,
           instance)
12     if err != nil {
13         if errors.IsNotFound(err) {
14             // Request object not found, could have been deleted
                   after reconcile request.
15             // Owned objects are automatically garbage collected. For
                    additional cleanup logic use finalizers.
16             // Return and don't requeue
17             return reconcile.Result{}, nil
18         }
19         // Error reading the object - requeue the request.
20         return reconcile.Result{}, err
21     }
22
23     var result *reconcile.Result
24
25     // == Agentkeeper ==========
26     result, err = r.await(mongo)
27     if result != nil {
28         return *result, err
29     }
30
31     result, err = r.await(cloudadmin)
32     if result != nil {
33         return *result, err
34     }
35
36     agentkeeper := getAgentkeeper(instance, 0)
```

```
37
38      for index := range instance.Spec.Agentkeeper {
39          c := getAgentkeeper(instance, index)
40
41          result, err = r.apply(request, instance, r.deployment(c))
42          if result != nil {
43              return *result, err
44          }
45
46          result, err = r.apply(request, instance, r.service(c))
47          if result != nil {
48              return *result, err
49          }
50
51          result, err = r.apply(request, instance, r.route(c))
52          if result != nil {
53              return *result, err
54          }
55
56          result, err = r.handleChanges(c)
57          if result != nil {
58              return *result, err
59          }
60
61          result, err = r.restoreConfigmap(c)
62          if result != nil {
63              return *result, err
64          }
65      }
66
67      // == Finish ==========
68      // Everything went fine, don't requeue
69      return reconcile.Result{}, nil
70  }
```

Listing 4.12: The `ReconcileIotSnapRoot` function in `root_controller.go`. This is the core function of the controller.

We will now take a look to the component reconciliation code. In example in Listing 4.12 we will find the AgentKeeper reconciliation between the AgentKeeper and Finish comments (rows 25-65).

AgentKeeper depends on MongoDB and CloudAdmin. So we will first await the mongo and cloudadmin services (respectively rows 26 and 31). If they are not ready, a reconcile function is requeued until they are up and running.

Then, we will declare an agentkeeper variable (row 36). This has two purposes. The first one is to call the await function on this component: CloudAdmin GUI depends on AgentKeeper. The second is that the `getAgentkeeper` function will create a default `ComponentSpec` object in the AgentKeeper `spec` list if it is empty. This will allow us to iterate on them even if no instances are provided via custom resource `spec`.

Inside the for loop at row 38 we reconcile each AgentKeeper instance. First, we retrieve the component information by calling the `getAgentkeeper` function (row 39), that returns a Component object. There is a similar function for each component. We will see an example object later. Then, from the component object we obtain all the Kubernetes object we need i.e., Secret (only for DBs), Deployment (row 41), Service (row 46), and Route (row 51). We pass them to the `apply` function that takes care of deploying the object if it do not exists yet. Finally, we call `handleChanges` (row 56) and `restoreConfigmap` (row 61). This are recovery functions. We talk about them in Section 4.4.

Component

```
1 // newReconciler returns a new reconcile.Reconciler
2 func newReconciler(mgr manager.Manager) reconcile.Reconciler {
3     return &ReconcileIotSnapRoot{client: mgr.GetClient(), scheme: mgr
          .GetScheme()}
4 }
5
6 // add adds a new Controller to mgr with r as the reconcile.
      Reconciler
7 func add(mgr manager.Manager, r reconcile.Reconciler) error {
8     // Create a new controller
```

```
9       c, err := controller.New("iotsnaproot-controller", mgr,
            controller.Options{Reconciler: r})
10      if err != nil {
11          return err
12      }
```

Listing 4.13: The `Component` struct in `component.go`. It contains all the information about a component.

A component object contains all the information needed to assemble the component's Kubernetes objects, i.e., the Secret, the Deployment, the Service, and the Route.

As we can see in Listing 4.13, it contains:

- the component's `name`;
- the `index` of the component in the `spec` list;
- a pointer to the IotSnapRoot `instance` we are reconciling;
- a pointer to the component's `spec` and `status`, in which are stored respectively the component's specifications and the ConfigMap data;
- the `image` for the main container;
- the `port` for the Pod and the Service;
- a map for the `secrets` key and values;
- the `podSpec` describing the Pod we want in the deployment.

Component objects are instantiated by the component getters, i.e., `getPostgres`, `getMongo`, and so on. Listing 4.14 shows the CloudStorage component, as example.

```
1  func getCloudstorage(instance *iotsnapv1.IotSnapRoot, index int) *
       Component {
2       component := &Component{
3           index:    index,
4           instance: instance,
5
6           name:   "cloudstorage",
```

```
7           spec:   getSpec(&instance.Spec.Cloudstorage, index),
8           status: getStatus(&instance.Status.Cloudstorage, index),
9           image:  "quay.extrasys.it/red-eriot/cloudstorage:4.2.0-
                operator-test",
10          port:   8080,
11      }
12
13      component.podSpec = cloudstorageDeployment(component)
14
15      return component
16  }
```

Listing 4.14: The `getCloudstorage` function inside the homonym file. It returns the `Component` struct we presented in Listing 4.13 for the CloudStorage component.

The `getCloudstorage` function return a Component object representing CloudStorage. Since not all the information in the Component struct are mandatory, it sets only those that are needed for the CloudStorage deployment. `index`, `instance`, `name`, `spec`, `status`, and `image` are mandatory. We added the `port` since it exposes a service.

`podSpec` is mandatory too, and is obtained from the `cloudstorageDeployment` function (Listing 4.15). As for the `getCloudstorage`, this function is present on each component too: We will find a `postgresDeployment` function, a `mongoDeployment` function, and so on.

```
1  func cloudstorageDeployment(c *Component) *corev1.PodSpec {
2      // -- Volumes -----
3      configmapVolume := c.configmapVolume()
4
5      // -- Main container -----
6      mainContainer := c.mainContainer()
7      mainContainer.VolumeMounts = []corev1.VolumeMount{
8          newVolumeMount(configmapVolume, "/deployments/config", ""),
9      }
10
11     // -- Deployment -----
12     return &corev1.PodSpec{
13         RestartPolicy: corev1.RestartPolicyAlways,
```

```
14          Containers:     []corev1.Container{mainContainer},
15          Volumes: []corev1.Volume{
16              configmapVolume,
17          },
18      }
19  }
```

Listing 4.15: The `cloudstorageDeployment` *function inside the* `cloudstorage.go` *file. It returns the* `PodSpec` *object that is assigned (row 13 of Listing 4.14) to the homonym field of the* `Component` *struct in Listing 4.13.*

The deployment function (Listing 4.15) setup volumes and containers and prepare the PodSpec. It is component-specific, even if all the deployment functions are similar. Among the volumes we can find the configmapVolume (3) as in this example, that mounts a ConfigMap, a persistentDataVolume that mounts the PVC, and an installationDataVolume that is an *EmptyDir* volume in which init-containers copy installation data files from the installation data image.

The `mainContainer` function (Listing 4.16) prepares an application container starting from the information of the Component object (image, port, etc.). Then the container can be personalized with component-specific information. In this example we add the *VolumeMount*, but in other components we override the default command, we add arguments, we add environment variables from secrets, and so on. The `mainContainer` function ease the deployment assembling by aggregating the common and repetitive parts. This function is inside the `component.go` file, together with other components utilities and the Component struct itself.

```
1  func (c *Component) mainContainer() corev1.Container {
2      return corev1.Container{
3          Name:            c.name,
4          Image:           c.image,
5          ImagePullPolicy: corev1.PullAlways,
6          Env:             c.getEnv(),
7          Ports: []corev1.ContainerPort{{
8              Name:          c.name,
9              ContainerPort: c.port,
```

```
10          }},
11      }
12  }
```

Listing 4.16: The `mainContainer` function in `component.go`. It assembles the main container for the application. The main container is then inserted in the component deployment (row 6 of the Listing 4.15).

`c.getEnv()` (Listing 4.17) is one of the utilities, while the other properties comes from the component object assembled in `getCloudstorage`. It assembles all the environment variables from the global (row 3) and component-specific (rows 4-7) env, performing an override, and then it transform the map in a Kubernetes object (row 9-15). In a similar way we have functions to build the labels, image names, volume mounts, ConfigMaps and so on. All the utility functions in this file are used by the components file, or by the `objects.go` file that provide Kubernetes objects to be deployed given a Component object.

```
1  func (c *Component) getEnv() []corev1.EnvVar {
2      // start from global env
3      envMap := c.instance.Spec.Env
4      // add (or override with) service-specific env
5      for name, value := range c.spec.Env {
6          envMap[name] = value
7      }
8      // generate the env list
9      env := []corev1.EnvVar{}
10     for name, value := range envMap {
11         env = append(env, corev1.EnvVar{
12             Name:  name,
13             Value: value,
14         })
15     }
16     return env
17 }
```

Listing 4.17: The `getEnv` function in `component.go`. It prepares and merges the global and component-specific environment variables that are mounted to the main container.

Objects

The `objects.go` file contains utilities to generate Kubernetes objects for a component. They are Secrets (taken as example in Listing 4.18), Deployments, Services, and Routes. Each object has the corresponding function that has as parameter the component.

```
1  // == Service ==========
2  func (r *ReconcileIotSnapRoot) service(c *Component) *corev1.Service
       {
3
4       componentLabels := c.getLabels()
5       componentLabels["component"] = c.name
6
7       service := &corev1.Service{
8           ObjectMeta: metav1.ObjectMeta{
9               Name:      c.getName(),
10              Namespace: c.instance.Namespace,
11              Labels:    c.getLabels(),
12          },
13          Spec: corev1.ServiceSpec{
14              Selector: componentLabels,
15              Ports: []corev1.ServicePort{{
16                  Name:       c.name,
17                  Port:       c.port,
18                  TargetPort: intstr.Parse(c.name),
19              }},
20              Type: corev1.ServiceTypeClusterIP,
21          },
22      }
```

Listing 4.18: The `service` function in `objects.go`. Given a component, it returns the service to be deployed for that component.

First, it gets the component labels (row 4). The `getLabels` is pretty similar to the `getEnv` function discussed in Listing 4.17. Then it adds a new label specific for this component (row 5). It is used to retrieve the Service from the Route. In the same way, the Service uses the `componentLabels` to find this component's Pods (row 14). Then the function return a new Service object, filling it up with the component information (rows

7-22). The `getName` function (row 8) returns the component name (`c.name` at row 16) prefixed by the custom resource instance name that makes it unique. For example, if we deploy an IotSnapRoot object called "my-root", `getName` will return "my-root-postgres" for the PostgreSQL component.

Apply Function

The `apply` function (Listing 4.19) deploys a given object if it do not exist. `getEmptyFound` (row 7) returns a new empty runtime object instance of the same type of the object we want to deploy. Then, we look for an existing object (rows 9-13). If it does not exists we create a new one (rows 16-29, in particular `r.client.Create` at row 19). Since multiple reconcile request can be fired within a short time, while the Kubernetes APIs may take some time to apply the requested changes, we handle *already exists* errors (rows 20-27). It is possible that we try to deploy the same object twice (or more) from two different reconcile requests: The second one will fail, but is fine since the object is deployed as we wanted, so we catch the error. If everything goes fine we return `nil, nil`, that means no error and no reconcile (row 29 after the deployment and row 36 if the object already exists).

