SCTE CABLE-TEC EXPO 22
SEPTEMBER 19-22 • PHILADELPHIA, PA

2022 Fall Technical Forum
SCTE • CABLELABS • NCTA

A Unified GitOps Continuous Deployment Approach for Telco Hybrid Workloads

A Technical Paper prepared for SCTE by:

Stephan Salas
DevOps Engineer
Comcast, Inc.
1800 Comcast Technology Center, Philadelphia, PA, 19103
+1 267-260-0881
stephan_salas@comcast.com

Ruibing Hao, Ph.D
Distinguished Engineer
Comcast, Inc.
1800 Comcast Technology Center, Philadelphia, PA, 19103
+1 267-260-0881
ruibing_hao@comcast.com

Table of Contents

List of Figures

List of Tables

1. Abstract

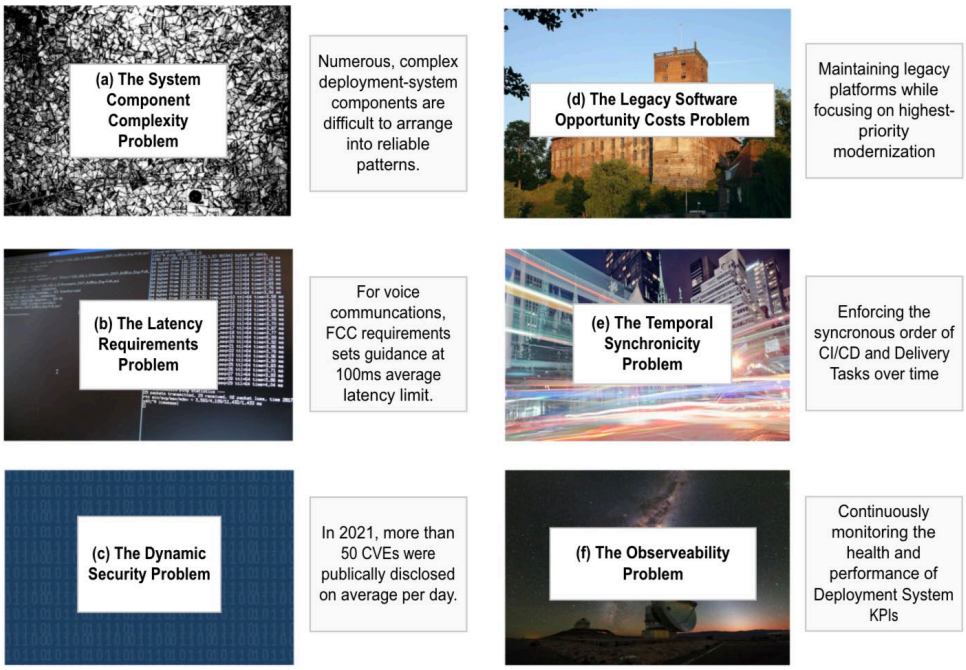
The telecom industry has moved toward a hybrid of cloud-native and virtualization technologies without a single, unified deployment approach for a variety of DevOps needs. While containerization and virtualization have both been used to solve a wide set of technical challenges in our industry, it is estimated that at least 30% of workloads worldwide still leverage virtualization technologies such as OpenStack [1]. For instance, while containerization might be advantageous for certain Layer 7 Workloads, it may be non-performant for Session Initiation Protocol (SIP) and Real-time Transport Protocol (RTP) processing needs. This trend will continue due to long-term investments to sustain both current operations and embrace more modernized ways of operating with new types of applications and infrastructure. This difference in needs across telecom organizations has led to the use of a diverse and complicated set of continuous integration and continuous delivery/deployment (CI/CD) tools and infrastructure arrangements. Unfortunately, the tendency of increasing technical-tool diversity is reflected by an increased division of organizations by technical expertise, which in turn can often-times prevent widespread adoption of modern CI/CD technologies among these organizations [2].

In this paper, we propose an approach and a framework to expand GitOps-based deployment orchestration automation into the virtualization stack, by leveraging customized Kubernetes Operators, ArgoCD, and Argo Workflows, Open Container Initiative (OCI) Containers, and Packer [3-7]. We demonstrate the feasibility and practicality of this approach on OpenStack with the help of an open source, full-stack voice over internet protocol (VoIP) implementation and Traefik HTTP Load Balancer [8]. The combination of these technologies enables several advanced deployment capabilities for OpenStack such as canary deployments and scaled rollouts. This solution has the potential to converge our industry toward a unified and modern CI/CD approach for DevOps teams and smoothen the transition towards cloud-native platforms, while helping to prevent the disorganized “tool-sprawl” [9] required to sustain both legacy and modern tech-stacks.

2. Introduction

While open-source software is “eating the world” [10] by increasing the capability of technology organizations to improve their product-competitiveness, some estimates put the proportion of costs attributed to software maintenance and sustainment at a whopping 60% of overall software project expenditures [11]. From the first monolithic inventions of a burgeoning telecommunication industry to today’s highly digital approach to scaling by means of distributed services, the burden of managing increasing levels of complexity as a result of fast-changing technologies and needs continues to be a significant cost-driver for telecom organizations [12].

DevOps professionals across our industry continue to struggle with a diverse set of organizational and technical challenges in sustaining existing operations while also looking toward future development platforms. This constant struggle to balance the two priorities can cause organizations to lose track of the core issues that are initially involved in software delivery lifecycles (SDLCs) and to get distracted by day-to-day problems. Thus, we have found that it is helpful to reframe the causes of this phenomena as a narrow set of key challenges facing modern DevOps teams:



[13]-[21]

Figure 1 - Six Key Challenges Facing Software/Systems Delivery Teams

These kinds of challenges have been denoted by organizational academics as "technical strategic bottlenecks", which drive increased reliance on a particular cross-section of expertise within a company to solve [22]. While these bottlenecks can often be viewed as both opportunities and challenges within organizations, there are certain problems that are inherently more difficult to solve than others, such as Fig. 1(a) and Fig. 1(d). A common tactic in mitigating these issues for deployment teams is to outsource these problems into separate systems, tools, and SMEs within an organization – all of which take a non-trivial amount of productive engineering time [23]. It is thus our view that the goal of every DevOps team should ideally be to manage the widest possible problem-set with the least and simplest possible tools.

Unfortunately, some analysts estimate that software-delivery teams interact with somewhere between 20-50 different tools daily [24], which often causes the all-too-familiar "too many tabs" issue for everyday engineers. The brain-drain of having to organize individuals or teams to manage this deployment tooling complexity can cost organizations a significant amount of focus in order to silo teams by expertise and can further hamper innovation due to lack of shared understanding of deployment platforms [25].

Multiple deployment platform availability can be a blessing in disguise -- Surveys of IT leaders demonstrate that organizations tend to shy away from using multiple deployment systems due to increased costs related to learning curves and lack of expertise. As of 2020, the percentage of organizations leveraging multi-cloud technologies was nearly half of those that chose to stick with on-premises solutions [26]. Even from a CI/CD software standpoint, most workloads as of 2020 run on older, more proven tools such as Jenkins [27] and TravisCI [28], with a minority of organizations choosing newer and more powerful open-source technologies [29].

As a strategic approach to the challenge of multi-platform management and the key challenges listed in Fig. 1, this technical paper proposes a proof-of-concept system built upon OpenStack and a Kubernetes

Administrative Cluster (KADC). The central goal of our approach is to share a helpful set of solutions with engineering teams in our industry that are tasked with ubiquitously deploying, testing, and validating changes across disparate, complex systems.

Using common industry use-cases, we demonstrate both web-scale workload and VoIP workload deployments on the OpenStack platform using Git as a declarative information store. We also leverage custom Kubernetes Operators as service abstractions of core and auxiliary deployment logic. By providing a framework for how engineers might abstract deployment details in two key telecom use-cases, we demonstrate a possible solution for managing both legacy and newer forms of workloads using a single set of tools.

Such a solution is even more important today considering the tremendous amount of investment going toward infrastructure management in the telecom industry in tandem with decreased margins of traditional telecom products [30]. In response to increased technological innovation from both entrenched and burgeoning competitors, process automation is being touted as the top factor of cost effectiveness in our industry because it drives increased organizational agility and responsiveness to telecommunication customer needs [31]. With our proposed approach, we hope that we may help engineering teams deploy software more quickly and reliably, which will in turn contribute to increased resource efficiency and increased value within our larger industry.

3. A High-Level Deployment Architecture for Multiple Infrastructure Platforms

While there are many CI/CD methodologies in the open-source community that could accomplish the goals we have outlined in our introduction, the option we chose is the GitOps methodology [32]. As a self-contained approach to infrastructure management, the GitOps methodology seeks to provide high observability and re-useability of deployed state. The key initial considerations for choosing this approach were its simplicity, popularity among the industry, and ability to tie into many different deployment platforms.

3.1. GitOps Architecture

GitOps is an opinionated deployment framework with numerous valid setups used throughout our industry, however there are a few key attributes that most GitOps implementations have in common. These attributes are more clearly defined than a traditional DevOps setup that can result in a wide variety of unintended side effects and outcomes for operations teams:

Table 1 - Traditional DevOps vs GitOps Key Attributes

Deployment Question	Traditional DevOps Setup	GitOps Setup
Where is Deployment State Stored?	State is stored in databases and procedural deployment scripts.	State is implemented and stored in a declarative fashion in Git.
How is Deployment State Stored?	State may or may not be stored with its version history depending on the implementation.	State is stored in a way that supports immutable versioning and retains a complete history of changes.
How Often do Deployment State Processes Trigger?	A variety of custom-created, procedural deployment systems are typically leveraged only once to perform deployment process updates.	Software agents continuously compare a system's actual state to its desired state in order to enforce eventual consistency.
Who interacts with the Deployment State?	DevOps Engineers will typically be the primary Operators of a deployment system due to its complexity.	Developers and Code Reviewers interact with a Git interface (the "Git" in GitOps) through pull requests as a security measure to approve and commit final deployment state.

