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

2022 Fall Technical Forum
SCTE • CABLELABS • NCTA

Delivering Network Agility and Automated Operations with GitOps

A Technical Paper prepared for SCTE by

David Bainbridge
Senior Director, Software Engineering
Ciena Corporation
dbainbri@ciena.com

Table of Contents

Title **Page Number**

List of Figures

Title **Page Number**

1. Introduction

The term "DevOps" is derived from the combination of the terms "development" and "operations". While there is no universal meaning of DevOps, it is generally accepted that it is a combination of specific practices, culture change, and tools [1] intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality [2]. Most commonly today DevOps is used in the operation of compute and storage resources, but increasingly these same concepts are being applied to connectivity (overlay and underlay network) resources.

The term "GitOps" describes additional practices, culture change, and tools that can be applied to DevOps to enable the state of a system to be version controlled through a persistent repository, such as the Git repository system [3]. As the state is updated in the versioned repository, automated processing pipelines are leveraged to verify, validate, and deploy the changes into normal production, while ensuring high quality. Like DevOps, GitOps is commonly applied to compute and storage resources today, but it is increasingly being applied to connectivity (overlay and underlay network) resources.

This paper describes how we integrated full network capabilities into a GitOps paradigm by first establishing a network model and integrating that model into Kubernetes to produce a capability that reconciles desired network state (intent) to a physical or virtual network. This paper then describes how the defined base network model can be leveraged to compose higher level or abstract intents changing the infrastructure from a "how" configuration model to a "what" configuration model. Finally, the network operator culture changes required to transition an organization to a controlled and stable GitOps infrastructure are discussed along with lessons learned and future work.

2. Kubernetes as an Intent Engine

Kubernetes is commonly used as an intent engine to orchestrate and reconcile the state of workloads onto a cluster of compute nodes. The implementation of Kubernetes provides many capabilities that intent systems require, including best practices for model definition, model control mechanisms, reconciliation capability, and model decomposition. Kubernetes also provides core context capabilities around the intent functions for security, scale, and tooling. This combination of capabilities makes Kubernetes a good platform on which to build a system that provides intent-based network operations. As Kubernetes already provides intent capabilities for non-network domains, building NetOps on Kubernetes means that a single intent can express a desired state across network, compute, and storage. Thus, Kubernetes can provide the backbone for a unified reconciling control plane.

2.1. Device Model

The most specific level of an intent system correlates to a specific target's configuration, in that when a specific configuration parameter is set on a target the operator is essentially specifying their intent for the parameters value. When designing our intent system, we decided to start at this most specific level and considered the models that should be used for and directly map to the targets.

Evaluating the targets of which our network consisted as well as industry standards it was found that a common management paradigm of NETCONF and YANG models was applicable. We considered attempting to create a unique device model and then create adapters to various vendor targets, but instead opted to support the existing YANG models both to simplify the overall system and because historically, in our experience, least common denominator (LCD) or common models tend not to be complete or successful long term. Simply put, models already existed, and we saw no justification to not use them.

2.2. YANG to Kubernetes

Kubernetes operates on a set or resource definitions, or models, defined using open schema, known as the Kubernetes resource model (KRM). The first step in supporting the existing YANG models in Kubernetes was translating the YANG models into the KRM. Extending the KRM in this way is referred to as extending Kubernetes with custom resource definitions (CRD) and involves not only creating the schema model, but also developing a controller to provide the behavior or implementation behind the model.

KRM schema is defined using Open API Schema [5] specification and so we built a tool that produced Kubernetes compatible CRDs from YANG source models. These models could then be imported into Kubernetes and instances of these models could be created.