```
1  func (r *ReconcileIotSnapRoot) apply(
2      request reconcile.Request,
3      instance *iotsnapv1.IotSnapRoot,
4      obj runtime.Object,
5  ) (*reconcile.Result, error) {
6
7      found := getEmptyFound(obj)
8
9      // See if obj already exists
10     err := r.client.Get(context.TODO(), types.NamespacedName{
11         Name:      getName(obj),
12         Namespace: instance.Namespace,
13     }, found)
14
15     // If not exists create it
16     if err != nil && errors.IsNotFound(err) {
```

```
17          // Create the obj
18          log.Info("Creating a new obj", "name", getName(obj), "kind",
                getKind(obj))
19          err = r.client.Create(context.TODO(), obj)
20          if err != nil && errors.IsAlreadyExists(err) {
21              // already exist
22              log.Info("Object already exists", "name", getName(obj), "
                    kind", getKind(obj))
23          } else if err != nil {
24              // cannot create the object
25              log.Error(err, "Failed to create new obj", "name",
                    getName(obj), "kind", getKind(obj))
26              return &reconcile.Result{}, err
27          }
28          // obj deployment was successful
29          return nil, nil
30      } else if err != nil {
31          // Error that isn't due to the obj not existing
32          log.Error(err, "Failed to get obj", "name", getName(obj), "
                kind", getKind(obj))
33          return &reconcile.Result{}, err
34      }
35      // The obj exists, okay
36      return nil, nil
37  }
```

Listing 4.19: The `apply` *function in* `reconcile.go`*. It creates components if they not exist.*

Similarly to the apply function you can find in this file an `update` function, used by the `handleChanges` and `restoreConfigmap` functions described in Sections 4.4.1 and 4.4.2, and a `delete` function used for backup and monitoring deactivation (Sections 4.5 and 4.6).

4.4 Lifecycle Management and Recovery

For what concerns lifecycle, the operator's reconcile function is not limited to deploying the components, but also to making sure that they keep running correctly, providing for their restoration. If a Deployment, a Service or a Route are mistakenly deleted, they are recreated automatically. Then, the Deployment makes sure there is a ReplicaSet, or recreates it, and the ReplicaSet in turn is responsible for the Pods and the number of replicas.

In addition, we have developed some custom functions aimed to restoring the number of replicas, the environment variables in the containers and the initial value of the ConfigMaps. If someone tries to change one of these values, the previous one is immediately restored. To update the values it is necessary to rollout a new version of the custom resource. The operator detects the new specifications and proceeds to update the components involved in order to minimize discontinuity and by using verified steps to reduce the possibility of error.

4.4.1 Component Recovery

The `handleChanges` function

We run the `handleChanges` function (Listing 4.20) across each component we have deployed every time the reconcile function is called, i.e., every time there is a change in one of the operator's components or in the custom resource.

```
1 func (r *ReconcileIotSnapRoot) handleChanges(c *Component) (*
      reconcile.Result, error) {
2     // Get deployment
3     found := &appsv1.Deployment{}
4     err := r.client.Get(context.TODO(), types.NamespacedName{
5         Name:      c.getName(),
6         Namespace: c.instance.Namespace,
7     }, found)
8     if err != nil {
9         // The deployment may not have been created yet, so requeue
```

```
10          log.Info("Deployment not found, requeued", "name", c.getName
                ())
11          return &reconcile.Result{RequeueAfter: 5 * time.Second}, nil
12      }
13
14      // Handle replica changes
15
16      // Handle env changes
17
18      // Handle sidecar changes
19
20      return nil, nil
21  }
```

Listing 4.20: The `handleChanges` *function in* `reconcile.go`*. It restore the actual status with the wanted components status.*

This function retrieves the actual deployment (rows 2-7), i.e., the one that is running, and checks that the replica number (Paragraph *Handle Replica Changes*) and the container environment variables (Paragraph *Handle Environment Changes*) correspond to the wanted specifications. Moreover, it checks for the number of containers inside the Pod (see Paragraph *Handle Sidecar Changes*). If the actual number is different from the number of container in the deployment generated from the custom resource specifications it means that a sidecar container has been attached or detached. In this case the operator proceeds with the rollout of an updated deployment with (or without) the affected sidecar.

Handle Replica Changes

This section of the `handleChange` function checks for the actual replica number, and if it is different from the `spec` it updates it. The replica number can differ if someone changes the Deployment or the ReplicaSet, if a Pod crashes or is deleted, or if the replica number of that component is updated in the custom resource. Except for the latter case, that is an explicit and voluntary update, the others are considered unwanted behaviors. In any case, the number of desired replicas is calculated from the custom resource. If the number of replicas differs (row 2 of the Listing 4.21), we update the number of replicas in

the Deployment currently running (row 5) and we update it with the reconcile's `update` function (row 6) that we will see later.

```
1       // Handle replica changes
2       if *found.Spec.Replicas != *c.getReplicas() {
3           log.Info("Update replicas.", "name", found.Name,
4               "actual-replicas", found.Spec.Replicas, "wanted-replicas
                    ", c.getReplicas())
5           found.Spec.Replicas = c.getReplicas()
6           return r.update(found)
7       }
```

Listing 4.21: Part of the ***handleChanges*** *function which handles the changes of the replicas number.*

Handle Environment Changes

This code uses the same approach of the handle replica section, but against the environment variables. Since we may have more than one container in the deployment, this code scans all the containers to find the main one (rows 3-5 in Listing 4.22). When it has found it, it checks that the environment variables are those desired for the component (row 15). If they differ, it updates the deployment (row 18-19).

Even if the ordering is indifferent, the environment variables are represented in Go as an array. Thus, it is necessary to sort the two arrays (rows 8 and 11) before comparing them. Otherwise, two arrays containing the same variables in different order would be different, while actually the corresponding environments are not. Instead, they may differ because someone has manually modified the env of the container, because the application inside the container has incorrectly changed the value of an environment variable, or because the custom resource has been updated. The latter action is voluntary and explicit and causes an update of the environment variables to align them with the new specification, rather than restoring them.

```
1       // Handle env changes
2       specEnv := c.getEnv()
3       for i, container := range found.Spec.Template.Spec.Containers {
```

```
4           // pick the right container
5           if container.Name == c.name {
6               containerEnv := container.Env
7               // Sort the two environment, since the env variables
                    order may differ
8               sort.Slice(specEnv, func(i, j int) bool {
9                   return specEnv[i].Name < specEnv[j].Name
10              })
11              sort.Slice(containerEnv, func(i, j int) bool {
12                  return containerEnv[i].Name < containerEnv[j].Name
13              })
14              // compare the env
15              if !reflect.DeepEqual(containerEnv, specEnv) {
16                  log.Info("Update Env.", "name", found.Name,
17                      "actual", containerEnv, "wanted", specEnv)
18                  found.Spec.Template.Spec.Containers[i].Env = specEnv
19                  return r.update(found)
20              }
21          }
22      }
```

Listing 4.22: Part of the `handleChanges` *function which handles the changes of the main container's environment variables.*

Handle Sidecar Changes

The last part of the `handleChange` function involves the sidecar. It has been added for the backup. As we will see in Section 4.5, the backup is implemented as a sidecar that reads the data directory mounted by the main container and pushes its encrypted content to an S3 bucket. The backup is not an always active feature, it can be activated or deactivated, even at runtime, depending on the plan purchased by the customer. For this reason, we must be able to add and remove sidecars from Pods. The implementation has been done in order to keep open the addition and removal of sidecars for all Pods, to allow in the future to add sidecars for other purposes. Sidecars are very useful for performing some background tasks, such as backup. Being inside the same Pod, they share the volume mounts and refers to "localhost" as the same address, facilitating communication even

on ports not exposed outside the Pod. For this reason, the possibility of further sidecars being added in the future is not that odd.

Unfortunately, it is not possible to add or remove sidecars from a Pod after it has been deployed. We eliminate the old Deployment (row 9) and let the deployment functions restore it in its updated version, by asking for a new reconciliation (row 13). To see if the sidecar has been added or removed, simply compare the number of actual (row 2) and desired (row 3) containers in the Pod: If they are different (row 4), it means that we have to update the Deployment.

```
1      // Handle sidecar changes
2      actualSidecarNum := len(found.Spec.Template.Spec.Containers)
3      wantedSidecarNum := len(c.podSpec.Containers)
4      if actualSidecarNum != wantedSidecarNum {
5          log.Info("Update sidecars.", "name", found.Name,
6              "actual", actualSidecarNum, "wanted", wantedSidecarNum)
7          // delete the deployment and requeue:
8          // a new deployment with the right sidecars number will occur
9          result, err := r.delete(found)
10         if result != nil {
11             return result, err
12         }
13         return &reconcile.Result{}, nil // requeue
14     }
```

Listing 4.23: Part of the ***handleChanges*** *function which handles the changes of the number of containers in the Pod, i.i, the inijection or removal of a sidecar.*

4.4.2 ConfigMap Preservation

ConfigMaps are not created by the operator but by the automation pipeline. This happens because not all deployment contexts require the use of this operator. So it was necessary to have a configuration management system compatible with all scenarios. Since there is already a valid automation we decided to keep that. However, once the ConfigMaps have been deployed, we want the operator to manage them. In particular, we want the operator to preserve the ConfigMaps data as it was at the time of deployment. Configurations are

not expected to change after deployment, so any changes are unwanted. To do this, the operator conquers the ConfigMaps that we passed to it and saves the data field in the internal state.

If the ConfigMap changes, Kubernetes fires a reconcile request. The content of the ConfigMap is compared with the one saved in the state and if they are different it is promptly restored. If the alteration of the configurations has caused the application to crash, the other recovery functions will restore correct operation.

```
1  func (r *ReconcileIotSnapRoot) restoreConfigmap(c *Component) (*
       reconcile.Result, error) {
2       // Get existing congigmap
3       found := &corev1.ConfigMap{}
4       err := r.client.Get(context.TODO(), types.NamespacedName{
5           Name:      c.getConfigmapName(),
6           Namespace: c.instance.Namespace,
7       }, found)
8       if err != nil {
9           // The configmap doesn't exist, print error and requeue
10          log.Error(err, "Configmap not found", "name", c.getName())
11          return &reconcile.Result{RequeueAfter: 5 * time.Second}, err
12      }
13
14      if c.status.ConfigMapData == nil {
15          // if there is no configmap data in status, dump the current
                configmap data in the status
16          log.Info("Saved ConfigMap data into component Status.", "name
                ", found.Name)
17          mapData := map[string]string{}
18          for k, v := range found.Data {
19              mapData[k] = v
20          }
21          c.status.ConfigMapData = &mapData
22          err = r.client.Status().Update(context.TODO(), c.instance)
23          if err != nil {
24              log.Error(err, "Unable to update the status", "name", c.
                    getName())
```

```
25              return nil, err
26          }
27
28          // set the owner references
29          log.Info("Set OwnerReferences to object", "name", found.Name)
30          controllerutil.SetControllerReference(c.instance, found, r.
                scheme)
31          return r.update(found)
32
33      } else if !reflect.DeepEqual(found.Data, *c.status.ConfigMapData)
            {
34          // otherwise, check if the current configmap reflect the
                status one
35          log.Info("Update Configmap.", "name", found.Name,
36              "actual", found, "wanted", c.status.ConfigMapData)
37          found.Data = *c.status.ConfigMapData
38          return r.update(found)
39      }
40
41      return nil, nil
42  }
```

Listing 4.24: The `restoreConfigmap` function in `reconcile.go`. It saves a copy of the configurations and restore them in case of changes.

In Listing 4.24, rows 1-12 are concerned with getting the ConfigMap object. If there is no data backup (row 14), rows 15-26 make a copy of it and set the OwnerReference on the ConfigMap, adding it to the resources belonging to the operator so that it is notified of any changes. On the other hand, if a data backup is already present, it is compared with the current data and if different it is restored (rows 33-39).

4.4.3 The `clearStatus` Function

As for the other components, we copy into the state the configurations of the backup and monitoring components too, in order to restore their ConfigMap in case of alteration. Anyway, if the component is deactivated we want to be able to change configuration later, as it will be reactivated. If we do not clean the state of their components, upon

activation the original ConfigMap is restored, overwriting the new one. The `clearStatus` function (Listing 4.25) does just that. The operator invokes it when the component is deactivated, so that at the next reactivation it is possible to specify a different ConfigMap. The component status is replaced with an empty status (row 3) and then we update the global status (row 4).