Based on our analysis, the benefits of using GitOps as opposed to traditional DevOps methods are threefold:

1. *Unification of Deployment State into a Single Location*

   Compared to many of the attributes of a traditional DevOps scheme, GitOps provides a much more streamlined and unified store of application and infrastructure state. The benefits of using the Git platform as opposed to others for this purpose is perhaps the most impactful reason why the methodology is becoming increasingly popular today, with an estimated 84% of developers considering themselves as active contributors to open-source tools [33] and 92% preferring Git as their primary source control software [34]. As a single data-store of infrastructure state, the open-source Git platform also serves the principal goal in this paper of reducing excessive DevOps tooling management.

2. *Declarative Deployment State*

   In addition to its unifying characteristics, GitOps also aids with separating minor, unimportant procedural details from state using the framework's declarative design. In contrast with traditional DevOps methods, specifying a group of declarative file manifests as state aids software engineers in organizing their deployments more effectively into logical units, and helps in increasing observability of what is currently deployed across different parts of their organizations.

3. *Secure Deployment State*

   From a security standpoint, most Git repository providers enable enterprise-grade functionality to log into repositories, pull down commits, and push new pull requests – just to name a few of the potential scenarios. In our case, by requiring pull-requests for changes to both infrastructure and application manifests, approvers can institute a code-review process that enforces certain requirements to deploy to different platforms or environments.

These three benefits serve to help mitigate many of the problems outlined in the introductory section's Fig. 1:

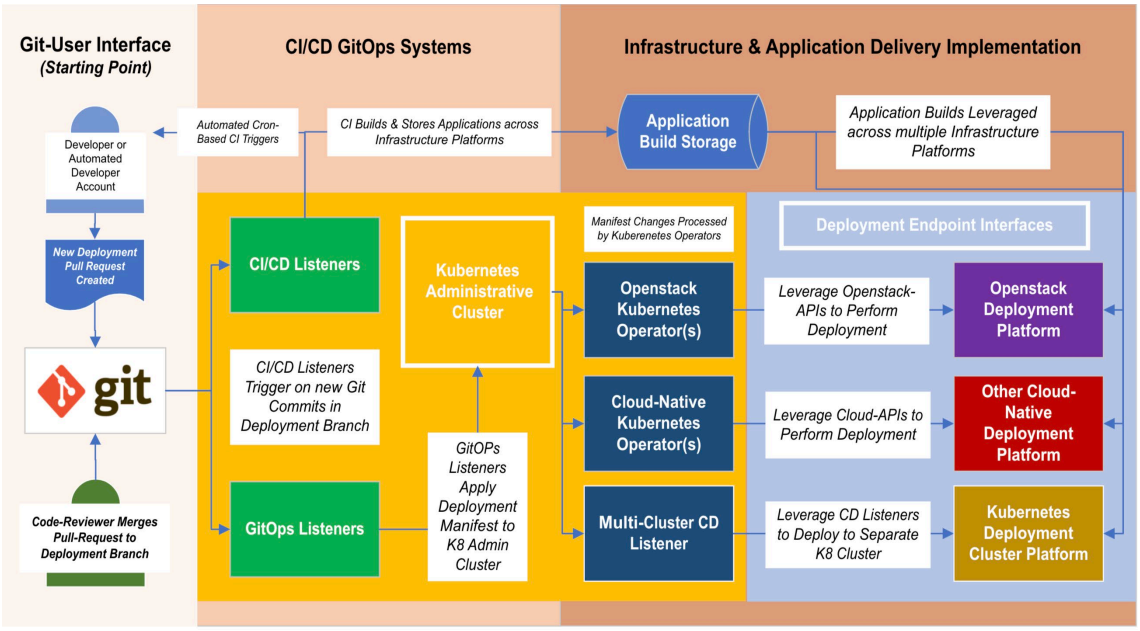
(a) **System Component Complexity Problem:**
- GitOps assists with reducing the number of components that store state in a deployment system.

(c) **The Dynamic Security Problem:**
- GitOps enables DevSecOps [35] via transparent declarative handling of secrets used in deployments.

(e) **The Temporal Synchronicity Problem:**
- GitOps can be used to aid in enforcing order of operations in deployments due to the time-based nature of Git and specifically because timestamps are associated with Git commits.

(f) **The Observability Problem:**
- GitOps enables high observability of deployment state in a Git repository using the Git CLI and related tooling.

3.2. Multi-Platform Deployments with CI/CD and GitOps



[36]

Figure 2 - CI/CD GitOps Multi-Platform Deployment Architecture

Combined with the popular Kubernetes Platform, GitOps provides consistent and highly observable deployment arrangements. In our proposed architecture, we introduce the unifying concept of the KADC, which is our continuous deployment orchestrator that leverages the following key components:

Git-based User Interface

The Git Interface serves as the deployment data-store that controls who has access to various repositories. Some examples of Git interfaces include on premise, cloud-hosted, and self-hosted systems. In our proposed Git interface, we envision the best-practice of developers using pull requests to commit infrastructure and application deployment changes after careful review from a group of authorized reviewers.

Continuous Integration/Continuous Delivery

The CI/CD Listeners' two responsibilities are to process application-builds within a storage context and to commit back to Git for cronjob-based automated deployment changes. For application builds, a CI/CD listener triggers build scripts upon commits to specific Git repositories. The build process may trigger testing and validation steps to verify that it is ready for deployment. Once ready, the listener pushes those built containers or images to an application build storage location.

Continuous Deployment using GitOps

GitOps Listeners enable continuous, asynchronous changes to infrastructure within targeted deployment environments based on Git repository commits. Engineers can program these systems to listen for certain changes and take automated actions. In our case, we designed the GitOps listeners system to deploy manifests (can be either YAML or JSON) into the KADC for further processing. If the manifests are Kubernetes native resources, they will be deployed directly in the target Kubernetes cluster; otherwise, these manifests describe resources deployed to non-Kubernetes platforms such as OpenStack or public clouds, and in this case KADC is used as a proxy.

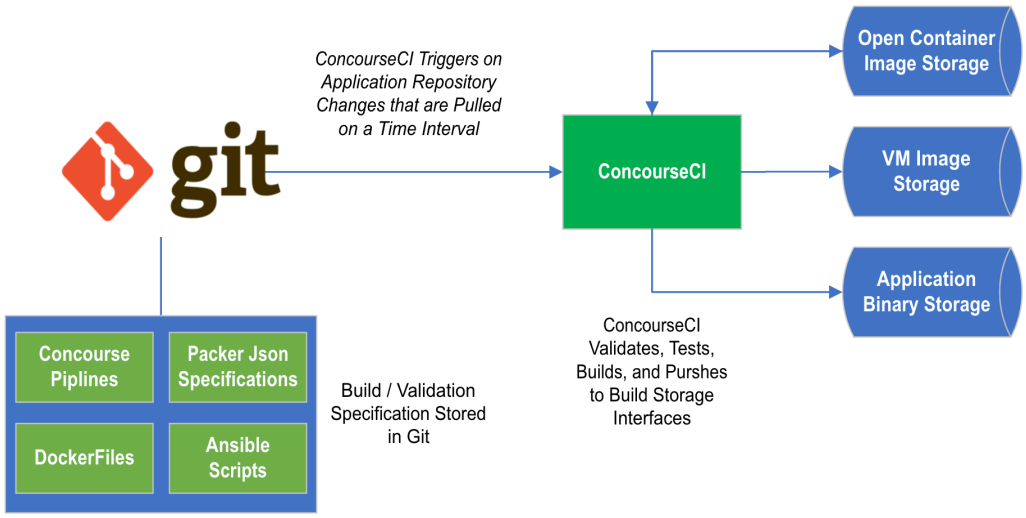
Application Delivery Implementation

The Infrastructure and Application Delivery Implementation contains the core logic to deploy manifests to various environments. This functionality is implemented with custom Kubernetes Operators in the case of OpenStack or public cloud deployments, and the built-in standard Operators in the case of Kubernetes. Depending on the *type* of manifest provided, either Kubernetes standard Operators or custom third-party Operators will handle the submitted manifest.

In all cases, Kubernetes resources hide the complexity of custom deployment logic with more simplified, declarative manifests that are easier to read than traditional procedural scripts.

4. CI/CD Listeners Implementation

While there are numerous options available that enable multi-platform CI/CD, ConcourseCI [37] and Tekton [38] were chosen for evaluation due to their maturity in the open-source ecosystem. In our implementation, we have narrowed down our scope to a ConcourseCI instance. We installed an on-premises instance of ConcourseCI inside our KADC, and also leveraged a shared instance installed on AWS. This setup acted as the "CI/CD Listener" noted in Fig. 2, and it accomplishes a variety of common application build, validation and integration tasks. The purpose of using this Kubernetes native tool is to further unify deployment state on top of a single platform. By implementing three key application-build types, as well as their corresponding testing and validation steps, we were able to integrate software build processes across the platforms listed in Fig. 2. (Kubernetes, OpenStack, and Cloud-Native) using industry-standard tooling:



[36]

Figure 3 - CI/CD for Multiple Platform Types

1. Open Container Image (OCI) Storage
    - Utilizes Dockerfiles [39] to create/push OCI container images
2. VM Image Storage
    - Utilizes Packer and Ansible Scripts [40] to create/push OpenStack VM images
3. Application Binary Storage
    - Storage of Application-Specific binaries (i.e. ".jar" file for Java [41], ".lib" for Golang [38]) that can be used by either a container or an virtual machine

Behind the scenes, ConcourseCI pushes the built container images into the OCI storage and pulls the necessary images from this storage to run each step of its pipeline. Therefore, there is a bidirectional link between ConcourseCI and Open Container Image Storage.