2.3. Model to Target Association

Leveraging the existing YANG models exposed a situation which was not originally considered, and is not common in existing Kubernetes models. Under a NETCONF/YANG configuration model, multiple, independent models are applied to a single target under control (TUC), where each associated model represents a unique configuration intent. In Kubernetes it is typical that a TUC and its configuration model are a single manifest or model definition. To account for this new associative model, we separated the definition of the TUC and the intent models into separate resource definitions with a reference from the models to the TUC to which they are applied, as depicted in Figure 1. By defining the association from the model to the TUC, the solution provides support for multiple models and allows associated models to be added or removed without having to update the TUC directly. Using Kubernetes's label selector pattern to associated models to TUCs was briefly considered, but discarded as this pattern tends to implement a 1:N relationship, while in a network environment a specific model instance is typically not applied to multiple TUCs, and thus only a 1:1 relationship was required.

It was determined that the model instance that represented a TUC would include the information required to provide connectivity to the target and the models to be applied to the TUC would reference the target via an "annotation" as described by the Kubernetes models [4].

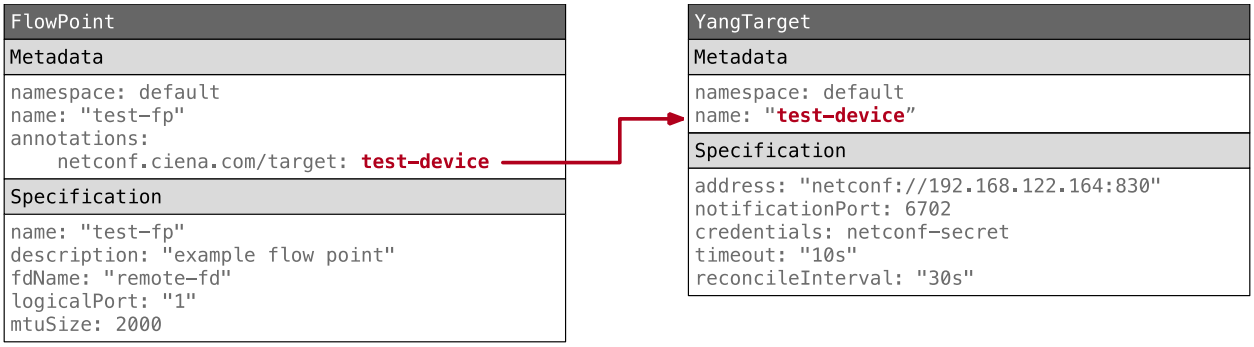

Figure 1 - Example Model to Target Relationship

2.4. Single Operator / Controller

In Kubernetes it is common that a single CRD is controlled by a unique controller that interprets the model changes and realizes them in the TUC, such as creating a Pod based on the Pod resource definition. When leveraging YANG as a CRD, if a unique controller was associated with each YANG model it would mean each model would require a controller and each of these controllers would have near 100% code overlap.

<working>

An additional goal was to allow YANG models to be added into the system without requiring the addition of more code, i.e., a new controller or the modification of an existing controller. This would allow individuals that were not "programmers" to more easily augment the system to support new models. This goal, coupled with the realization that most YANG controllers would have near complete code overlap, drove our decision to build a single controller that could provide the behavior for all YANG models. It was felt that this approach would be more sustainable as well as allow for dynamic model additions. Using this approach meant an individual could execute a tool that would generate a CRD schema from the YANG model and then add the new model into the Kubernetes environment with a simple command line action, as depicted in Figure 2. This capability also could be used in conjunction with the ability to query YANG schema from the TUCs to directly pull and add the models supported by a TUC into the system at runtime.

```
$ yang-to-crd -I /path/to/models ./new-model.yang | kubectl apply -f -
customresourcedefinition.apiextensions.k8s.io/new-model.company.com created
```

Figure 2 - Sample command to add YANG model into system

During implementation, it was discovered that some YANG models had attributes that duplicate names already common to the KRM, such as name. We originally planned to eliminate this repetition to simplify usage of the model, but this led to the need to have custom mappings from existing KRM attributes, such as `name`, to YANG specific attributes, such as `name` or `fdName`. This quickly led to model specific behavior in the operator / controller or external metadata that represented that mapping that could not be automatically generated. Both these scenarios meant a more complex procedure to enable dynamic additions of YANG models at runtime. This path was abandoned, and it was determined that there should be a separation of the KRM object information, represented as Kubernetes metadata, i.e., Name, Namespace, and Labels, from the models which represented the TUC configuration, i.e., the YANG model.