```
1  func (r *ReconcileIotSnapRoot) clearStatus(c *Component) (*reconcile.
       Result, error) {
2       log.Info("Clearing status", "name", c.getName())
3       *c.status = iotsnapv1.ComponentStatus{}
4       err := r.client.Status().Update(context.TODO(), c.instance)
5       if err != nil {
6           log.Error(err, "Unable to clear the status", "name", c.
                getName())
7           return nil, err
8       }
9       return nil, nil
10 }
```

Listing 4.25: The `clearStatus` function in `reconcile.go`. It saves a copy of the configurations and restore them in case of changes.

4.5 Backup

In the third level we also find the backup. We decided to develop backup only for one component, MinIO, which provides a typical use case. AgentKeeper persistent storage contains temporary files and it uses external storage only because the space on the Pods is limited for security and performance reasons, therefore this component does not require backup. For databases the backup will be implemented with replicas and distributed databases with master-slave technique. This will provide, in addition to backup, also redundancy and high availability. The other components do not use persistent storage, thus do not require backup.

MinIO is an ObjectStorage compatible with S3 and is intended for saving files via APIs. The files are not encrypted, compressed or altered. S3 organizes files into buckets,

and in MinIO buckets are implemented as folders. The original files passed through APIs are saved inside the directory corresponding to the bucket. It is sufficient to make a copy of the entire data directory to back up the MinIO data.

The data directory corresponds to the path `/data`, on which we have mounted a PVC for persistent storage. We cannot make assumptions about the type of storage used for PVC. Obviously the simplest case is a *ReadWriteMany* type storage, which can be read by multiple Pods simultaneously. In most cases, however, the storage will be *ReadWriteOnce*, usually less expensive and more efficient. At Extra we did tests with both the ReadWriteOnce and the ReadWriteMany, but probably the first one will be the definitive one. In order not to constrain the operator too much, it is good to assume the worst case, that is ReadWriteOnce.

ReadWriteOnce type storage can only be mounted on one Pod at a time. The possibility of detaching the PVC from the Pod to back it up is excluded, because it would cause too high discontinuity. Therefore, it will not be possible to backup with an external Pod.

With these constraints there are two possibilities for backing up PVC. The first one is to launch commands against the Pod or the PVC. OpenShift provides utilities for backing up PVC from the command line. Or it is possible to do an `exec` across the Pod and execute a command to copy the files, reading them from the attached PVC. The second possibility is to use a sidecar.

A sidecar is nothing but a secondary container inside the Pod. A ReadWriteOnce PVC can be mounted to only one Pod at a time, but all the containers inside the Pod can mount it and use it without restrictions. The sidecar allows to mount the PVC both on the main container in which the application runs and on the backup container, because both are in the same Pod. It is a more flexible and more Kubernetes-oriented solution than using the command line inside an external Pod.

When the backup is activated, we inject a sidecar into the Pod, attach the PVC to the container and the backup service will copy the files at regular intervals (Figure 4.7). If the backup is disabled, we simply remove the sidecar.

Unfortunately, it is not possible to change the number of containers in a Pod at runtime, therefore this operation proves a short disservice. The old Pod is arrested, the

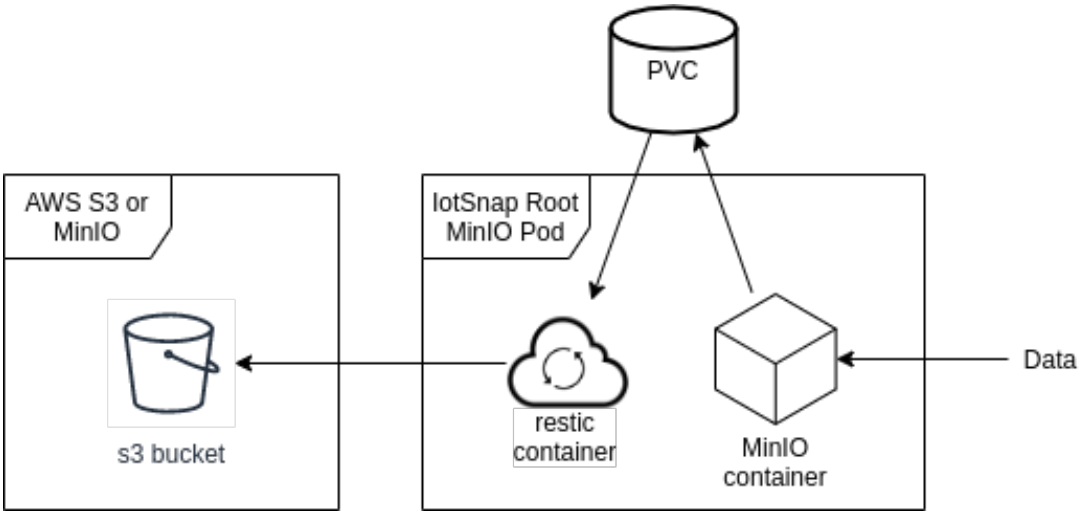


Figure 4.7: Backup service diagram.

PVC is detached and a new Pod is launched with the sidecar, to which the old data PVC is attached. As we have seen in Section 4.4.1, the `handleChanges` function takes care of handling the sidecar attachment and detachment. It deletes the old Deployment, then makes a new Deployment that also contains the sidecar. The same operation is also done to disable the backup: In the new deployment the sidecar is no longer present.

The disservice caused by the Pod recreation is not a problem because it lasts a few seconds and occurs very rarely: Backup is activated or deactivated only if the customer changes its plan, therefore few times over the years. Furthermore, MinIO is a service used only by internal applications because it is used for the distribution of software updates of some components. All microservices that use it handle up to 5 minutes of MinIO downtime.

Once injected the sidecar periodically backs up the data on the PVC to an S3 storage. The backup service running on the sidecar is a modified version of restic. Lobaro, a German startup specialized on full-stack industrial IoT development, added a cronjob to the restic image to allow scheduling the service [44], and with a recent Pull Request we have added a fix for OpenShift [1].

The S3 bucket is implemented with a MinIO Pod. Since this is a proof of concept, MinIO runs inside the same cluster, but the backup sidecars has been implemented in a way that allows to specify the S3 bucket coordinates. We can easly make the sidecar

[1] https://github.com/lobaro/restic-backup-docker/pull/48

point to an Amazon Web Service S3 service or a MinIO instance outside the cluster.

It can also be done at runtime simply by changing the sidecar environment variables. This operation can be done by acting on the custom resource, setting these variables on the service we are backing up.

```
1 RESTIC_REPOSITORY: s3:http://hostname-or-service-name:port/bucketname
2 AWS_ACCESS_KEY_ID: <repository-id>
3 AWS_SECRET_ACCESS_KEY: <repository-secret>
4 BACKUP_CRON: "* * * * *"
```

Listing 4.26: Environment variables to override the target bucket in the backup service.

As for the sidecar, the backup server is deployed and removed dynamically. This operation is inside the `root_controller.go` file, but the code is a bit different from the other components' implementation. Here there is the BackupServer example. We will see a similar behavior also for the Monitoring components we discuss in Section 4.6.

```
1     // == BackupServer ==========
2     if instance.Spec.Backup {
3         backup := getBackupServer(instance, 0)
4
5         result, err = r.apply(request, instance, r.deployment(backup)
              )
6         if result != nil {
7             return *result, err
8         }
9
10        result, err = r.apply(request, instance, r.service(backup))
11        if result != nil {
12            return *result, err
13        }
14
15        result, err = r.handleChanges(backup)
16        if result != nil {
17            return *result, err
18        }
19
```

```
20          result, err = r.restoreConfigmap(backup)
21          if result != nil {
22              return *result, err
23          }
24
25      } else {
26          backup := getBackupServer(instance, 0)
27
28          result, err = r.clearStatus(backup)
29          if result != nil {
30              return *result, err
31          }
32
33          result, err = r.delete(r.deployment(backup))
34          if result != nil {
35              return *result, err
36          }
37
38          result, err = r.delete(r.service(backup))
39          if result != nil {
40              return *result, err
41          }
42      }
```

Listing 4.27: Portion of the `reconcile` function in `root_controller.go` that deploys or deletes the backup server according to its status (active or disabled).

In Listing 4.27 instead of the `for` that iterates on instances as for the other components, we find an if that checks if the backup feature are available or disabled (row 2). If the backup is active the function proceed as for the other components: It applies the deployment (row 5) and the service (row 10), it handles the changes to the runtime objects and the custom resource updates (row 15) and finally it ensure the ConfigMap has not changed, and in case restore its original value (row 20). Instead, if the backup is disabled (row 25) it removes the backup server objects id they exists. If the backup service was active and has been deactivated the corresponding components must be removed. Like when we had to create objects we retrieve the backup component data (row 26). We clear

the component status (row 28) so that in a further activation it can start with a new configuration.

4.6 Monitoring

Like backup, monitoring is an optional feature that can be turned on or off at any time. Its implementation (Figure 4.8), however, is pretty different.

Unlike backup, monitoring is not implemented as a sidecar but as an external component. We chose Telegraf as a tool for collecting application monitoring data. Plugins are installed at Telegraf which interact with the Mongo, PostgreSQL, and Java Spring APIs to collect data from those services. Telegraf sends the data to an InfluxDB database.

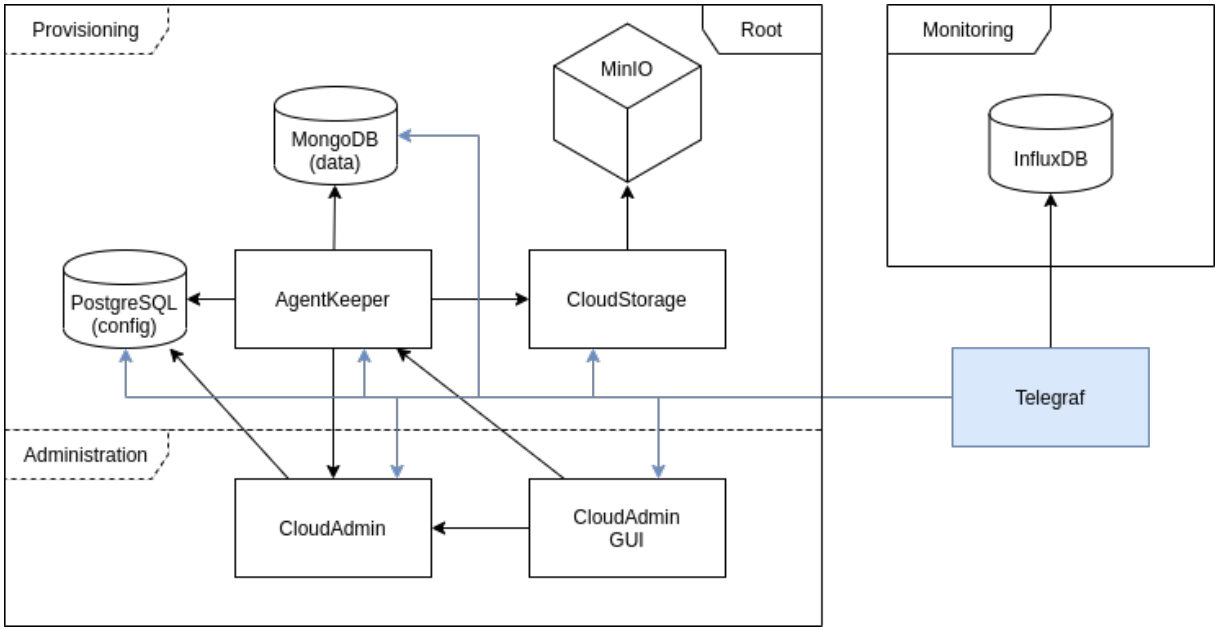


Figure 4.8: Monitoring service architecture diagram.

The database is used by the monitoring area. This area is not always present in IotSnap, therefore monitoring of Root services can be activated or deactivated. Depending on the contract, the monitoring area offers dashboards for data exploration, control panels to set the expected parameters, alarms when certain thresholds are exceeded, data export services to third-party services and interoperability APIs.

Chapter 5

Testing and Validation

In this section we present the operator testing and the formal validation of the operator behavior. Testing an operator should involve both unit tests and end-to-end tests. The operator-sdk provides libraries to mock the Kubernetes client for unit testing and utilities to run end-to-end tests on the cluster. The Root operator tests consist of 1300 rows of code, divided in around 700 rows for end-to-end, 300 for unit tests, and the remaining for common utilities. Unit tests cover the 75% of the code while end-to-end tests almost a hundred percent. The different coverage is due to the fact that some errors and failures cannot be simulated, and because the scaffolding code cannot be reached from the unit tests.

The operator validation is made through Barrel. We formally modeled the Root stack components, their dependencies and offered capabilities, and the transition between their states in TOSCA. Then using the end-to-end tests we compared the actions taken by the operator with the steps described by Barrel. We validated both the deployment and the failure recovery.

5.1 Unit Testing

Unit tests assess the expected outcomes of individual operator components without requiring coordination between components.

Operator unit tests should test multiple scenarios likely to be encountered by a custom operator logic at runtime. Much of the custom logic will involve API server calls via a

client. These API calls can be mocked by using controller-runtime's fake client, intended for unit testing [27].

5.1.1 Test Batteries

In our test batteries we ensure the correctness of the following aspects of the operator:

- the custom resource yaml file parsing, including exhaustive tests on optional values and custom deployments;
- that components labels and environment variables overrides the global ones when specified;
- the correctness of the generated Kubernetes components, according to the specification;
- the resource deployment;
- the scalability, including tests to ensure that all the Pods are accessible by the corresponding service;
- the recovery of Pods after crashes, down scaling or deletion;
- the restore of Secretes, Deployments, Services, and Routes after an involuntary deletion;
- the ConfigMap backup and restore;
- the deletion of all the IotSnapRoot components when the custom resource is deleted.

5.1.2 Testing Environment

The `testEnvironment` structure (Listing 5.1) contains the testing environment `t` (row 2) from the Go testing package, the fake `client` (row 5), a reconcile controller `r` (row 3) that uses the fake client instead of the real one, a reconcile request `req` (row 4), and a `root` custom resource instance (row 6) that we will deploy and reconcile.