5. GitOps Listeners Implementation

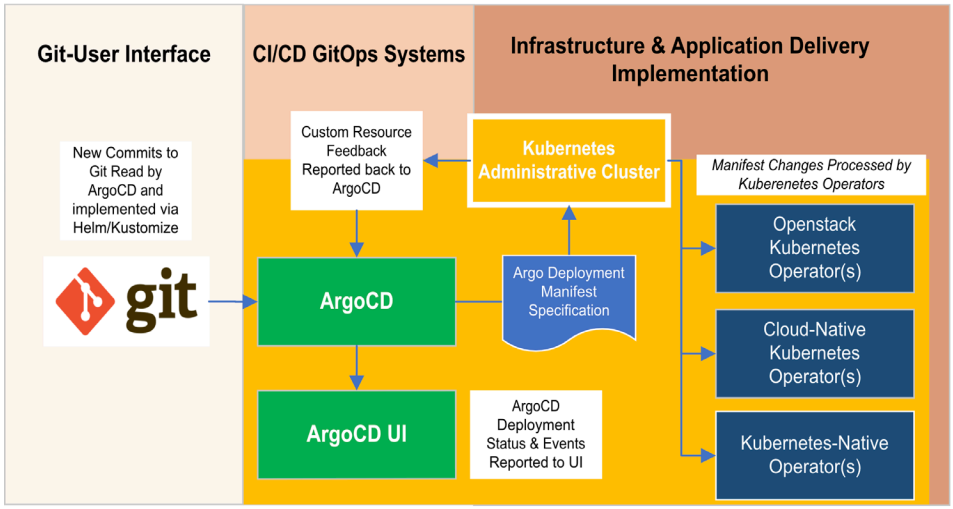
For the GitOps functionality abstractly mentioned in Fig. 2, two key opens-source DevOps components were chosen: *Argo Workflows [5]* and *ArgoCD [4]*.

Argo Workflows accomplishes the orchestration of a full stack deployment in which there can be dependencies between stack components. For instance, the passing of information from one stack component to the other so the latter can be configured properly is one example of this dependency. This tool is also ideal for scheduled and repeatable deployment tasks that would otherwise burden teams with manual steps, such as scheduling the scaling up of the stack in anticipation of peak hour traffic or scaling down in the inverse case.

In comparison, ArgoCD was specifically chosen to perform the deployment of a single stack component for continuous deployment integrations. Dependencies between deployment steps are handled in this case by Kubernetes Operators that implement custom logic in a manner that is more complex than is practical to implement in Argo Workflows.

The settings for both software packages are stored and controlled by the KADC, and this design furthers our overarching goal of deployment platform unification.

5.1. ArgoCD Implementation



[36]

Figure 4 - ArgoCD High-Observability Architecture

ArgoCD Deployment Manifest Resource

An ArgoCD Application is a Kubernetes Custom Resource (CR) that reacts to changes within a specific Git repository. Within an ArgoCD Application CR, the most important attributes are: 1) the Kustomize/Helm directory to listen on, 2) the number of times to retry a CR change before declaring failure, 3) whether to auto sync changes, and 4) which deployment customization approach to use.

Kustomize [42] and Helm [43] are the two most popular open-source deployment customization approaches to use within ArgoCD at the time of this writing. While Helm is a templating solution for Kubernetes that allows for major deployment details to be highly-reusable, Kustomize is more of a patching solution that allows you to replace specific fields without a template on a more case-by-case basis.

The two CRs status fields for an ArgoCD Application are the sync and health attributes. Because an ArgoCD Application refers to a Git repository for either a Helm or Kustomize deployment, its health and sync status are "all or nothing", meaning that for the Application to be considered fully deployed and healthy, all resources need to be successfully deployed *and* fully up to date with latest Git commit events. This information is propagated from the KADC into ArgoCD components for observability purposes.

ArgoCD UI

ArgoCD UI is the main user interface typically used to interact with for DevOps continuous delivery tasks. This interface provides a single place for both developers and DevOps engineers to view the status of their deployments of Kubernetes CRs.

The ArgoCD UI displays three key pieces of information that are useful to the end-user:

1. The health and status of each component of an ArgoCD Application
2. Kubernetes "Info" and "Warning" events associated with each component of an ArgoCD Application
3. The health & status of the overall ArgoCD Application

ArgoCD Health Checks

ArgoCD Health Checks are either Kubernetes-native (supported by ArgoCD "out of the box"), or custom-made with Lua [44] scripts for non-Kubernetes-native resources. For instance, we create several custom health checks for resources managed by our proposed OpenStack Operator.

ArgoCD Kubernetes Event Reporting

Kubernetes events are propagated to the ArgoCD UI for each CR deployed within a single ArgoCD Application. These events are published by a Kubernetes Operator and are meant to help users troubleshoot deployment issues and give more visibility into error details logged by KADC Operators.

5.2. Argo Workflows Implementation

Argo Workflows is an open-source container-native workflow engine for orchestrating tasks on Kubernetes. It is implemented as a set of Kubernetes custom resource definitions (CRDs) and its own custom Operator. The core primitive of Argo Workflows is the workflow resource, wherein each task of the workflow is implemented by a container, and the workflow itself contains a sequence of tasks with dependencies between tasks captured in a directed acyclic graph (DAG).

Argo Workflows was initially designed to run compute intensive jobs for machine learning or data processing but has been adopted to orchestrate continuous delivery tasks as well. In our proposed GitOps architecture, a workflow will be used to capture the steps required to deploy a stack of applications using a DAG. Each step of the DAG can have one or multiple success conditions that make sure this step is only considered as complete when its resources have been fully deployed and readily available. Each step is also typically responsible for the deployment of one component of the full stack, or a subcomponent of a complex component in the full stack.

Each workflow is captured as a Workflow CR in YAML format and can either be deployed to the KADC using an ArgoCD Application or directly into the KADC using a Kubernetes interface. The first approach is more appropriate when the component manifest requires much more information than is made available during deployment time. The second approach is more appropriate when the component manifest is relatively static and does not change often over time.

6. Custom Kubernetes Operators

KADC Custom Operators provide computational and logical separation of concerns for deployment to various platforms. These Operators come in the form of third-party software, and in the case of this paper, a set of custom Kubernetes Operators we developed that integrate with OpenStack. The "Operator design pattern" as described by the CNCF Whitepaper, splits functionality of CRs into controllers, which continuously reconcile changes from requested state to desired state in order to accomplish a deployment:

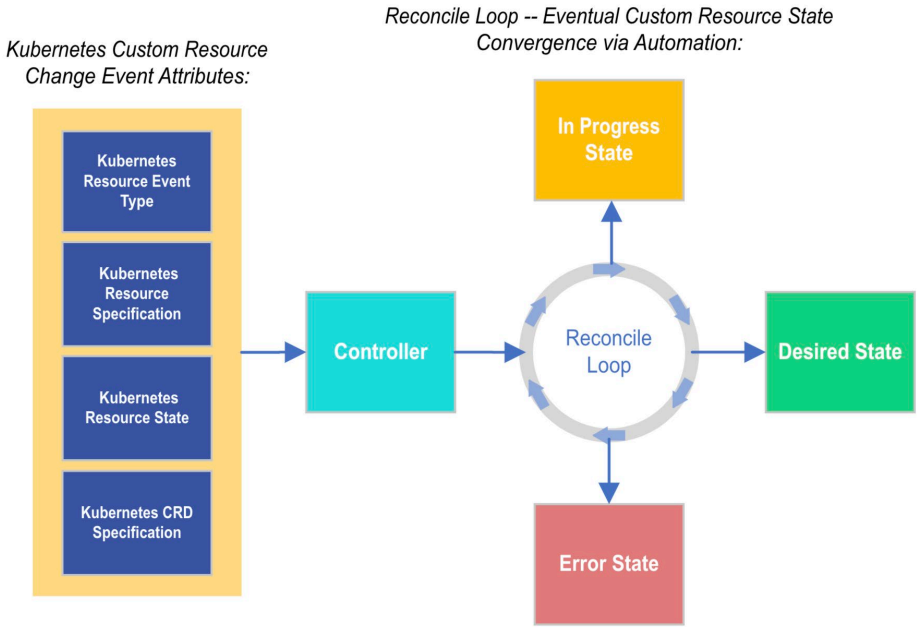


Figure 5 - Custom Operator Reconciliation Components

Operators can contain one or more controllers, which in turn typically manage one CR per controller. The controller is a code-bound component of a Kubernetes Operator and can be written in various languages that are compatible with the Kubernetes runtime. In our case, we chose the open source OperatorSDK Framework [45] and Golang Language [46], although any Kubernetes-compliant Operator implementation will work as well.