Following this pattern of separation meant that we could provide a generic translation from the specification portion of the CRD to a NETCONF/YANG request, but also meant that there is some repetition between the Kubernetes metadata and the specification. It was determined that this tradeoff was worth being able to provide a generic translation capability, which additionally allows the system to dynamically add new YANG models without recompiling existing controller code, creating a customized controller per YANG model, or accommodating a manually generated mapping via metadata.

During the implementation of the portion of the controller that generates NETCONF/YANG requests from the CRD model, it was discovered that some level of YANG model specifics needed to be included and could not be avoided. To avoid embedding specifics into the operator / controller, patterns were determined and an additional CRD was developed to specify how these patterns applied to the CRD representation of a YANG model. This CRD, named `YangMetadata`, allows for the specifics such as key fields, YANG module information, and model XML namespace information to be specified externally and used to drive the translation of a CRD specified model to a NETCONF/YANG request. An example of a `YangMetadata` instance is depicted in Figure 3. It is important to note that the information contained in the `YangMetadata` resource relates only to the mechanics of translating one model to the other and does not involve mapping KRM attributes to YANG model attributes.

```
YangMetadata
Metadata
namespace: default
name: “fp-metadata"
Specification
reference:
  kind: Fp
  group: netconf.ciena.com
  apiVersion: netconf.ciena.com/v1alpha1
wrapper:
  xmlRequestName: "fps"
  xmlName: "fp"
  keyField: "name"
  module: "ciena-mef-fp.yang"
  moduleSearchPath: "yang"
```

```
YangMetadata
Metadata
namespace: default
name: "interface-metadata"
Specification
reference:
  kind: Interface
  group: netconf.ciena.com
  apiVersion: netconf.ciena.com/v1alpha1
wrapper:
  xmlRequestName: "interfaces"
  xmlName: "interface"
  keyField: "name"
  module: "openconfig-interfaces-ciena.yang"
  moduleSearchPath: "yang"
  augmentor:
    config.type: "xmlns=http://ciena.com/ns/yang/ciena-openconfig-interfaces"
    config.underlayBinding: "xmlns=http://ciena.com/ns/yang/ciena-underlay-binding"
    ipv4: "xmlns=http://ciena.com/ns/yang/ciena-openconfig-if-ip"
```

Figure 3 - YangMetadata Instance Examples

By externalizing the patterns required for special handling of the translations, the reconciling model controller maintains its independence from any specific model and allows models to be dynamically added to the system.

It is possible that there is overlap between the YANG models supported by a single TUC, i.e., two different YANG models can be used to manage a single TUC attribute. While this is acknowledged as a possible issue, in the system being described, the user controls which models are applied to which TUCs and, as such, this overlap can manually be avoided at present. As the implementation progresses some level of precedence may need to be introduced into the model structure to deterministically process model overlap.

3. Reconciling Model Controller

Kubernetes functions as a set of models that are reconciled by a set of controllers that perform operations when model instances or observed states change. The controllers can act on both virtual and physical resources. For example, when an instance of a standard Kubernetes [virtual] resource named `Deployment` is created, it creates another standard Kubernetes [virtual] resource named `ReplicaSet`. The `ReplicaSet` in turn creates `Pod` [virtual] resources, which are then physically realized on a compute capability as set of container instances. If an operator modifies the intent, as specified by the `Deployment` resource, the changes are propagated by the controllers through the `ReplicaSet`, `Pods`, and containers. If the status of the container changes, the status is propagated from the container, through the `Pods`, `ReplicaSet`, and `Deployment`.