```
1 type testEnvironment struct {
2     t      *testing.T
3     r      *controller.ReconcileIotSnapRoot
4     req    reconcile.Request
5     client client.Client
6     root   *iotsnapv1.IotSnapRoot
7 }
```

Listing 5.1: The `testEnvironment` struct in `utils.go`. It contains all the objects needed to unit-test an operator.

To set up the test environment we first need a scheme (Listing 5.2, row 3). As in the operator controller, we create a new scheme, then we add the Routes (row 5) and IotSnapRoot (row 4) resources because the scheme includes only Kubernetes native resources. Once we have a scheme we create a new fake client using the scheme (row 21). Then we add to the client the external resources, i.e., the PVCs and ConfigMaps that the automation pipeline would deploy (retrieved at row 8), and the root custom resource (row 14). Since the client emulates the Kubernetes APIs, we do not need to wait the resource creation as in a real Kubernetes cluster. The client is used to create the Reconcile object (row 24) that contains the client and the scheme needed to reconcile an IotSnapRoot resource. Finally, we prepare a reconcile request (row 27-32) for the operator and we return a `testEnvironment` structure we discussed in Listing 5.1 (rows 34-40).

```
1 func prepareTestEnvironment(t *testing.T) testEnvironment {
2     // Register operator types with the runtime scheme.
3     s := scheme.Scheme
4     s.AddKnownTypes(iotsnapv1.SchemeGroupVersion, &iotsnapv1.
          IotSnapRoot{})
5     s.AddKnownTypes(iotsnapv1.SchemeGroupVersion, &routev1.Route{})
6
7     // External resources
8     externalResources, err := testutils.GetExternalResources("../../
          deploy/resources", namespace)
9     if err != nil {
10         t.Error(err)
11     }
```

```
12
13      // The IotSnapRoot CR
14      root := testutils.GetIotSnapRootMin("iotsnap-root", namespace)
15
16      // Objects to track in the fake client.
17      objs := []runtime.Object{root}
18      objs = append(objs, externalResources...)
19
20      // Create a fake client to mock API calls.
21      client := fake.NewFakeClientWithScheme(s, objs...)
22
23      // Create a ReconcileIotSnapRoot object with the scheme and fake
            client
24      r := controller.NewReconcileIotSnapRoot(client, s)
25
26      // Mock request to simulate Reconcile() being called on an event
            for a watched resource
27      req := reconcile.Request{
28          NamespacedName: types.NamespacedName{
29              Name:      "iotsnap-root",
30              Namespace: namespace,
31          },
32      }
33
34      return testEnvironment{
35          t:      t,
36          r:      r,
37          req:    req,
38          client: client,
39          root:   root,
40      }
41  }
```

Listing 5.2: The ***prepareTestEnvironment*** *function in* ***utils.go****. It prepares the testing environment and return the* ***testEnvironment*** *structure in Listing 5.1.*

5.1.3 Test Example

In most of the unit tests we check for the correctness of the Reconcile function response to certain situations. In Listing 5.3 we present a test that ensures the correct resources deployment when the minimum IotSnapRoot custom resource is deployed i.e., when no specifications are provided and the operator uses the default values.

```
1  func TestMinimumDeployment(t *testing.T) {
2      // Prepare a testing environment
3      te := prepareTestEnvironment(t)
4
5      // Ensure a full deploy
6      // Send a reconcile request and check the result:
7      // we expect postgres, mongo and minio, but
8      // not cloudstorage since it is waiting for minio to be up.
9      // We also expect a requeue since it need to chown the configmap
10     te.expectReconcileRequeueToBe(true)
11     postgres := te.ensureDeployment("iotsnap-root-postgres", 1)
12     mongo := te.ensureDeployment("iotsnap-root-mongo", 1)
13     minio := te.ensureDeployment("iotsnap-root-minio", 1)
14
15     // A new reconcile request should not be requeued
16     te.expectReconcileRequeueToBe(false)
17
18     // Set minio as ready and resend a new reconcile request
19     minio.Status.ReadyReplicas = 1
20     te.client.Update(context.TODO(), minio)
21     te.expectReconcileRequeueToBe(true)
22
23     // Check the result: we expect cloudstorage but not cloudadmin
           since it depends on cloudstorage
24     cloudstorage := te.ensureDeployment("iotsnap-root-cloudstorage",
           1)
25     // Again, a new reconcile request should not be requeued
26     te.expectReconcileRequeueToBe(false)
27
28     // ..
```

```
29
30      // Check the result: we finally expect cloudadmingui
31      te.ensureDeployment("iotsnap-root-cloudadmingui", 1)
32      te.expectReconcileRequeueToBe(false)
33  }
```

Listing 5.3: The `TestMinimumDeployment` function in `root_controller_test.go`. It tests the deployment correctness when no `spec` is passed in the custom resource.

In Listing 5.3, first we prepare the testing environment (row 3) we have presented in the previous Section. Then we send a reconcile request that should return a requeue request (row 10): The operator has found the ConfigMaps of the deployed services and has tried to set its `ownerReference` on them. It has asked for a requeue to verify the correctness of the operation. PostgreSQL, MongoDB, and MinIO components should be deployed because they have no dependency. Instead, because they are not running, the other components that depends on them (e.g., CloudStorage) are `await`ing them and are not deployed. We ensure their deployment (rows 11-13) and then we send a new reconcile request (row 16). This time we expect no requeue: Kubernetes should have set the `ownerReference` on the ConfigMaps. The operator performs their backup and ends its execution. CloudStorage is still missing, but the operator cannot deploy it until components it depends on do not reach a ready state. When they change state Kubernetes will fire a brand new reconcile request, so there is no reason to requeue the current one.

Since we are in an emulated cluster, we can and need to set the ready replicas manually. By setting the MinIO `ReadyReplicas` to 1 (rows 19-20) the number of ready and wanted replicas coincide and the deployment will reach a ready state. Then, by simulating the Kubernetes behavior we fire a new reconcile request (row 21). The MinIO readiness state allows the operator to deploy CloudStorage. Again, the first request is requeued (row 21) because of the ConfigMap, the second one (row 26) is not, since CloudAdmin is waiting for PostgreSQL. We have omitted the entire code, since the test function keeps updating the readiness of components and send reconcile requests until the last component, CloudAdmin GUI, is deployed (rows 31-32).

As we have seen in this function, it is possible to emulate the component behavior. We manipulated the Status to set the Pod replicas to 1. In a similar way, in other

test functions we change simulate crashes, down and up scaling, we remove objects from the client, we simulate component availability and unavailability, and we change labels, environment variables, and ConfigMap data. By firing reconcile request we can observe the operator reaction to a certain event and ensure the correctness of the operator behavior.

5.2 End-To-End Testing

End-to-end tests are essential to ensure that an operator works as intended in real-world scenarios. While unit tests run in a simulated environment with a fake client, end-to-end tests run on a real cluster. This calls cluster reaction times into play and involves real application that may fail or crash due to an incorrect communication and interaction or when not properly configured.

As for the operator, we can run end-to-end tests both from inside or outside the cluster. When we execute end-to-end tests locally, the operator code and tests code is run on the local machine. Instead, when running on a cluster the test code is packaged by the operator-sdk in an image, together with the operator's code. Then the operator is deployed on the cluster and both the operator's and tests calls are made from inside the cluster.

The Operator SDK includes a testing framework to make writing end-to-end tests simpler and quicker by removing boilerplate code and providing common test utilities. The Operator SDK includes the test framework as a library under pkg/test and the end-to-end tests are written as standard go tests [28].

The test framework includes a few components. The most important are Framework and Context. Framework contains all the global variables, such as kubeconfig, kubeclient, scheme, and dynamic client provided via the controller-runtime project. It is initialized by the MainEntry function and can be used anywhere in the tests. Context stores important information for each test, such as the namespace and the cleanup functions. By handling namespace and resource initialization through Context, we are sure that all resources are properly handled and removed after the test finishes.

5.2.1 Test Initialization

```
1  package e2e
2
3  import (
4      "testing"
5      f "github.com/operator-framework/operator-sdk/pkg/test"
6  )
7
8  func TestMain(m *testing.M) {
9      f.MainEntry(m)
10 }
```

Listing 5.4: Content of the `main_test.go` file, that initializes the test framework.

A `main_test.go` (Listing 5.4) file initializes the test framework. Its role is to call the framework `MainEntry` function (row 9) that prepares the Go testing instance for end-to-end testing.

5.2.2 Test Batteries

The operator framework guidelines suggest to organize test in one file per controller. In our case we have a single controller for the Root stack and its tests stands inside `root_test.go`. The test file should have a single test function corresponding to the custom resource name, so `TestIotSnapRoot` (Listing 5.5).

```
1  func TestIotSnapRoot(t *testing.T) {
2      // Add CRD to operator-sdk test framework
3      rootList := &iotsnapv1.IotSnapRootList{}
4      err := framework.AddToFrameworkScheme(apis.AddToScheme, rootList)
5      if err != nil {
6          t.Fatalf("Failed to add custom resource scheme to framework:
               %v", err)
7      }
8
9      // Run tests
10     t.Run("Deploy", func(t *testing.T) {
11         t.Run("MinDeployTest", minDeployTest)
```

```
12          t.Run("FullDeployTest", fullDeployTest)
13      })
14      t.Run("Recovery", func(t *testing.T) {
15          t.Run("RecoveryAfterDelete", recoveryAfterDelete)
16          t.Run("RecoveryAfterScaling", recoveryAfterScaling)
17          t.Run("RestoreConfigMap", restoreConfigMap)
18      })
19      t.Run("Runtime update", func(t *testing.T) {
20          t.Run("Replicas", updateReplicas)
21          t.Run("Env and Labels", updateEnvLabels)
22      })
23  }
```

Listing 5.5: The `TestIotSnapRoot` function in `root_test.go`. This is the main test function, that prepares the cluster and runs the test batteries.

The first step we have to do in the Test function is to register the operator's scheme with the framework's dynamic client. To do this, we pass the custom resource Definition's `AddToScheme` function and its List type object to the framework's `AddToFrameworksScheme` function (row 4). The List, that in our case is called `IotSnapRootList` (row 3), contains the custom resource types and meta data definitions and a list of IotSnapRoot resources that will be populated gradually deploying resources. We add this structure to the framework scheme because the framework needs to ensure that the dynamic client has the REST mappings to query the API server for the custom resource type. The framework will keep polling the API server for the mappings and timeout after 5 seconds, returning an error if the mappings were not discovered in that time. Then, we can run test batteries (rows 10, 14 and 19). Each battery is ran with the function `Run` of the `testing.T` class. In the same way we run tests inside the battery, by just nesting other `t.Run` calls inside the battery testing function (e.g., rows 11-12).

5.2.3 Cluster Preparation

In each test function we need to create a Context for the current test and defer its cleanup function.

```
ctx := framework.NewContext(t)
```

```
defer ctx.Cleanup()
```

Now that there is a Context, the Kubernetes test's resources can be initialized. They are the test namespace, Service Account, RBAC, and operator deployment.