The fundamental primitive of a controller is the control loop, which reacts to state-change events until either the desired state is achieved ("reconcile" in Fig 3.) for a particular CR, or until it reaches a final error state. As the runtime-manager of CRs, the control-loop within a controller act as a time-bound polled reconciler of changes. It can also publish events that give a DevOps engineer further information about deployment checkpoint status or error details:

Custom Resource Definition Specification (CRD)

A custom resource is a conceptual representation of an object within Kubernetes. Included in a CRD specification are the structure of the object, the variables within the structure, and the datatype of each variable. Thus, this CRD primitive is highly configurable, and much like object-oriented programming, demands its own design considerations when developing custom controllers to manage them.

Event Types

Within a Kubernetes controller, there are three major events that are reconciled for the current state: 1) Create; 2) Update; and 3) Delete. The controller must have logic that handles each of these cases in a graceful manner for both happy-path and error-path situations so that it is reliable and feature-complete.

Current State

The current state is the latest requested state committed into the KADC. For instance, when the current state is updated, an update event is sent to the Kubernetes Operator along with the next state to be reconciled.

There are two forms of state that can change in a CR: 1) The specification field, and 2) the status field. The change of one of these fields will trigger the *Reconcile Loop*.

Reconcile Loop

The reconcile loop is a function that processes events in order to converge these events toward a desired state. Within a reconcile function, there are three possible outcomes:

1. Successfully process state event and don't requeue
2. Requeue the event to be processed again later in time due to an error scenario
3. Stop requeuing due to error scenario (with exponential backoff being an option for repeated errors)

With these three options, programmers can code controllers to be resilient to faults that may occur in the event of network issues or one-time errors, while also handling repeated errors gracefully with an exponential backoff option.

Operator services are hosted within the Kubernetes runtime as Deployments, Replica Sets, and Pods, and are easily configurable with the settings of these common Kubernetes resources. The main idea of hosting these Operators within the KADC runtime is to utilize high availability (HA) deployment capabilities inherent within the Kubernetes control plane that contributes to increased reliability of deployments.

7. Openstack Deployment Orchestration Architecture

OpenStack is a complex virtualization platform with many possible arrangements and use-cases. For deploying different kinds of workloads – namely VoIP and Web-Scale, it is important to first decide which API integration we wanted our Kubernetes Operators to interact with within the OpenStack Ecosystem. We could then decide how Operators should interface with this integration.

Upon careful exploration of available options, we decided to integrate with the popular Heat Orchestration Template (HOT or HEAT) APIs [47] because they leverage declarative resource templates that are more easily compatible with our chosen GitOps approach:

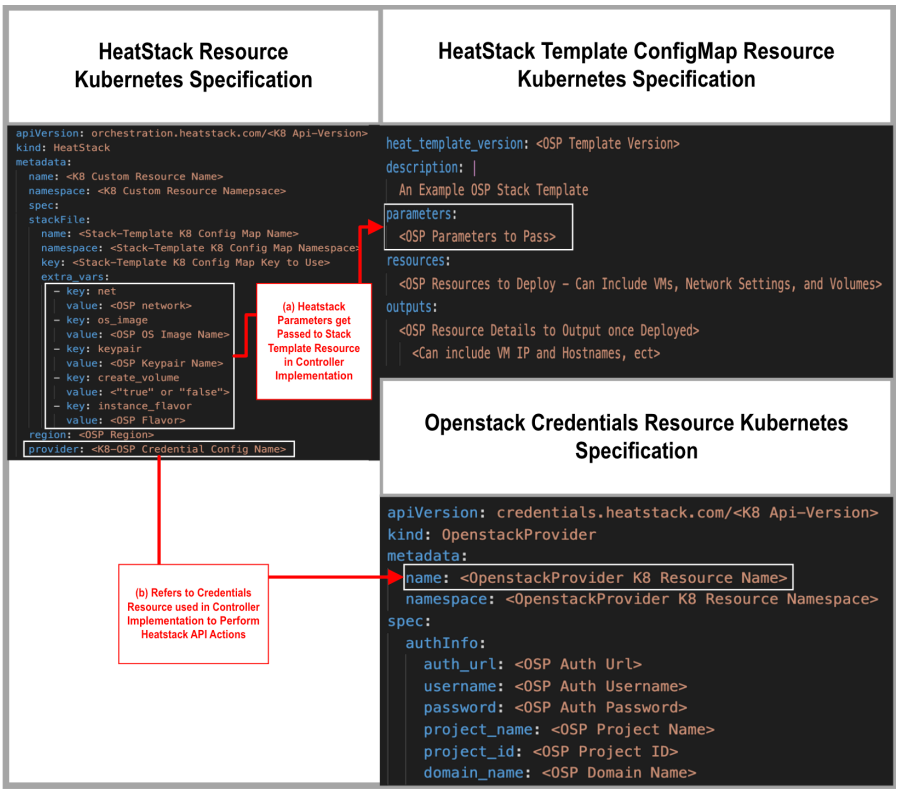


Figure 6 – Example Kubernetes Resource Specifications with Heatstack CRD

There are two essential components within our custom resource specification for OpenStack: 1) An orchestration template, and 2) An authentication template.

The resource scheme developed with the declarative specification listed in Fig. 6 allows for a high amount of flexibility in deploying key OpenStack resources. For instance, rather than having a set schema that declares which variables can be passed to a HEAT template, the "extra_vars" field in Fig. 6(a) can have an arbitrary number of parameters that work with a wide variety of Heat templates. The Heat-Template abstraction is then meant to be a highly flexible schema that serves a wide variety of use cases on the OpenStack platform.

With the "OpenstackProvider" resource listed in Fig. 6(b), we can further configure the authentication settings that we are using to interact with the HEAT APIs, which are needed for creation, update, and deletion of Heatstack CRs. This template-based approach is also applicable to other platforms such as cloud-native and other virtualized setups as well, as declarative API specifications have become popular within the IT Industry in general.

KADC resource specifications can thus be used to convert very broad requirements into specific ones, without a high amount of setup effort for simple deployments. In this example of a HEAT Template API interface, the outputs of the Heatstack Template are returned by the API, and these fields are populated as state in the Heatstack CR.

7.1. Core Deployment Orchestration

In this section we propose an opinionated way of utilizing the primitives we have designed in Fig. 6 so that we can create a base set of API interfaces with OpenStack for the HEAT Template primitive:

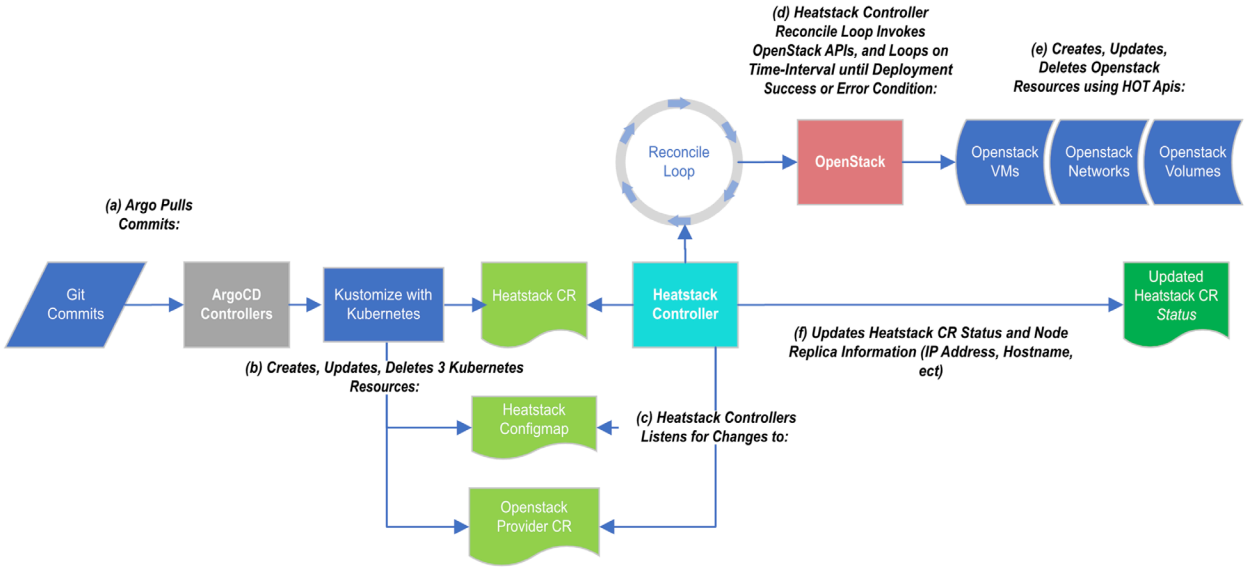


Figure 7 - Kubernetes Resource Specifications for Heatstack Deployment

With Fig. 7(a) and Fig. 7(b), we utilize ArgoCD's Kustomize interface (although Helm is also possible) to submit updates to each of the three Kubernetes CRs listed in Fig. 6. The controller then listens for change events for each of the custom resources and reacts to those changes in steps Fig. 7(c)-(e).

OpenStack API Control-Loop Logic

In the event of a bulk *create* event on resources, the Heatstack Controller reacts to the create events in Fig. 7(c), which will create the Heatstack CR, the Heatstack Config Map, and the OpenStack Provider CR as resources in Kubernetes. In step Fig. 7(d), the controller invokes an initial API call with the OpenStack API, however finalization of this API integration takes time. For example, a large-sized VM deployment can take a few minutes to a few hours to complete depending on the magnitude of scale you are targeting. Thus, it was important for us to design the Heatstack controller to poll the OpenStack resource it just created in order to validate the health of the deployment over time and report its state to our GitOps Listeners.