As state and status are propagated the controllers can modify the resources that exist. For example, if the operator changes the value of the `replicas` property on the `Deployment,` this changes value `replicas` on the `ReplicaSet`, which in turn determines how may `Pod` instances are created. If a container fails, then this status is propagated to the `Pod` resource where the `Pod`’s status is updated. This status is then propagated to the `RecplicaSet` controller, which in turn may delete the failed `Pod` and create a new `Pod` instance.

This behavioral model has been implemented as part of our solution. When the resource instances that represent a YANG model are applied to a TUC, the solution’s controller converts the model from the CRD representation to a NETCONF/XML edit request to the TUC. The controller is also watching for asynchronous change notifications from the TUC so that if the TUC’s configuration is changed the controller will receive a notification and reconcile the desired state, as represented as the Kubernetes resources, back down to the TUC. This reconciling control loop, depicted in Figure 4, ensures that the system continually maintains the operator’s desired state.

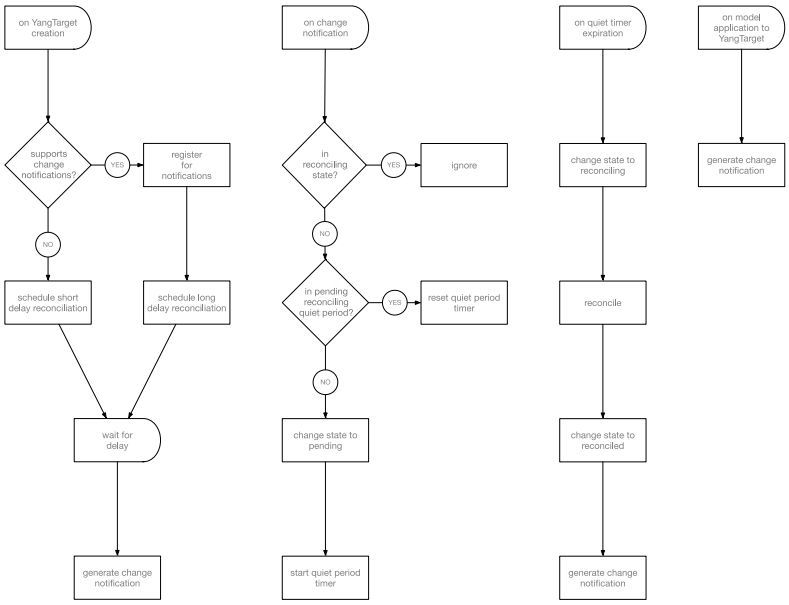

Figure 4 - Reconciliation Behavior

The difference of a TUC's observed or actual state from the desired state is known as drift. When a TUC supports asynchronous change notification the system utilizes that capability to optimize drift detection, using the change notification as an indication of drift. For TUCs that don't support asynchronous change notification, a periodic poll and drift determination is used.

A further optimization is the implementation of a quiet period. Experience demonstrated that changes to a device typically come in rapid succession and thus attempting a reconcile on every change is compute intensive and not efficient. To optimize for this situation, a quiet period is used so that reconciliation will not proceed until no change for a given TUC is detected within this quiet period. This mechanism has the effect of squelching changes to minimize reconciliation without compromising the time it takes to complete the reconciliation.

3.1. Order of Operation and Eventual Consistency

When configuring a TUC using traditional mechanisms or manual manipulation (CLI), order of operations can be important. This is due to dependencies between objects. If object A references B and then A cannot be created until B is created. Management systems, including orchestrators, often are designed, and implemented with knowledge of these order dependencies to ensure the higher-level goals are achieved.

In a model-based reconciliation system, order of operation does not have to be considered by the operator or the implementation. Using the previous example, if an instance representing A is created before an instance representing B, the reconciling of A to the TUC will fail, but will be continually retried. Eventually the reconciling of A to the TUC will succeed because the controller will also be working to reconcile B to the TUC. At some future time, B will exist on the TUC at the time A is reconciliation is retried, thus allowing A to be created.