```
err := ctx.InitializeClusterResources(&framework.CleanupOptions{
    TestContext: ctx,
    Timeout: cleanupTimeout,
    RetryInterval: cleanupRetryInterval,
})
```

The `InitializeClusterResources` function uses the custom `Create` function in the framework client to create the resources provided in your namespaced manifest. The custom Create function uses the controller-runtime's client to create resources and then creates a cleanup function that is called by `ctx.Cleanup` which deletes the resource; then it waits for the resource to be fully deleted before returning and run the following test.

Our `prepareTestCluster` function (Listing 5.6) prepares the cluster for a test ran. We call this function at the beginning of each test case. `t.Parallel()` (row 11) instruct Go that tests can be run in parallel. In this case, different namespaces are used for each test, so that there is no overlapping. Then we allocate the context (row 14) and set the cleanup options for resources removal (rows 19-20). Once the cluster resources are initialized we deploy the operator (row 35) and finally (rows 40-47) we return a `TestCluster` (definide in rows 1-7) containing all the objects we need to run tests and interact with the cluster.

```
1  type TestCluster struct {
2      t              *testing.T
3      f              *framework.Framework
4      ctx            *framework.TestCtx
5      namespace      string
6      cleanupOptions *framework.CleanupOptions
7  }
8
9  func prepareTestCluster(t *testing.T) *TestCluster {
10     // Tests can be run in parallel
11     t.Parallel()
```

```
12
13      // Create a test framework
14      ctx := framework.NewTestCtx(t)
15
16      // Setup the cleanup
17      cleanupOptions := framework.CleanupOptions{TestContext: ctx,
            Timeout: cleanupTimeout, RetryInterval: cleanupRetryInterval}
18      err := ctx.InitializeClusterResources(&cleanupOptions)
19      if err != nil {
20          t.Fatalf("failed to initialize cluster resources: %v", err)
21      }
22      t.Log("Cluster resources initialized")
23
24      // Get global framework variables
25      f := framework.Global
26
27      // Get the namespace
28      namespace, err := ctx.GetNamespace()
29      if err != nil {
30          t.Fatal(err)
31      }
32      t.Log("Working on namespace", namespace)
33
34      // Wait for root-operator to be ready (only if not testing
            locally)
35      err = e2eutil.WaitForOperatorDeployment(t, f.KubeClient,
            namespace, "iotsnap-root", 1, deployRetryInterval,
            deployTimeout)
36      if err != nil {
37          t.Fatal(err)
38      }
39
40      // return the test cluster
41      return &TestCluster{
42          t,
43          f,
44          ctx,
```

```
45          namespace,
46          &cleanupOptions,
47      }
48  }
```

Listing 5.6: The `prepareTestCluster` function in `root_test.go`. It performs a cluster setup before running tests.

5.2.4 Test Function

Each test function starts by preparing the cluster and the namespace for the test (row 3 of the Listing 5.7). We defer the context cleanup (row 4), so that the `ctx.Cleanup` function is called at the end of the test function. Then we deploy the external resources we need (row 7), that are ConfigMaps and PVCs, and last we deploy the custom resource.

In the example in Listing 5.7, we deploy the default custom resource (row 10), but in other test we deploy a fully customized custom resource or a specific custom resource configuration according to the tests we need to run on it. Finally we can run our tests. In this simple example we ensure that Secrets (rows 20-21), Deployments (rows 23-29), and Services (rows 31-37) are correctly deployed, calling some utility functions.

```
1  func minDeployTest(t *testing.T) {
2      // Prepare the cluster and schedule the cleanup
3      c := prepareTestCluster(t)
4      defer c.ctx.Cleanup() // can't be done in prepareCluster
5
6      // Deploy the external resources
7      c.deployExternalResources()
8
9      // Deploy root
10     root := testutils.GetIotSnapRootMin("iotsnap-root", c.namespace)
11     if err := c.f.Client.Create(goctx.TODO(), root, c.cleanupOptions)
           ; err != nil {
12         t.Fatal(err)
13     }
14
15     // Expected labels and env
```

```
16      labels := c.getLabels()
17      env := c.getEnv()
18
19      // Verify that everything is deployed correctly
20      c.ensureSecret("iotsnap-root-postgres", labels)
21      c.ensureSecret("iotsnap-root-mongo", labels)
22
23      c.ensureDeployment("iotsnap-root-postgres", labels, nil, 1)
24      c.ensureDeployment("iotsnap-root-mongo", labels, nil, 1)
25      c.ensureDeployment("iotsnap-root-minio", labels, env, 1)
26      c.ensureDeployment("iotsnap-root-cloudstorage", labels, env, 1)
27      c.ensureDeployment("iotsnap-root-cloudadmin", labels, env, 1)
28      c.ensureDeployment("iotsnap-root-agentkeeper", labels, env, 1)
29      c.ensureDeployment("iotsnap-root-cloudadmingui", labels, env, 1)
30
31      c.ensureService("iotsnap-root-postgres", labels)
32      c.ensureService("iotsnap-root-mongo", labels)
33      c.ensureService("iotsnap-root-minio", labels)
34      c.ensureService("iotsnap-root-cloudstorage", labels)
35      c.ensureService("iotsnap-root-cloudadmin", labels)
36      c.ensureService("iotsnap-root-agentkeeper", labels)
37      c.ensureService("iotsnap-root-cloudadmingui", labels)
38  }
```

Listing 5.7: The `minDeployTest` function in `deployment.go`, taken as example to explain a test function structure.

In Listing 5.8 we show one of the utility functions mentioned above, `ensureSecret`. The role of this function is to check if the passed Secret has been correctly deployed on the cluster. End-to-end tests interact with the cluster though a polling function (row 2) that runs any `deployRetryIntervall` second until the condition is not satisfied or the `deployTimeout` is reached. In this latter case the test fails. Inside the `Poll` function we check for the secret existence (row 4) and we ensure the correctness of its labels (row 16). If the secret does not exist we requeue the request (row 9), awaiting for the next polling. If the secret is found, but the labels does not match, an error is returned (row 17) and the test fails.