Heatstack Controller Status Update Implications

This idea of polling the HEAT-API for details on deployment state is fundamental to the implementation of resources that rely on the Heatstack, such as Load Balancers, TLS Certificates, and DNS entries, because each of these features rely on an up-and-running deployment. Even without these auxiliary features, reporting the status of the deployment back to ArgoCD via health checks enables better visibility of the deployment logic within the ArgoUI interface detailed in Section 5.

The status field of the Heatstack resource serves as the fundamental way that health checks are implemented, with a status field having to be "complete" for the Heatstack resource to be considered healthy in ArgoUI. Other details in Fig. 7(f) are also updated within the status field, as such:

1. **deploymentStatus**
   - Can be either: IN_PROGRESS, CREATE_COMPLETE, UPDATE_COMPLETE, or ERROR
2. **deploymentStatusReason**
   a. A string field that indicates success or error reasons
3. **outputs**
   - A list of key value pair objects that store critical data from HEAT deployments.

For each successful create, update and delete event, the deployment status gets updated with a simple tag that aids us in tracking the deployment status. The "outputs" field is updated in Fig. 7(f) with multi-VM IP and Hostname attribute details following a completed change event, which aids in health checks and further controller processing for auxiliary features.

Resilience Features in the Heatstack Controller Design

The control-loop approach for CR Status updates improves overall deployment resilience. Using control-loops, controllers can continuously integrate the latest changes committed to Kubernetes via ArgoCD while also validating previous changes or discarding them depending on the situation. This level of runtime control within our deployment implementation also allows for proper handling of errors. With exponential backoff capabilities within the control loop, we can eventually stop processing changes that are causing repeated and sustained errors over time, while also informing users through the "deploymentStatusReason" field of the underlying issue.

7.2. Auxiliary Deployment Orchestration

Operators leverage create, read, update, and delete (CRUD) APIs to orchestration HEAT-template resources. In our case, we have chosen to use a single controller within our OpenStack Operator called the "Heatstack-Controller" in order to manage these resources, while using other controllers as auxiliary integrations around this fundamental controller to supports dependent features.

Overall, the 5 key areas of solutions we incorporated into our OpenStack integration design via various APIs were:

Table 2 - OpenStack Deployment Architecture Components

Resource Deployed	API Used by Operators	Key Objective Accomplished	Leverages OpenStack API?
Heatstack(VM, Storage, Network Management)	OpenStack HEAT, aka "Heatstack Template" APIs	Orchestrate / Deploy VMs, Volumes, and networks to OpenStack using Images	Yes
Application Management	OpenStack Image APIs	Managed Packer-Built Images used by OpenStack VMs	Yes
DNS Management	VinylDNS API	Manage DNS Records in VinylDNS System	No
Certificate Management	Certificate Manager API	Manage Certificates tied to DNS Records	No
Load Balancer Route Management	Traefik Kubernetes CRD API	Manage Traefik-LB Routes Exposed on various HOT VMs	No

At the core of the deployment is the VM, Storage, and Network resource management solution, while several additional open-source ancillary components (VinylDNS, Certificate Manager, Traefik Kubernetes CRD provider) were chosen to demonstrate additional functionality [48-50]. These additional features were chosen because they are typically challenges that are faced by engineering teams in getting their applications to production and in managing complexity of common deployment setups on OpenStack. It is also important to note that these additional components may be replaced within this design with other software that has similar API functionality to support interchangeable components.

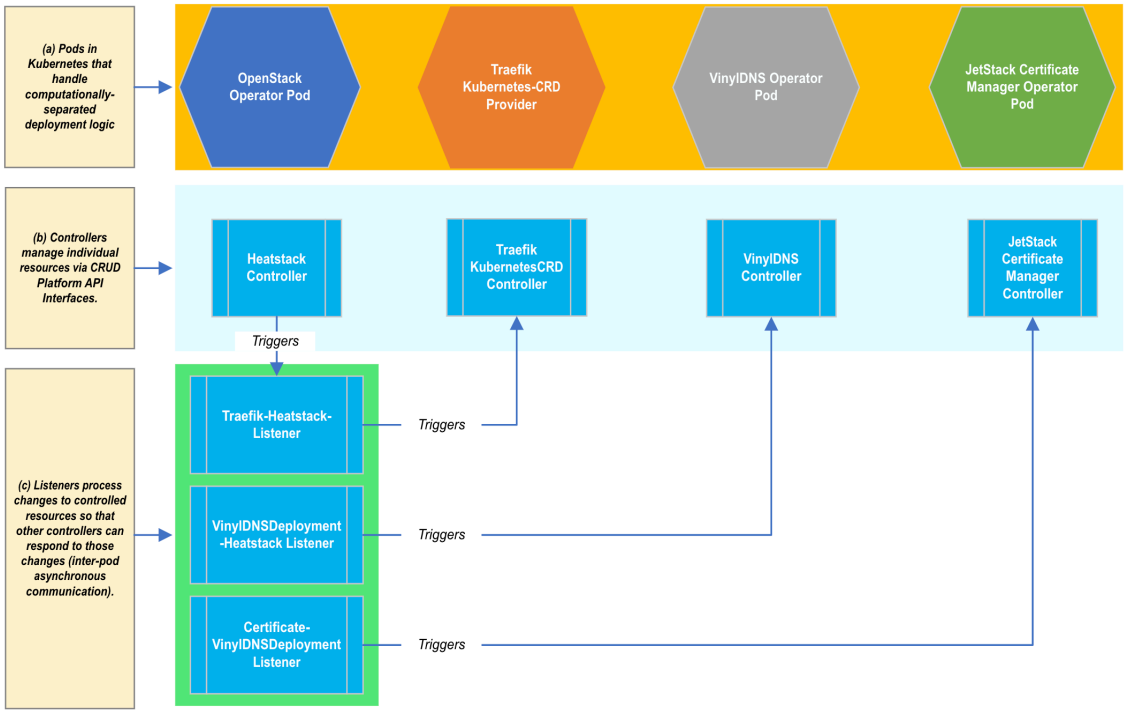


Figure 8 - OpenStack Kubernetes-Operator Architecture Components

There are two essential relationships between the resources listed in Table 2 and the Operator Architecture listed in Fig. 8, which are the *controller-resource relationship* and *the listener-resource relationship*.

In the *controller-resource relationship*, custom resources in Kubernetes are processed by various Operators to create new platform-specific implementations via their various API endpoints. In our case, the core resource being deployed is the Heatstack (described in Table 2) which supplies the declarative specification of resources supported by OpenStack. In the case of a Heatstack CR, a single controller will implement this relationship. In a similar way to the HeatStack Controller, the ancillary controllers - VinylDNS, CertificateManager, and Traefik, enable the management of DNS, TLS Certificates, and Load Balancing Routes.

In the *controller-listener relationship,* controllers listen to changes on *controlled* resources and react to those changes based on additional information captured through Kubernetes CR annotations to implement synchronous and orderly deployments. For example, the "Traefik-Heatstack-Listener" waits until a Heatstack has been fully deployed before exposing it through a Traefik Load Balancer Route using settings specified within the Heatstack CR annotations / metadata fields.

For each pod in Fig. 8(a), the controllers and listeners underneath match with the pod from a service perspective. This architecture separates concerns on both the computational level and from a logical standpoint. Using this clear separation of concerns, we built an architectural scheme with 4 pods that interfaces with OpenStack, Traefik, VinylDNS, and Certificate Manager:

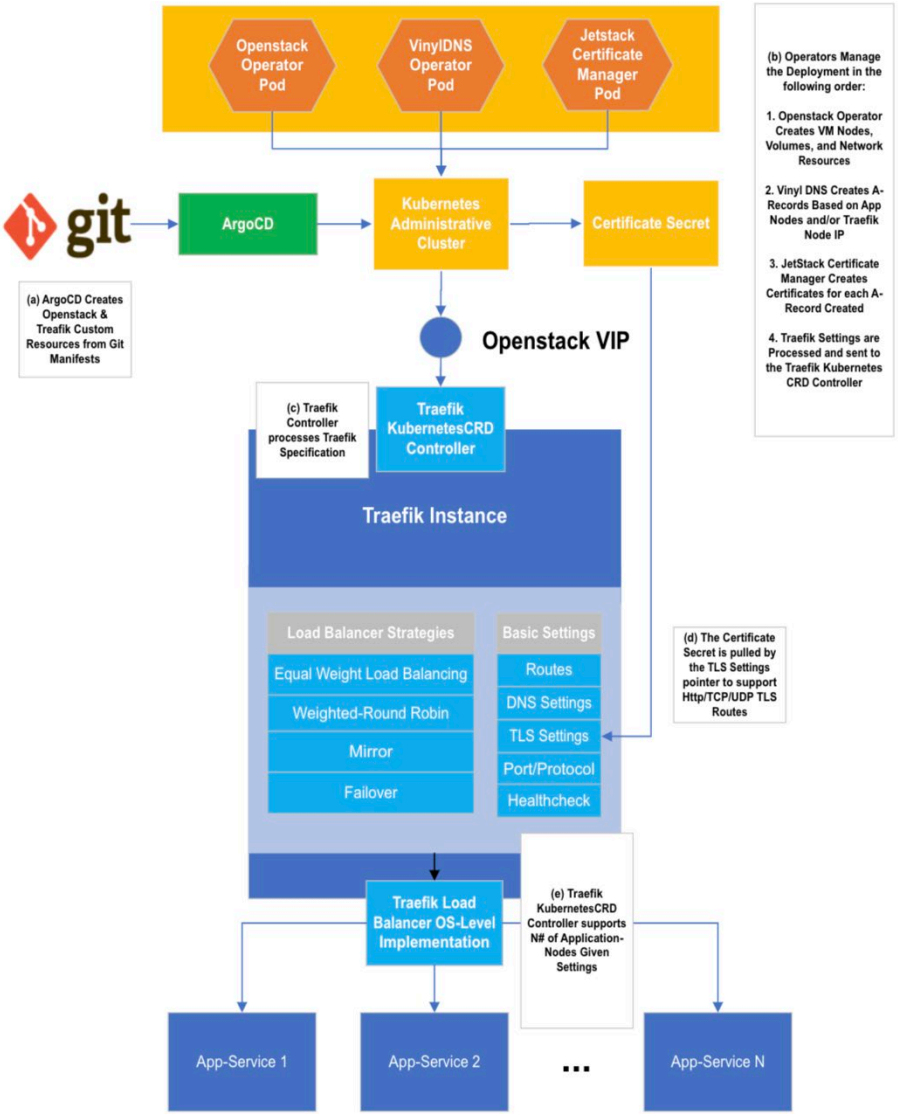


Figure 9 - Traefik Http-Route Integration with Auxiliary OpenStack Operators

Our GitOps implementation supports the synchronous processing of core and auxiliary components of a web-scale deployment as described in Fig. 9(a) and Fig. 9(b), and with all the components listed in Table 2. As mentioned earlier in this section, we leverage KADC Operators to ensure that core components are created before auxiliary components are processed. The key example in Fig. 9 is that Fig. 9(c) and Fig. 9(d) are processed after Fig. 9(b)1-3. This ensures that infrastructure VMs, VinylDNS and Certificates prerequisites are all created before services are exposed via the Traefik Load Balancer. After this initial work is performed, step Fig. 9(e) triggers with the settings passed to the Traefik KubernetesCRD provider-controller which exists on the Traefik Load Balancer instance itself, and this step processes the Traefik Route custom resources created by the Traefik-Listener controller in the KADC in order to expose groups of OpenStack VM services to load-balancer routes. The configurable settings within Traefik are listed in the Fig 9(e), split into load balancer strategies and basic settings for our ease of understanding. With a single exposed route, you can typically choose a single strategy to work with

depending on your needs, however virtually unlimited routes can be exposed with a single manifest specification in the KADC.

In addition to enforcing order in the bulk-create-case, in the case of a scalability update scenario the design also ensures zero-downtime deployments. This is accomplished with careful coding of custom KADC Operators for these common scaling cases:

1. **Scale up VM nodes**: Traefik listener will not process VM node scale-ups until the action is complete. Scaling up with the OpenStack HOT API does not delete existing nodes or recreate them, and thus this setup enables zero-downtime deployments.

2. **Scale down VM nodes**: Traefik immediately removes the necessary VM(s) from exposure in the event of a scale-down before the Heatstack-Controller deletes them. This also ensures zero-downtime deployments as well.

8. Advanced Deployment Capabilities for Web-scale Workloads

A web-scale workload is a component in many telecom products that use REST, gRPC, and other popular HTTP-based communication protocols. To demonstrate the augmentation of web-scale services with advanced load balancer strategies, we propose leveraging the weighted-round-robin strategy (listed in Fig. 9) within Traefik to allows assignment of different integer weights to groups service nodes, such that certain nodes can proportionally receive more traffic than others. Our aim in implementing this functionality is to demonstrate that our design can take advantage of the following advanced deployment capabilities not widely available in virtualization infrastructures and typically reserved for cloud-native/Kubernetes platforms:

- Blue/Green Deployment [51]
- Canary Deployment [52]
- Scaled-Rollout Deployment [53]

The general process by which the weight changes are leveraged is using the previously mentioned integration with Git in Fig. 2, which is accomplished with either manual Git commits, or with automated Git commits using a service account triggered by Argo Workflows or ConcourseCI:

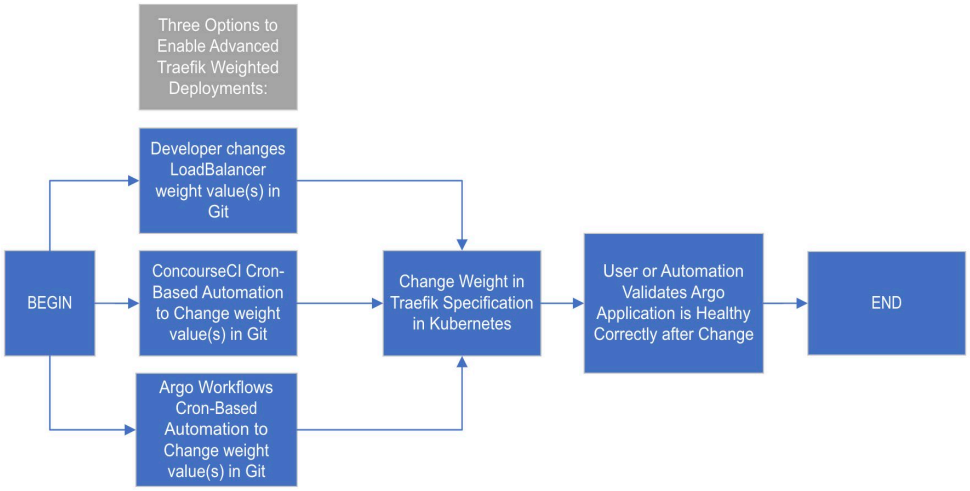


Figure 10 - A Simplified Advanced Deployment Process for Traefik Orchestration

Blue-Green Deployment

For a blue-green deployment, two separate weight changes for two groups of applications occur. Within the Traefik Load Balancer specification, we can group applications under label "A" and label "B" and give one group a weight of "1", while the other gets a weight of "0". This is implemented as a Kubernetes annotation on two separate HeatStack CRs, which correspond to the applications "A" and "B". Using an atomic switch functionality implemented with our Heatstack controllers, we can ensure that A and B switch weights via the Traefik configuration in a single transaction, such that Traefik immediately switches over from A to B with zero downtime. Using the Operator framework and a Config Map lock, we can successfully process both weight changes via the Traefik Listener such that it is transactionally atomic, and thus accomplishes the goal of integrating Heatstacks with Traefik on OpenStack, while keeping processing scaling of each Heatstack independent of this functionality.

Canary Deployment

For a canary deployment, a similar technical scheme is used as the blue-green strategy in order to change the proportion of traffic going to services. As opposed to blue-green deployment where we perform an immediate switch-over from one application to another, in this case a new web application is introduced and validated over time with increasing levels of traffic so as to reduce risk of application issues in the event of a blue-green deployment.

As mentioned in Fig. 10, the process of committing weight changes to Git can be either performed through manual Git commits by a developer or Git commits via planned automation. In the case of canary deployments, it is preferrable to use a platform such as ConcourseCI or Argo Workflows to perform the canary deployment changes so that incremental traffic changes can be automatically applied over time without human intervention. This was demonstrated in Fig. 10 with the "Automated Cron-Based CI Triggers", which make it easier for planned changes to be continuously integrated based on a pre-scheduled change.

Scaled-Rollout

Scaling up / down actions can also be combined with load balancer weight changes to perform even more complex and useful arrangements of deployment schemas such as scaled-rollout strategies. As discussed earlier in Section 7, order is enforced in scale up and scale down situations between core and auxiliary components in Fig 8. This augmented functionality enables us to perform a scaled-rollout scenario with zero downtime, similar to how Kubernetes performs this same action for Replica Sets and Deployments.

With many open-source tools available at our disposal that are alternatives to Traefik, there are a whole host of different load balancers that could be very easily supported in similar ways. In fact, the popular open-source load balancers Nginx [54] and HAProxy [55] also provide Kubernetes manifest interfaces that would allow for similar scale up / down functionality, although the only caveat is that this support would require significant investment in Operator development to expand your load balancer option.

9. VoIP Stack POC Deployment using ArgoCD/Argo Workflow

As we called out in the introduction of this paper, most telecom workloads are still deployed on private on-premises cloud and running on virtualization solutions such as OpenStack. In this section, we will introduce a representative VoIP stack that is fully consisting of open-source implementations and mirroring of what a typical telecom provider might have in their network, and explain how to use the proposed CI/CD GitOps architecture to achieve end to end automation.

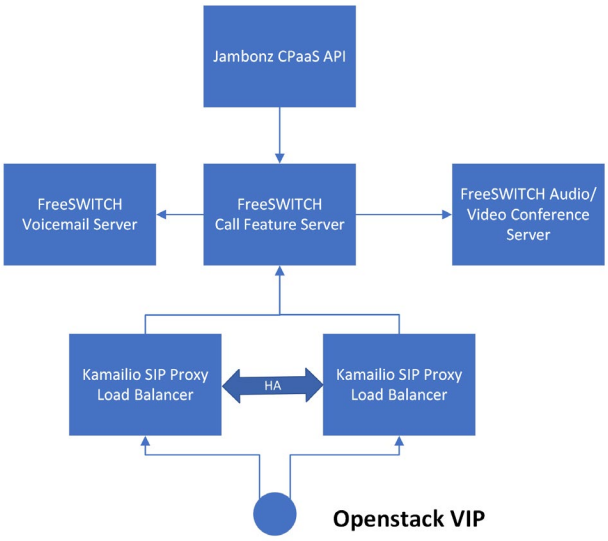


Figure 11 - Freeswitch-Openstack High Level Architecture

This VoIP stack consists of the following components: 1) FreeSWITCH SIP feature server [56]; 2) Kamailio SIP Proxy [57]; 3) Jambonz CPaaS solution [58].