```
1  func (c *TestCluster) ensureSecret(name string, labels map[string]
       string) {
2      err := wait.Poll(deployRetryInterval, deployTimeout, func() (done
           bool, err error) {
3          // Get the object
4          found, err := c.f.KubeClient.CoreV1().Secrets(c.namespace).
               Get(name, metav1.GetOptions{})
5          if err != nil {
6              // not available, requeue
7              if apierrors.IsNotFound(err) {
8                  c.t.Logf("Waiting for Secret: %s", name)
9                  return false, nil
10             }
11             // error
12             return false, err
13         }
14         // Check the labels
15         if labels != nil {
16             if !reflect.DeepEqual(labels, found.Labels) {
17                 return false, fmt.Errorf("Secret labels doesn't match
                       : %s (ACTUAL: %s - EXPECTED: %s)",
18                     name, found.Labels, labels)
19             }
20             c.t.Logf("Secret labels match: %s", name)
21         }
22         return true, nil
23     })
24     if err != nil {
25         c.t.Fatalf("Error while waiting for Secret: %s - %s", name,
               err)
26     }
27     c.t.Logf("Secret available: %s", name)
28 }
```

Listing 5.8: The `ensureSecret` function in `e2eutils.go`, taken as example to show how the end-to-end testing functions interact with the real cluster to verify the operator behavior.

5.3 End-To-End Deployment Validation

Validation of a Kubernetes Operator can be done in various ways, including the OpenShift certification program. For this prototype we opted for a more formal way, relying on the Management Protocols based on TOSCA and implemented in the open source Barrel tool (Section 2.5).

5.3.1 Component Definition

We defined the IotSnap Root components as TOSCA entities thanks to Winery. Components are represented as Nodes. Each node has a Node type, that defines the node requirements offered capabilities.

Node	Node type	Capabilities	Requirements	Operations
PVC	PVC	Storage		
Postgres	Database	DBEndpoint	PVC	
Mongo	Database	DBEndpoint	PVC	
MinIO	MinIO	Endpoint	PVC	
CloudStorage	CloudStorage	Endpoint	Connection	
CloudAdmin	CloudAdmin	Endpoint	Connection1 Connection2	
AgentKeeper	AgentKeeper	Endpoint	Connection1 Connection2 Connection3	
CloudAdminGui	CloudAdmin	Endpoint	Connection1 Connection2	

Figure 5.1: IotSnap Root components defined in TOSCA and parsed by Barrel.

In our implementation we defined the following node types (Figure 5.1).

- PVC, the simplest component. It has no requirements and offers the Storage capability. It represent the Root components' persistent storage.

- Database, to represent both PostgreSQL and MongoDB. It has a PVC requirement and offer a DB endpoint.
- MinIO, that needs a PVC and offers an Endpoint.
- CloudStorage, CloudAdmin, and AgentKeeper that all offers an Endpoint and require respectively one, two, and three connection each. The connection requirement is satisfied by an endpoint, that means that, for example, AgentKeeper depends on three components.

Starting from this node types we modeled the storage and the seven microservices. We did not model ConfigMaps because a missing CM is equivalent to a misconfiguration, that produces a crash.

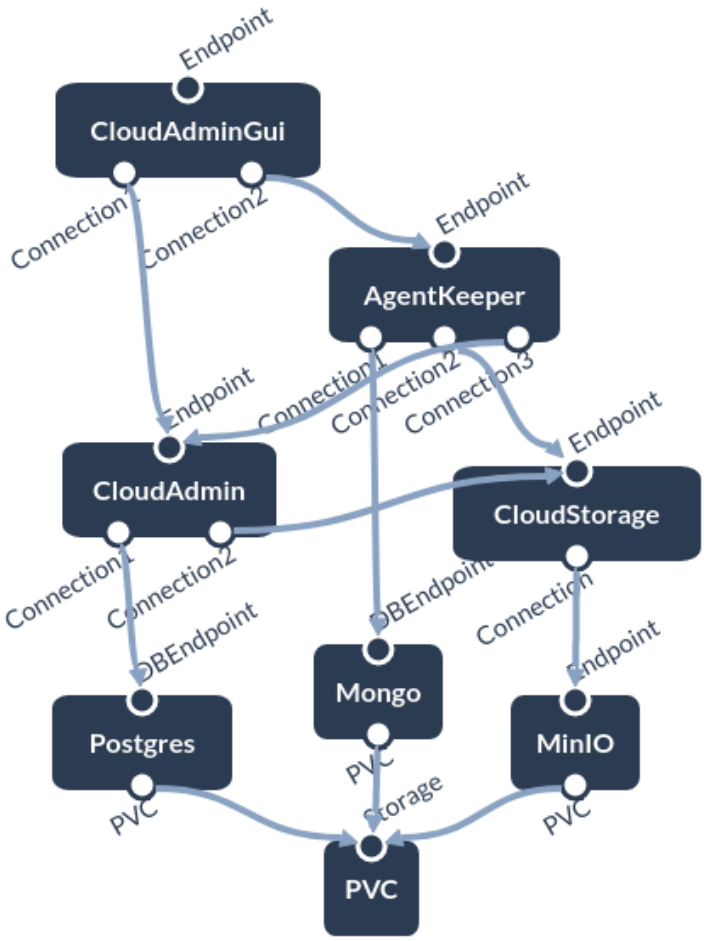


Figure 5.2: IotSnap Root components diagram represented in Barrel.

Figure 5.2 shows the component interaction. PostgreSQL and MongoDB implement the Database node type. Together with MinIO, that implements the MinIO type, they satisfy the PVC requirement with the Storage capability offered by the PVC. CloudStorage, implementing homonym type, satisfies its single connection requirements with MinIO. CloudAdmin has two connection requirements: One is satisfied by the DB Endpoint offered by PostgreSQL and the other by CloudStorage. AgentKeeper connects to MongoDB, CloudAdmin, and CloudStorage. Finally, CloudAdminGUI of type CloudAdmin connects to CloudAdmin and AgentKeeper. Its endpoint capability is not used by any other node, since it is a GUI and the connection is used by humans through a browser.

5.3.2 State Transitions

The PVC node type has only two states i.e., *Unavailable* and *Available* (Figure 5.3). In the first one it does not exist, because either has been delete or not yet created. By creating a PVC, it arrives in the Available state, that offers the Storage capability needed by some other nodes. A delete move the PVC back to the Unavailable state.

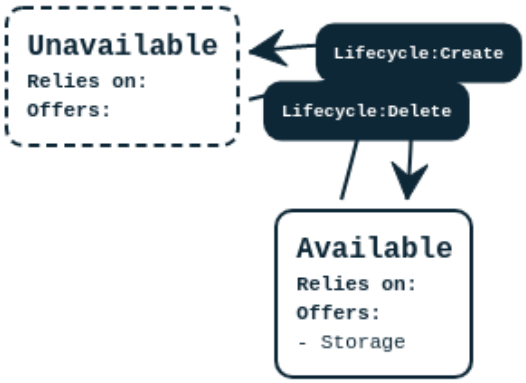


Figure 5.3: PVC state diagram represented in Barrel.

The other node types have three states i.e., *Unavailable*, the starting state in which the component does not exist, *Running*, the normal state, and the special state *Crashed* (Figures 5.4 and 5.5). The *Run* operation moves the component from the Unavailable to the Running state. The operation relies on the requirements needed by the Running state. In the Database node type that we see in Figure 5.4 is the PVC, for other components

are the connections. In the Running state components offer their capability, either DB Endpoint or Endpoint depending on the node type. Like for the PVC, from the Running state we can move back to Unavailable with the *Delete* operation.

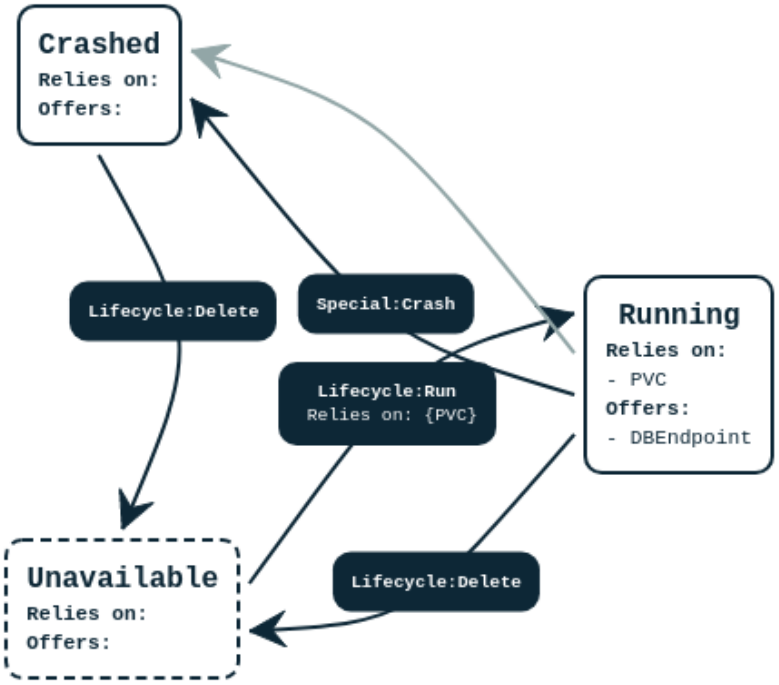


Figure 5.4: State diagram of the Database node type represented in Barrel.

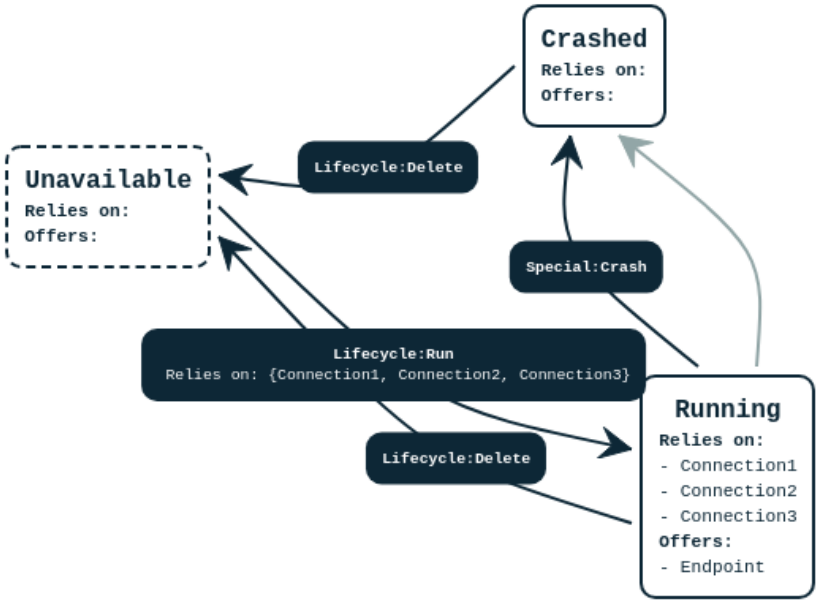


Figure 5.5: State diagram of the AgentKeeper node type represented in Barrel.

The third state, Crashed, is a special state. It is reachable from the Running state. A node can go in the Crashed state for two reasons i.e., for a crash or when a requirement is not satisfied anymore. The crash is simulated through a special *Crash* operation. This operation always leads to a Crashed state. The only way to recover from a crash is to delete the component (i.e., the Pod) and perform again the Run operation. Finally, when a requirement is not satisfied anymore, a fault handler takes care of managing the situation. Since it was a state requirement, the node cannot stay in this state anymore. We can set up many fault handler, according to the requirement that has faulted. Each fault handler lead to a state and for each of this state we can specify the requirements. The node will follow the fault handler that leads to the state with as many met requirements as possible. In our case, independently from the number of requirements lost, the component behaves similar to a crash, so we set up only a handler for the Crashed state. Then the recovery is the same of a crash: Delete and Run again.

5.3.3 Simulator

	State	Offered capabilities	Assumed requirements	Available operations
PVC	Available	Storage		Lifecycle:Delete
Postgres	Running	DBEndpoint	PVC	Lifecycle:Delete Special:Crash
Mongo	Running	DBEndpoint	PVC	Lifecycle:Delete Special:Crash
MinIO	Running	Endpoint	PVC	Lifecycle:Delete Special:Crash
CloudStorage	Running	Endpoint	Connection	Lifecycle:Delete Special:Crash
CloudAdmin	Running	Endpoint	Connection1 Connection2	Lifecycle:Delete Special:Crash
AgentKeeper	Unavailable			Lifecycle:Run
CloudAdminGui	Unavailable			Lifecycle:Run

Figure 5.6: The Barrel simulator. It shows the current state of each node in the simulation, their offered capabilities and assumed requirements. On the right we have the available operations. The orange one cannot be performed for missing requirements. The current state shows the PVC in the Available state and the other components in a Running state, except for AgentKeeper and CloudAdminGUI that are still Unavailable.

The Barrel simulator allows to perform operations on nodes (Section 2.5). Each oper-

ation is a transition through two states. Operations can be ran only if the requirements are satisfied. For example, in Figure 5.6 we can call the Run operation on AgentKeeper (the green button) but not CloudAdminGUI (orange bottom). As we have seen in Section 5.3.2, AgentKeeper requires the CloudAdmin and CloudStorage Endpoint, and the MongoDB DB Endpoint capabilities, represented as three connection requirements. All this requirements are satisfied, as we can see from the *Offered capabilities* column. Instead CloudAdmin GUI requires CloudAdmin and AgentKeeper. The CloudAdmin endpoint is satisfied, but since AgentKeeper is still Unavailable it does not offer any capability.

	State	Offered capabilities	Assumed requirements	Available operations
PVC	Available	Storage		Lifecycle:Delete
Postgres	Running	DBEndpoint	PVC	Lifecycle:Delete Special:Crash
Mongo	Running	DBEndpoint	PVC	Lifecycle:Delete Special:Crash
MinIO	Running	Endpoint	PVC	Lifecycle:Delete Special:Crash
CloudStorage	Running	Endpoint	Connection	Lifecycle:Delete Special:Crash
CloudAdmin	Running	Endpoint	Connection1 Connection2	Lifecycle:Delete Special:Crash
AgentKeeper	Running	Endpoint	Connection1 Connection2 Connection3	Lifecycle:Delete Special:Crash
CloudAdminGui	Unavailable			Lifecycle:Run

Figure 5.7: The Barrel simulator after executing the Run operation on AgentKeeper.

As we can see in Figure 5.7, if we perform the Run operation on AgentKeeper the CloudAdminGUI's Run operation becomes available. AgentKeeper has reached the Running state and is offering it endpoint, that is a CloudAdminGUI requirement. We can see also that AgentKeeper is holding its requirements i.e., Connection1, Connection2, and Connection3 in the *Assumed requirements* column.

To test the fault tolerance we need to remove a requirement. By simulating a crash on CloudAdmin we make it move to the Crashed state through the Crash operation. At this point AgentKeeper lose one of its connections and reaches the Crashed state through the fault handler. Figure 5.8 shows the situation after the crash: CloudAdmin is in the Crashed state and AgentKeeper is still in Running but with a missing requirement for the Connection3. The only action we can do is to handle the missing requirement by clicking on the red button: This will execute the fault handler and led AgentKeeper to

the Crashed state.

	State	Offered capabilities	Assumed requirements	Available operations
PVC	Available	Storage		Lifecycle:Delete
Postgres	Running	DBEndpoint	PVC	Lifecycle:Delete Special:Crash
Mongo	Running	DBEndpoint	PVC	Lifecycle:Delete Special:Crash
MinIO	Running	Endpoint	PVC	Lifecycle:Delete Special:Crash
CloudStorage	Running	Endpoint	Connection	Lifecycle:Delete Special:Crash
CloudAdmin	Crashed			Lifecycle:Delete
AgentKeeper	Running	Endpoint	Connection1 Connection2 Connection3	Lifecycle:Delete Special:Crash
CloudAdminGui	Unavailable			Lifecycle:Run

Figure 5.8: The Barrel simulator after the CloudAdmin crash.

In Figure 5.9 we can see the operations that are available again. Both CloudAdmin and AgentKeeper are crashed and we can only call the Delete operation on them. This will restore their Unavailable state. Then, we can the Run operation on CloudAdmin, and only when it is running its endpoint become available, so that we can call Run also on AgentKeeper. This action will restore the situation as in 5.6.

	State	Offered capabilities	Assumed requirements	Available operations
PVC	Available	Storage		Lifecycle:Delete
Postgres	Running	DBEndpoint	PVC	Lifecycle:Delete Special:Crash
Mongo	Running	DBEndpoint	PVC	Lifecycle:Delete Special:Crash
MinIO	Running	Endpoint	PVC	Lifecycle:Delete Special:Crash
CloudStorage	Running	Endpoint	Connection	Lifecycle:Delete Special:Crash
CloudAdmin	Crashed			Lifecycle:Delete
AgentKeeper	Crashed			Lifecycle:Delete
CloudAdminGui	Unavailable			Lifecycle:Run

Figure 5.9: The Barrel simulator after the fault handler execution.

This was just a simple example, but with Barrel we can test more complicate situations. For example, if we remove the PVC PostgreSQL, MongoDB, and MinIO will lose their PVC requirements. By calling the fault handling operations we will move them to the

Crashed state in which they do not offer their capabilities. This will cause the other components to lose their requirement, that means other fault handling calls. When the cascade fault is totally handled we can remove all the components and restart with a new deployment, executing one by one the Run operation.

At each step Barrel shows the possible actions we can perform, the capability that components are offering and the requirements they are holding. It can be used to see which step the operator can perform in a certain situation or what we expect to succeed in case of fault or crash on a certain component.

5.3.4 Planner

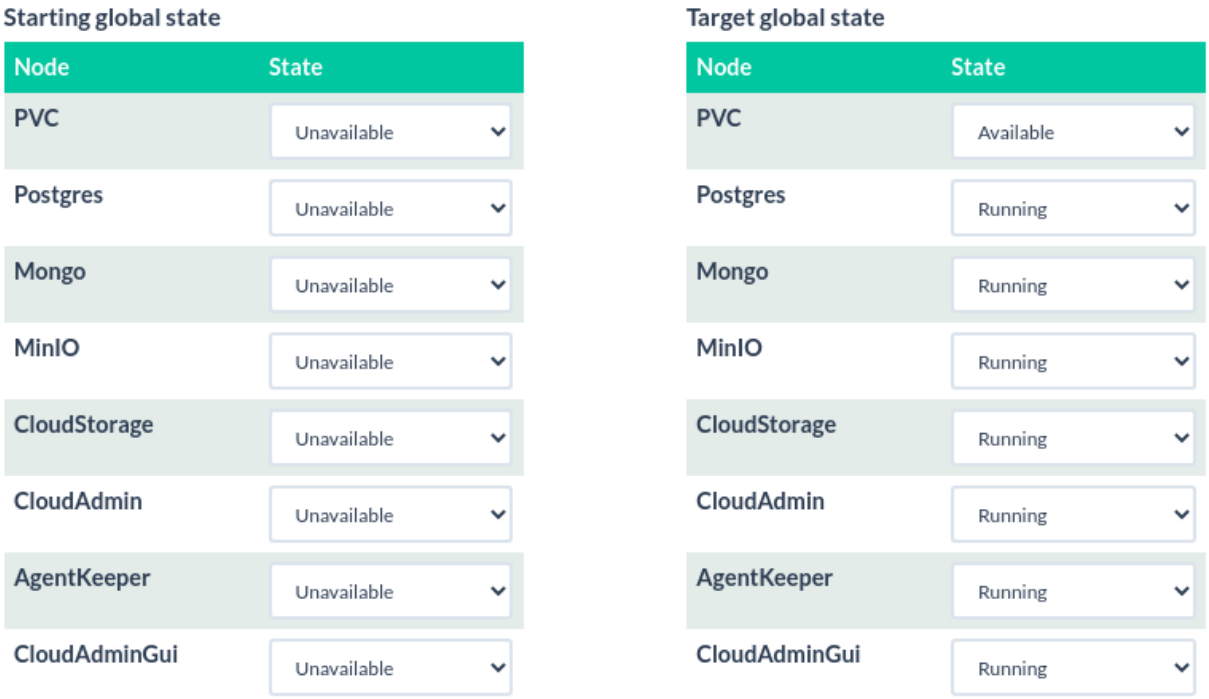


Figure 5.10: The setup panel of the Barrel planner.

Barrel offers also a planner, which we also used to validate our operator. On the planner we can specify the starting global state, i.e., the initial state of each component, and the target global state, in which we specify which state we want each component to reach. In the example in Figure 5.10 we want to perform a full deployment, i.e., we want to reach the Available and Running state in all the components starting from the initial state Unavailable on each.

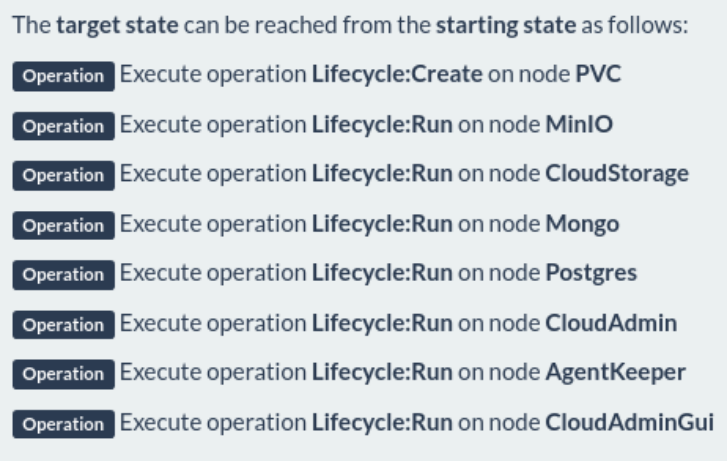


Figure 5.11: The Barrel planner showing the steps for a full deployment.

Given the starting and target states, the planner shows if it is possible to reach the target state and which steps we have to perform. Figure 5.11 shows the deployment operations. Similarly we can obtain the steps for a recovery given a certain fault, and verify that in our model there is a path from each fault to the deployment state.

5.3.5 Validation Process

We validated the operator using Barrel in conjunction with the tests. Unfortunately Barrel offers only a web UI, it does not integrate with tests, it has no APIs nor other ways to programmatically interact with it. For that reason the validation must be done manually, and must be repeated every time we change the operator's actions or behavior. Through the Barrel UI we identified all the possible failure scenarios and we wrote tests for each of them. Then, we ran the tests one by one and we compared the operator's actions with the Barrel planner steps. We observed the operator behavior by inspecting its logs, using a debugger, and by observing the Kubernetes object mutation on the cluster.

Sometimes there was more than one way to reach the global target state and in some cases the operator and Barrel choose two different ways. In that case we reproduced the initial state in the simulator, and we executed the operator's steps to ensure the way the operator used to reach the target was valid. We also evaluated all the different paths to reach the state emerged thanks to Barrel in order to find the one that better fit for the purpose. All possible alternatives have always proved to be equivalent, from a

conceptual, implementation, and efficiency point of view. We decided to implement what in our opinion was the most intuitive in order to facilitate the code readability.

During the simulation, Barrel was always able to found all the possible failure scenarios, covering both faults and crashes, by taking into account the unexpected behavior of the applications and considering the possible dependencies fault (i.e., when a dependency is no longer satisfied), even in response of crashes. All the deployment and failure recovery operations have been successfully validated.

Chapter 6

Conclusions

6.1 Contributions Recap

The objective of the thesis was to design and prototype a Kubernetes Operator for deploying and managing the IotSnap microservice-based platform. This was done in the scope of an internship in Extra Red, which is the company leading the development of IotSnap itself. In particular, the Kubernetes Operator had to target the IotSnap Root subsystem, made of two databases, an object storage and four Java Spring services.

The presented operator meets all the requirements: It successfully creates and manages all the Kubernetes and OpenShift objects necessary in an IotSnap Root installation, i.e., Deployments, Services, Routes, and Secrets. The deployed services were PostgreSQL and MongoDB databases, the MinIO object storage, and four services developed by Extra Red: AgentKeeper, which is responsible for the software provisioning, CloudStorage that is a middleware between AgentKeeper and MinIO to store and retrieve software update binaries, CloudAdmin to administrate the cloud, and CloudAdmin GUI that is the user interface to manage the entire cloud.

Among the operators' features we found that backup and monitoring can provide additional business value by automating two operations that are currently performed manually. The IotSnap Root Operator allows to easily activate and deactivate those features, by handling all the orchestration complexity to perform this context-specific operations. We exploited the backup by injecting sidecars into the services we want to backup, while monitoring is implemented using Telegraf that retrieve application metrics

from the IotSnap Root microservices and store it inside an InfluxDB for further analysis and automated alerts. The operator has reached the fifth and last level described in the Operator Framework by offering all the features provided by this automation approach that were suitable in this context and which were required by Extra Red.

Moreover, the IotSnapRoot operator has been tested with both unit and end-to-end testing, and has been validated and tested for resiliency and fault-tolerance by exploiting the fault-aware management protocols. We modelled the IotSnap Root subsystem with the TOSCA-based management protocols and using the tool Barrel we validated such model. Thanks to the Barrel Planner we compared the operator's steps to reach a desired status and the steps planned by Barrel, in order to validate the operator's behavior. We considered the possible causes of fault and we wrote tests to validate all this paths. The model considered also the possibility of unexpected behaviors of the applications, by simulating crashes and the consequent faults that they may cause.

6.2 Lessons Learned

The operator-sdk is designed for the development of operators for individual applications rather than application stacks. Although it is still possible to manage entire stacks, as demonstrated by our prototype, the larger grain does not allow to go very deeply into the details without complicating the operator code, obtaining a very large and complex codebase, which may become difficult to maintain. The IotSnap Root subsystem is made up of different complexity level applications. In particular, the operator developed proved to be capable of managing Java Spring applications developed by Extra Red and the MinIO component, while still presenting some critical points in database management. Among the reasons of the discrepancy, we have identified not only the difference in complexity between the services, as the databases are much more complex and delicate objects to manage, but also the fact that the applications developed by Extra Red had a context-dependent behavior. Since they had been implemented for this product, some issues were already handled within the application. Also, the fact that they were all services in Java Spring meant that their management was very similar. With a good reuse of the code, the management of four components required very few efforts compared to the management

of only one of them. On the other hand, a database is a much more generic component, which must interface with many more applications and which has a more complex management.

When we designed the operator we decided to respect as many guidelines and best practices as possible, in order to fully evaluate the pros and cons of what the standard approach to the design and implementation of an operator should be. Many of the indications have proved to be extremely useful and valid during implementation, but we believe that some are not appropriate for our context, or at least that in this specific case they involve more limits than advantages. Among these best practices, what was undoubtedly the most binding is the choice to not deploy custom resources with the operator. As discussed in Section 3.2.1, there are operators for PostgreSQL and MongoDB, developed and tested by many people for very different use cases. These operators are certainly able to manage the databases better than we could manage. The choice to respect the constraint that imposed not to deploy custom resources with the operator prevented the use of these external operators in the development of the IotSnap Root Operator. The alternative was to move PostgreSQL and MongoDB out of the scope of IotSnap Root and manage them as architectural pieces by themselves, with their operators. We excluded this possibility because of the requirements imposed by Extra Red, which wished all seven IotSnap Root microservices to be managed by the same operator.

However, after having developed the operator, we would suggest to consider other possibilities too regarding this best practice. The guidelines were written with public operators in mind that anyone could use. The creation of dependencies between operators would have created problems, and some operators could have not work properly due to the lack of other operators. We agree with the opinion that tying the user to install one operator before another to make the second one work is actually a best practice. However, in our case the operator is not public, as it is intended for internal use only. Furthermore, the deployment is not done manually but from an automation pipeline that prepares clusters and namespaces for the applications purchased by the customer. Since the pipeline will also install the IotSnap Root Operator, we can assume that installing the operators it depends on first is acceptable. In this way we could violate the best practice without risk and have the IotSnap Root perform the deployment of a custom resource

for the other two operators. Then, the IotSnap Root Operator would handle only its components, while PostgreSQL and MongoDB would be managed by their respective operators.

Some unwanted behaviors that we have not considered may emerge from this different approach, but we feel at least to recommend their evaluation, perhaps through the modification of the prototype. In this specific case, the advantages could be greater than the disadvantages. The current IotSnap Root Operator implements the opposite approach, i.e., the one recommended by the best practices, providing a good comparison.

6.3 Future Work

For future work, together with Extra Red we plan to engineer and extend the current prototype of the IotSnap Root Operator to automate the deployment and management of the whole IotSnap platform in production. For example, this can be done by developing operators for the other IotSnap subsystems and by externalizing the backup and monitoring services to use them on all the IotSnap microservices.

The backup service, as well as the monitoring service, is currently included in the same custom resource as the IotSnap Root subsystem. However, they should be moved to another controller, as they are not IotSnap Root services but external services. With a view to automating the entire IotSnap platform with operators, backup and monitoring could be services that apply to the resources of multiple operators, therefore it is good that they are moved to a separate operator or at least in a dedicated controller and a custom resources. The main reason why this has not been done in the current implementation of the prototype is the lack of adequate tools to do it at the time of development.

The critical point was the backup. To achieve this, it was necessary to inject a sidecar into the Pod of the service we wanted to back up. The changes to this choice are widely discussed in the Section 4.5. In the current implementation, the sidecar with the backup microservice is injected during the deployment of the component. Even in the case the backup is activated or deactivated subsequently, a new deployment will be carried out. To insert the sidecar into the deployment it is necessary not only to be inside the same controller, but also that the backup's custom resource coincides with that of the component

you want to backup. Otherwise at the time of deployment the controller does not see the backup custom resource, as the one that fired the request is the one with the specification of the applications, and similarly when the backup custom resource is processed the other is unknown. Therefore, keeping this strategy it is not possible to separate the custom resources neither the controller nor to obtain separate operators.

An alternative approach, to be preferred, is to use admission webhooks. Once registered, Kubernetes invokes the admission webhook before deploying a resource. The server running on that webhook can inspect the resource and deny its deployment, or modify the resource before it is deployed. The latter case is the one of our interest: With an admission webhook we could intercept the deployment of the components to be backed up and inject the backup sidecar into the Pod. With admission webhooks this operation could be carried out with a separate custom resource and a controller, inside the same operator or in an external one.

However, at the time of implementation to set up an admission webhook it was necessary to create a dedicated server. This server is a component totally detached from the operator and its controller, to the point that it can be developed separately and in another language: The two components have nothing in common. Once the server capable of injecting the sidecars has been obtained, the operator only has to deal with the deployment of that server and the admission webhook that points to the server.

This system is strongly in contrast with the operator's best practices, according to which all the business logic must be within the operator. For this reason the APIs for admission webhooks have been integrated into the operator-sdk. Now, the operator creates an admission webhook which registers itself as a server, and the admission webhook code is integrated into the controller [26].

Unfortunately, the operator-sdk update containing these features was released a few days after the internship ended. However, when we developed these features we discovered that the team was working on integrating the admission webhooks[1]. For this reason

[1]There was a work-in-progress pull request on GitHub to integrate the admission webhook in the operator lifecycle manager, i.e., the tool that installs and manages operators (https://github.com/operator-framework/operator-lifecycle-manager/pull/1102). Moreover, Kubebuilder had already the code to implement admission webhook scaffolding code inside a controller and there was the plan to integrate this code in the operator-sdk (https://github.com/operator-framework/operator-sdk/pull/2858).