FreeSWITCH is an open-source modular SIP feature server that can be configured in different ways to fulfill roles such as a call feature server, a voicemail server, a multi-party audio/video conference server, a media server, a transcoding SBC, or a WebRTC gateway.

Kamailio SIP Proxy is a very popular open-source SIP load balancer that can be used to front the FreeSWITCH and perform different kind of SIP load balancing. It is often deployed in a HA setup to allow for local redundancy.

Jambonz is an open source CPaaS platform that exposes Webhooks and RESTful APIs layer for invoking communication network capabilities such as call control, digit collection, or voice interaction. Underneath it is utilizing FreeSWITCH with additional modules to integrate with public cloud offering for text to speech, speech to text, or even voice dialog solution such as Google DialogFlow.

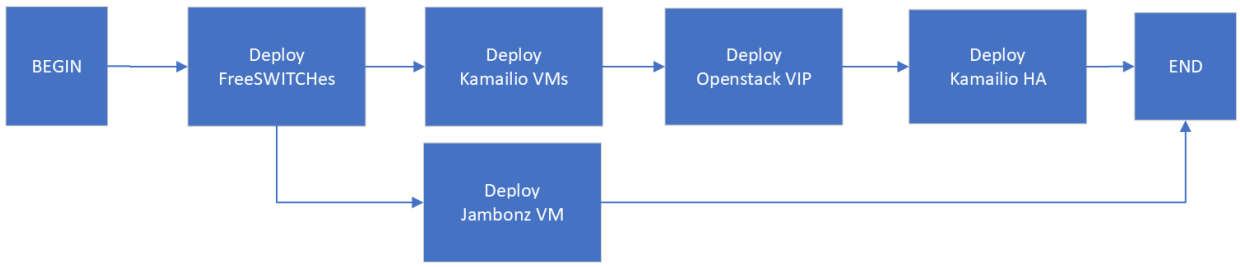


Figure 12 - Freeswitch-Openstack Deployment Process

The above Argo Workflow describes a desired deployment sequence of the VoIP stack and the dependencies between components.

1) The set of FreeSWITCHes will be the first group of components to be deloyed; Those FreeSWITCHes will be created in Openstack using the same FreeSWITCH packer image we generated in the application CI stage but will be instantiated with the proper configuration depending on which role the VM install is going to play, for example, a voicemail server might load a voicemail FreeSWITCH configuration for it to load the required modules and the correct dialplans;

2) Once the IP addresses and SIP ports of FreeSWITCHes are known, we can deploy the two Kamailio SIP Proxy VMs using the Kamailio SIP Proxy packer image, each with the proper configuration to load balance SIP requests to the above FreeSWITCH instances.

3) Once the two Kamailio SIP proxy VM are created, the deployment process will need to acquire a VIP resource from the underlying Openstack infrastructure and modify the Kamailio SIP VM network port configuration so the VIP can be honored by those two network ports. The interaction with Openstack is done through Openstack CLI client running within an Argo Workflow container. This step is required before the next step HA configuration for Kamailio SIP Proxy.

4) With the VIP generated from Step 3, the deployment process can install keepalived on those two Kamailio SIP proxy VMs with the proper keepalived to monitor each other so they form a HA pair. Only one VM will be claiming the ownership of the VIP at a time, the other will only take over when the current one fails to respond to keepalive pings beyond a defined threshold.

5) In parallel to step 2-4, a separate deployment task will be used to kick off the Jambonz CPaaS API VM deployment using a Jambonz Packer image, and instantiated with the proper configuration to point to the FreeSWITCH IPs and Ports.

10. Impact & Caveats of Unified Deployment Strategy

In this paper, we presented two key methods of tackling the "Temporal Synchronicity Problem" in Fig. 1:

1. The Operator Design Pattern
2. The Argo-Workflow Design Pattern

The primary outcome of our efforts with these two patterns was the compression of complexity into manageable abstractions that help simplify the continuous deployment process.

10.1. Operator Design Pattern

With the *Operator Design Pattern*, we solve the "Temporal Synchronicity Problem" in Fig. 1 with independent controllers separated across fault-tolerant pods within Kubernetes. Operator architecture leverages clear separation of concerns as a key aspect of this solution, which was demonstrated in the Traefik Load Balancer example in Section 8. Combined with the GitOps methodology, the Operator Design Pattern also shines in its capability to *continuously* integrate with platform endpoints to ensure that actual state converges with latest state in Git over time. This enables easier handling of complex interactions between components, such as the interaction between OpenStack Infrastructure and DNS allocation logic where one step is dependent on another.

While it is favorable from an engineering standpoint to compress deployment complexity into the Operator Design Pattern to solve the issue in Fig. 1(a), there is admittedly a significant fixed cost in setting up the Operator infrastructure to support a new platform. In addition to this fixed cost, there are some variable costs to maintaining a deployment KADC platform Operator, however our evaluation is that this cost is minimal compared to maintaining a diversity of different DevOps tooling. The general rule of thumb we have discovered in designing and developing Kubernetes Operators for custom needs is that if you need a highly complex and continuously validated API-based integration with a new platform, Operators are probably your best option.

10.2. Argo Workflows Design Pattern

Whereas the Operator design pattern enforces deployment order through sub-patterns such as the resource-listener architecture, the *Argo Workflows Design Pattern* does so via its native "direct acyclic graph" compatibility, or DAG for short. DAGs provide a very powerful way to orchestrate both synchronous and asynchronous actions based on containerized workloads so that order may be easily implemented. Compared to the *Operator Design Pattern*, Argo Workflows does not require nearly as much custom coding and setup, as it is a template-based solution.

As most notably demonstrated with the VoIP Stack deployment in Section 9, Argo Workflows is leveraged in order to deploy various components within the proposed VoIP Stack. Due to the simplicity of this use-case where VM configuration needs to be updated, Argo Workflows shines with a short, containerized script that accomplishes a small set of tasks.

10.3. Resource Savings using Unified Platform Approach

Revisiting all the issues tackled in this paper listed in Fig. 1, the end-goal of solving these problems is to aid engineers within telecom organizations in ultimately saving time and effort in performing complex deployments. Having experimented with both manual and automated approaches in designing the proposed systems outlined in this paper, we can comfortably report those automating deployments with our proposed GitOps-based architecture speeds up our deployments significantly in the two examples we explored:

1. Web-Scale Deployment on OpenStack with Traefik
    a. Without GitOps Automation: 1 Work Day Average
    b. With GitOps Automation: 2 Minutes Average
2. VoIP stack on OpenStack
    a. Without GitOps Automation: 1~2 Week Average
    b. With GitOps Automation: 30 Minutes Average

In addition to the quantitative resource time-savings, we were also able to unify and significantly augment our current deployment capability from a qualitative standpoint. With improvements to the workflow of deployments and increased observability via GitOps Operators such as Argo Workflows and ArgoCD, we demonstrated a straightforward and streamlined method of deploying resources across platforms, while also abstracting away key details of deployment procedures from the deployer.

The OpenStack Operators listed in this paper further allowed us to augment our currently available deployment approaches with advanced methodologies such as blue-green, canary, and scaled-rollout strategies. Our goal in augmenting the OpenStack platform with the Traefik Load Balancer is to demonstrate that advanced capabilities are possible on virtualized platforms and can be reasonably implemented with speed and efficiency in mind. On the other hand, by using Argo Workflows as a multi-stack orchestrator, we demonstrated how resources could further be updated on a scheduled basis to remove critical manual steps from routine deployment situations.

11. Conclusions

One of our industry's most burdensome software-development trends is a diversity of application requirements that will continue to cause major strategic bottlenecks in deploying new types of workloads, while also driving increased long-term costs of sustaining older legacy apps of an assorted variety. In this paper, we have proposed a proof-of-concept solution that seeks to solve these problems by enabling increased platform deployment diversity and velocity by using a single administrative control-plane for both web-scale and VoIP workloads, as well as across different deployment use-cases.

With our proof-of-concept web-scale and VoIP Stack deployment approach for the OpenStack platform, we demonstrate one possible implementation for a variety of common telecom industry-specific scenarios. With a GitOps methodology for deploying OpenStack resources, we established that it is possible to create opinionated deployment abstractions that compress complexity into fault-tolerant Operator-pattern primitives, while allowing for extensibility and reusability of these primitive in an object-oriented manner. With a scheme of custom-resource organization, we implemented recognizable and easily understood constructs with general implementations of design-patterns to deploy compliant infrastructure and software across multiple platforms. By extending this approach into the realm of cloud-native and other more modern types of workloads, it is also easy to imagine adding similar ways to deploy to newer and more experimental, cutting-edge platforms through similar designs and architectures.

While the IT Industry moves toward GitOps as a popular methodology, we believe the key lesson for our telecom organizations is that it may be difficult to onboard complex applications such as VoIP Stacks to most platforms without better forms of deployment orchestration. While GitOps Operators such as ArgoCD and Argo Workflows provide a good starting point for abstracting deployment listeners, most of the DevOps work in our proposal resides with custom Operator development and sustainment for highly complex scenarios, while Argo Workflows shines in simple DAG workflow cases. This methodology provides a basis for future development with Kubernetes Operators or other chosen organizational constructs that are practical for software teams to adopt over time.

The decision to move toward multi-platform deployments will no doubt require careful thought and investment in compressing key implementation details of deployments into manageable abstractions. Within the realm of web-scale and VoIP workloads, it is important to appreciate the complexity of deployment logic, the tools available to solve common problems experienced by DevOps Teams, and the proposed solution's architectural tradeoffs. While the industry continues to move toward increased complexity of newer, more modern and powerful platforms, we should consider from a resource standpoint that managing all these systems can become unduly burdensome and subject to human error. Our hope for future research in this area is that it continues to find increasingly efficient and simplified ways to use GitOps, Kubernetes and similar tools that augment the overall DevOps experience.

12. Acknowledgements

We thank our colleagues who have contributed to our GitOps innovation work: Chris W., Johan C., Pawan T., Jaipal K., Eugene N., Robert S., and Sachin P. We also want to thank the Red Hat Open Innovation lab team for their constructive input and guidance with OperatorSDK. Finally, we want to thank John D., Arvind K., John G. and Brett S. for their continued sponsorship of the GitOps innovation work inside Comcast Communication Engineering organization.

Abbreviations

API	Application Programming Interface
CI/CD	Continuous Integration & Continuous Delivery
CLI	Command Line Interface
CR	Custom Resource
CNCF	Cloud Native Computing Foundation
CRD	Custom Resource Definition
DAG	Directed Acyclic Graph
HA	High Availability
REST	Representational State Transfer
RPC	Remote Procedure Call
RTP	Real-time Transport Protocol
SDLC	Software Development Lifecycle
SIP	Session Initiation Protocol
VoIP	Voice Over IP

Bibliography & References

[1] Kurek, Tytus, "Openstack is Dead? The numbers speak for themselves.", *Ubuntu,* https://ubuntu.com/blog/openstack-is-dead, 3 March, 2022.

[2] Tozzi, Chrisopher, "DevOps Tools: Why We Don't Need More CI/CD Suites", *ITProToday*, https://www.itprotoday.com/devops-and-software-development/devops-tools-why-we-don-t-need-more-cicd-suites, 7 July, 2020.

[3] https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

[4] https://argo-cd.readthedocs.io/en/stable/

[5] https://argoproj.github.io/argo-workflows/

[6] https://opencontainers.org/

[7] https://www.packer.io/

[8] https://github.com/traefik/traefik

[9] Piscaer, Joep, "DevOps tool sprawl: is 'tool tax' just the tip of the iceberg?", *Azmatic*, https://amazic.com/devops-tool-sprawl-is-tool-tax-just-the-tip-of-the-iceberg/, 12 November, 2020.

[10] Andreessen, Marc, "Why Software is Eating the World", *Andreesen Horowitz*, https://a16z.com/2011/08/20/why-software-is-eating-the-world/, 20 August, 2011.

[11] Wood, David, "Metadata Foundations for the Life Cycle Management of Software Systems", *David Wood PhD Thesis*, https://www.researchgate.net/publication/43496613_Metadata_Foundations_for_the_Life_Cycle_Management_of_Software_Systems, December, 2008.

[12] "Digital transformation for 2020 and beyond -- A global telecommunications study", *Ernst & Young*, https://assets.ey.com/content/dam/ey-sites/ey-com/en_gl/topics/tmt/tmt-pdfs/ey-digital-transformation-for-2020-and-beyond.pdf, 19 February, 2019.

[13] "-,- complexity [4]" https://wordpress.org/openverse/image/a346ff48-ea6b-4f5b-80d6-ecd9316c9fe4 by nerovivo is licensed under CC BY-SA 2.0. To view a copy of this license, visit https://creativecommons.org/licenses/by/2.0/?ref=openverse." 6 March, 2007.

[15] "4.29M/s and 4ms latencies" https://wordpress.org/openverse/image/ed1ad5c1-3b0c-4206-bb3d-780d5209b248 by Kai Hendry is licensed under CC BY 2.0. To view a copy of this license, visit https://creativecommons.org/licenses/by/2.0/?ref=openverse." 13 December, 2008.

[16] Pg. 23, "FCC Fact Sheet - Measuring CAF Recipients; Broadband Performance – Order on Reconsideration – WC Docker No. 10-90", *Federal Communications Commission*, 4 October, 2019.

[17] "File:Koldinghus - Old castle in Kolding - Denmark 017.jpg" https://wordpress.org/openverse/image/b7e220a0-4208-467b-be5a-7521f3954d15 by S.Juhl is licensed under CC BY-SA 3.0. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/3.0/?ref=openverse.

[18] "Computer Security - Protect Data - Computers" https://wordpress.org/openverse/image/464bebf1-9575-4018-b8b6-990953dd0cb0/ by perspec_photo88 is licensed under CC BY-SA 2.0. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/2.0/?, 10 June, 2015.

[19] Henriquez, Maria, "2021 Breaks the record for security vulnerabilities", *Security Magazine*, https://www.securitymagazine.com/articles/96668-2021-breaks-the-record-for-security-vulnerabilities, 9 December, 2021.

[20] "New York City - Timelapse on Vimeo by stimul" by Retinafunk is licensed under CC BY-SA 2.0. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/2.0/?ref=openverse.

[21] "La Silla Dawn Kisses the Milky Way" https://wordpress.org/openverse/image/a7619e1e-96e1-4cf1-88d2-e9a95988c2b4/ by European Southern Observatory is licensed under CC BY 2.0. To view a copy of this license, visit https://creativecommons.org/licenses/by/2.0/?ref=openverse, 22 March, 2016.

[22] Baldwin, Carliss, "Design Rules, Volume 2: How Technology Shapes Organizations: Chapter 7 The Value Structure of Technologies, Part 2: Technical and Strategic Bottlenecks as Guides for Action", *SSRN Electronic Journal*, 10.2139/ssrn.3270955, January, 2018.

[23] Baldwin, Carliss Y., Design Rules Volume 2: Chapter 16—Capturing Value by Controlling Bottlenecks in Open Platform Systems. Design Rules Volume 2: How Technology Shapes Organizations, https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3482538, 10.2139/ssrn.3482538, 7 August, 2021.

[24] McKendrick, Joem "The snags holding back DevOps: culture, delivery, and security", *ZDNet*, https://www.zdnet.com/article/devops-is-a-mixed-bag-so-far/ 15 March, 2021.

[25] Baldwin, Carliss Y., Design Rules Volume 2: Chapter 16—Capturing Value by Controlling Bottlenecks in Open Platform Systems. Design Rules Volume 2: How Technology Shapes Organizations, https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3482538, 10.2139/ssrn.3482538, 7 August, 2021.

[26] Pg. 4, "CNCF Survey 2020", *Cloud-Native Computing Foundation,* https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf, 17 November, 2020.

[27] https://github.com/jenkinsci/jenkins

[28] https://github.com/travis-ci/travis-ci

[29] Pg. 12, "CNCF Survey 2020", *Cloud-Native Computing Foundation,* https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf, 17 November, 2020.

[30] Gaibi, Zakir, Jones, Gareth, Pont, Pierrem, and Vaidya, Mihir, "A blueprint for telecom's critical reinvention", *Mckinsey* & Company, https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/a-blueprint-for-telecoms-critical-reinvention, 28 April, 2021.

[31] Pg. 12, "Digital transformation for 2020 and beyond -- A global telecommunications study", *Ernst & Young*, https://assets.ey.com/content/dam/ey-sites/ey-com/en_gl/topics/tmt/tmt-pdfs/ey-digital-transformation-for-2020-and-beyond.pdf, 19 February, 2019.

[32] https://www.gitops.tech/.

[33] Grams, Chris, "3 charts that show how open source developers think", *opensource*, https://opensource.com/article/20/6/open-source-developers-survey#:~:text=The%20vast%20majority%20of%20developers,be%20active%20open%20source%20contributors, 6 June 2020.

[34] Lindberg, Erica, "Global Developer Survey reveals need for more collaborative workflows", *Gitlab Blog*, https://about.gitlab.com/blog/2016/11/02/global-developer-survey-2016/#:~:text=TL%3BDR%3A%20New%20survey%20shows,work%3B%2092%25%20prefer%20Git., 2 November, 2016.

[35] https://www.devsecops.org/

[36] "File:Git-logo.svg. (2022, February 14)", *Wikimedia Commons, the free media repository*. https://commons.wikimedia.org/w/index.php?title=File:Git-logo.svg&oldid=629775061 is licensed under Attribution 3.0 Unported (CC BY 3.0), To view a copy of this license, visit https://creativecommons.org/licenses/by/3.0/, 14 February, 2022.

[37] https://concourse-ci.org/

[38] https://tekton.dev/

[39] https://docs.docker.com/engine/reference/builder/

[40] https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html

[41] https://www.java.com/en/

[42] https://github.com/kubernetes-sigs/kustomize

[43] https://github.com/helm/helm

[44] https://github.com/lua/lua

[45] https://github.com/operator-framework/operator-sdk

[46] https://go.dev/

[47] https://docs.openstack.org/heat/rocky/template_guide/hot_guide.html

[48] https://github.com/vinyldns/vinyldns

[49] https://cert-manager.io/

[50] https://doc.traefik.io/traefik/reference/dynamic-configuration/kubernetes-crd/

[51] Fowler, Martin, "BlueGreenDeployment", *MartinFowler* https://martinfowler.com/bliki/BlueGreenDeployment.html, 1 March, 2010.

[52] Fowler, Martin, "CanaryRelease", *MartinFowler* https://martinfowler.com/bliki/CanaryRelease.html, 25 June, 2014.

[53] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#scaling-a-deployment

[54] https://kubernetes.github.io/ingress-nginx/

[55] https://haproxy-ingress.github.io/

[56] https://github.com/signalwire/freeswitch

[57] https://github.com/kamailio/kamailio

[58] https://github.com/jambonz