we decided to wait to complicate things pending the new update. Instead, we have implemented the injection of the sidecar using the same controller, in order to obtain a prototype that simulates the final solution in order to experiment and evaluate the benefits.

The next steps in the development of the operators will certainly have to involve the implementation of the webhook admissions. Then, it will be possible to move the backup and monitoring logic into two separate operators.

As in the case of IotSnap Root, the development of a Kubernetes Operator could also bring advantages for the other subsystems of IotSnap. In particular among the other subsystems we find: The ingestion system, the monitoring system, business intelligence and interoperability, and the authentication manager. The data ingestion system is an extended version of the TICK stack (Telegraf, InfluxDB, Chronograf, and Kapacitor), therefore it could benefit from existing operators, as in the case of PostgreSQL and MongoDB. Also in the Authentication Manager subsystem there is the official Keycloak operator. Instead, as IotSnap Root, the Interoperability and Business Intelligence subsystem is composed almost exclusively of services developed by Extra Red, therefore a custom operator could bring many benefits. Furthermore, since these services are written in Java Spring, their management is very similar to that of the IotSnap Root microservices and the operator developed during the internship can be a good starting point. To achieve optimal code reuse, parts of the IotSnap Root Operator code may be outsourced as libraries. Instead, we do not recommend automating the deployment and management of this subsystem using the IotSnap Root Operator, even if with a separate controller and custom resource. Being two subsystems with very different objectives, and since the Business Intelligent subsystem may not always be present, it is good to keep the management of the two portions of IotSnap very separate.

Once all operators have been developed, it would be interesting to understand how to automate the subsystems interaction. As we have already said, although not recommended by the guidelines, in this context it might make sense to use cascading operators, i.e., to allow an operator to deploy custom resources managed by another operator, while leaving the Operator Lifecycle Manager (OLM) handling the installation and upgrade of the operators. This would open up new opportunities. For example, a new higher

level operator could take the place of part of Ansible automation. Instead of managing all IotSnap operators from outside, a high-level operator could manage the whole platform, providing for the creation and configuration of all custom resources. Moreover, this operator could manage the creation of PVC and ConfigMap as it would be aware of the endpoints of each subsystem. Also, it would allow labels to be set consistently for all subsystems. From the user's point of view, the management of the installation and customization of IotSnap for a customer would turn into the writing of the yaml which describes a new type of custom resource IotSnap, rather than a configuration file to be passed to the Ansible automation. The Ansible scripts themselves could be simplified so that they only have to assemble this custom resource. Finally, such a solution would be much more Kubernetes-oriented than the current approach. As mentioned above, it remains to be verified which negative aspects could emerge from this approach, while it is not recommended by the creators of the Operator Framework, the advantages in this specific context could be worth violating this best practice.

Bibliography

[1] The Kubernetes Authors. *Kubernetes Operator Documentation*. The Linux Foundation, September 2019. Kubernetes v1.17.

[2] The Kubernetes Authors. *What is Kubernetes*. The Linux Foundation, November 2019. Kubernetes v1.17.

[3] The Kubernetes Authors. *Kubernetes Concepts*. Kubernetes, v1.17 edition, March 2020.

[4] The Kubernetes Authors. *Kubernetes Tasks*, chapter Set up High-Availability Kubernetes Masters. Kubernetes, v1.17 edition, March 2020.

[5] T. Binz, C. Fehling, F. Leymann, A. Nowak, and D. Schumm. Formalizing the cloud through enterprise topology graphs. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 742–749, 2012.

[6] Tobias Binz, Uwe Breitenbucher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. Opentosca - a runtime for tosca-based cloud applications. In Samik Basu, Cesare Pautasso, Liang Zhang, and Xiang Fu, editors, *Service-Oriented Computing: 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings*, volume 8274, pages 692–695. Springer, December 2013.

[7] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Tosca: Portable automated deployment and management of cloud applications. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Advanced Web Services*, pages 527–549. Springer New York, 2014.

[8] Steve Bradbury, Brian Carpizo, Matt Gentzel, Drew Horah, and Joël Thibert. Digitally enabled reliability: Beyond predictive maintenance. *McKinsey & Company*, October 2018.

[9] Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Modelling and analysing cloud application management. In Schahram Dustdar, Frank Leymann, and Massimo Villari, editors, *Service Oriented and Cloud Computing - 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015. Proceedings*, volume 9306, pages 19–33. Springer, September 2015.

[10] Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Fault-aware application management protocols. In Marco Aiello, Einar Broch Johnsen, Schahram Dustdar, and Ilche Georgievski, editors, *Service-Oriented and Cloud Computing - 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings*, volume 9846 of *Lecture Notes in Computer Science*, pages 219–234. Springer, September 2016.

[11] Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Modelling and analysing cloud application management. In Marco Aiello, Einar Broch Johnsen, Schahram Dustdar, and Ilche Georgievski, editors, *Service-Oriented and Cloud Computing - 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings*, volume 9846 of *Lecture Notes in Computer Science*, pages 219–234. Springer, September 2016.

[12] Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Fault-aware management protocols for multi-component applications. *Journal of Systems and Software*, pages 189–210, 2018. *In press, DOI: 10.1016/j.jss.2018.02.005.*

[13] Antonio Brogi, Antonio Di Tommaso, and Jacopo Soldani. Validating tosca application topologies. In Schahram Dustdar, Frank Leymann, and Massimo Villari, editors, *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*, volume 1: MODELSWARD, pages 667–678. SciTePress, 2017.

[14] Antonio Brogi, Jacopo Soldani, and PengWei Wang. Tosca in a nutshell: Promises and perspectives. In Massimo Villari, Wolf Zimmermann, and Kung-Kiu Lau, editors, *Service-Oriented and Cloud Computing*, pages 171–186. Springer Berlin Heidelberg, 2014.

[15] George Candea, Aaron Brown, Armando Fox, and David Patterson. Recovery-oriented computing: Building multitier dependability. *Computer*, 37(11):60–67, December 2004.

[16] Jaafar Chraibi. Enterprise kubernetes with openshift. *Red Hat OpenShift Blog*, February 2020.

[17] The Docker Company. Docker and red hat announce major alliance, September 2013.

[18] Richard I. Cook. How complex systems fail. *Cognitive Technologies Laboratory, University of Chicago*, January 1998.

[19] Hongchao Deng. Operator SDK's initial commit on its github repository, February 2018.

[20] Jason Dobies and Joshua Wood. *Kubernetes Operators - Automating the Container Orchestration Platform.* OReally, March 2020.

[21] Justin Ellingwood. An introduction to kubernetes. *Digital Ocean Community*, May 2018.

[22] Christian Endres, Uwe Breitenbücher, Michael Falkenthal, Oliver Kopp, Frank Leymann, and Johannes Wettinger. Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications. In *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*, pages 22–27. Xpert Publishing Services (XPS), 2017.

[23] Joe Fernandes. Openshift and kubernetes: Where we've been and where we're going part 1. *Red Hat OpenShift Blog*, December 2018.

[24] Cloud Native Computing Foundation. cri-o project, June 2020.

[25] Eclipse Foundation. Eclipse winery. *Read the Docs*, May 2020.

[26] Operator Framework. Admission webhook. Operator SDK Documentation, June 2020.

[27] Operator Framework. Unit testing, June 2020.

[28] Operator Framework. Using the operator sdk's test framework to write e2e tests, June 2020.

[29] FreeBSD. Freebsd 4.0-release announcement, March 2000.

[30] Brian Grant and Kubernetes SIG. Kubebuilder's initial commit on its github repository, March 2018.

[31] Jim Gray. Why do computers stop and what can be done about it? In Marco Aiello, Einar Broch Johnsen, Schahram Dustdar, and Ilche Georgievski, editors, *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, CA, USA*, pages 3–12. IEEE Computer Society, 1986.

[32] Lars Grunske, Bernhard Kaiser, and Yiannis Papadopoulos. Model-driven safety evaluation with state-event-based component failure annotations. In George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens Szyperski, and Kurt Wallnau, editors, *Component-Based Software Engineering*, pages 33–48. Springer Berlin Heidelberg, 2005.

[33] Red Hat. *OpenShift Container Platform Product Documentation*, chapter 27: Configuring Persistent Storage Using Azure File. Red Hat Customer Portal, 3.11 edition, 2018.

[34] Red Hat. Facing permission issue when creating application using postgres image. Red Hat Customer Portal Knowledge Base, April 2020.

[35] Red Hat. *OpenShift Container Platform Product Documentation*, chapter 4: Dynamic provisioning: Considerations when using Azure File. Red Hat Customer Portal, 4.3 edition, 2020.

[36] Solomon Hykes. Lightning talk - the future of linux containers. PyCon 2013, March 2013.

[37] Einar Broch Johnsen, Olaf Owe, Ellen Munthe-Kaas, and Juri Vain. Incremental fault-tolerant design in an object-oriented setting. In *Proceedings Second Asia-Pacific Conference on Quality Software*, pages 223–230, February 2001.

[38] Oliver Kopp, Tobias Binz, Uwe Breitenbücher, and Frank Leymann. Winery - a modeling tool for tosca-based cloud applications. In Samik Basu, Cesare Pautasso, Liang Zhang, and Xiang Fu, editors, *Service-Oriented Computing: 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings*, volume 8274, pages 700–704. Springer, December 2013.

[39] Miroslav Lachman. *FreeBSD Jails Wiki*. FreeBSD, May 2008.

[40] Frank Leymann. Cloud computing. *it - Information Technology*, 53:163–164, 2011.

[41] Linux-VServer. Vserver news, November 2003.

[42] Linux-VServer. Vserver hall of fame, May 2017.

[43] Guang Ya Liu and Danielle Mustillo. Does the SDK and kubebuilder team had some discussion to merge those two projects to one project for easy maintain?, February 2020.

[44] Lobaro. Restic backup docker container, June 2020.

[45] Josh Manning. *Operator Best Practices*, March 2020.

[46] Craig McLuckie. From google to the world: the kubernetes origin story. *Google Cloud Blog*, July 2016.

[47] Cade Metz. Google open sources its secret weapon in cloud computing. *Wired*, June 2014.

[48] Microsoft. *Azure Product Documentation - Features not supported in Azure Files*, September 2019.

[49] MinIO. Minio operator's pulls counter. Docker Hub, May 2020.

[50] OASIS. Cloud service archive (csar). *OASIS Standard*, November 2013.

[51] OASIS. Topology and orchestration specification for cloud applications. *OASIS Standard*, November 2013.

[52] OASIS. Tosca simple profile in yaml version 1.0. *OASIS Standard*, December 2016.

[53] Red Hat Press Office. Red hat to acquire coreos, expanding its kubernetes and containers leadership. *Red Hat Press Releases*, January 2018.

[54] OpenShift. Coreos projects and community, January 2018.

[55] The Containers organization. Podman, June 2020.

[56] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3):677–692, 2019.

[57] Brandon Philips. Introducing the operator framework: Building apps on kubernetes. *CoreOS Blog*, March 2018.

[58] Rob Pike. Another go at language design. In *Stanford EE Computer Systems Colloquium*, HP Auditorium, Gates Computer Science Building B01, April 2010. Google, Stanford University.

[59] Joep Piscaer. *The Gorilla Guide to Kubernetes in the Enterprise*. ActualTech Media in collaboration with Platform9, May 2019.

[60] Siamak Sadeghianfar. Kubernetes ingress vs openshift route. *Red Hat OpenShift Blog*, September 2018.

[61] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

[62] Integreatly Integration solutions by Red Hat. Grafana operator's github repository, March 2020.

[63] Unix. *Unix Seventh Edition Manual, Section 2, chdir*, 1979.

[64] Dan Walsh. Skopeo 1.0 released. *Red Hat Blog*, May 2020.

[65] Michael Wurster, Uwe Breitenbücher, Michael Falkenthal, Christoph Krieger, Frank Leymann, Karoline Saatkamp, and Jacopo Soldani. The essential deployment metamodel: A systematic review of deployment automation technologies. *SICS Software-Intensive Cyber-physical Systems*, May 2019.