4. Intent

An intent is the declarative specification of a desired state at some level of abstraction, the "what", with limited or no direction to the implementation on "how" to achieve the desired state. There is a cultural change that an organization must make as they transition to intent-based specifications. This is because today, in general, the "how" is tightly bound to the "what" in many organizations, or at least the relationship between them is commonly known and utilized for daily operations including troubleshooting. The urge to include "guidance" as part of the "what" should be resisted so that the system that realizes the intent can fully optimize across multiple requests and the available resources.

4.1. Abstract to Specific Decomposition

Because an intent is specified at a level of abstraction, it can be decomposed to a set of more specific abstractions recursively until the level of abstraction is roughly the same as the specific configuration values of a TUC.

As an example, walking into a restaurant, asking the server to provide you food, and leaving the selection of the actual meal to the server can be viewed as an intent. The server might translate that level of abstraction to something more specific as the "daily special" when they submit the order to the kitchen. The kitchen staff decomposes the "daily special" into specific dishes and sides, and so on down to specific food elements and their preparation. At each level of intent specification, a set of declarative states can be defined that represents the translation of the more abstract request to one or more increasing specific intents. This same decomposition, in the context of network resources is depicted in Figure 5.

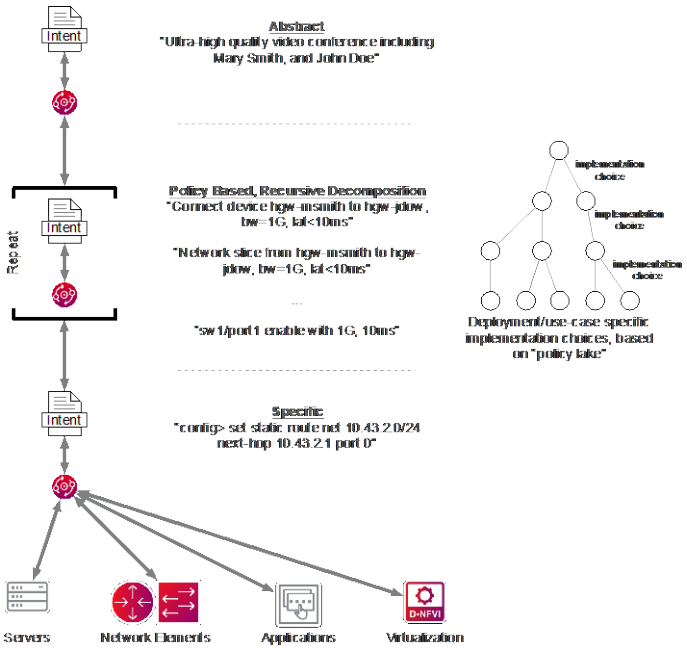

Figure 5 - Intent Decomposition Example

In the system being described, the most specific intent is represented by the YANG models that are applied to a TUC. On top of these intents increasing abstract sets of intents can been established, such as `Fabric`, `Bridge`, and `Link`. For each of these abstract intents there exists a controller that analyzes the intent and generates more specific intents that will realize the desired state.

As abstract intents are decomposed into a set of more specific intents decisions are made by the controller of the abstract intent with respect to how to implement the abstraction, as depicted in Figure 5. Decisions such as which more specific intents should be leveraged, i.e., VLAN or segment routing, as well as which TUCs should be considered.

How a more abstract intent is decomposed includes implementation logic that is outside the bounds of a single, "generic" controller. While a single controller manages all YANG models and their application to YangTargets, the larger system consists of multiple controllers to manage the abstract intents. This abstraction model, is common in Kubernetes for managing resources, as described in the `Deployment` example above and allows additional intents to be added to system at runtime as well as allows implementation up/down grades to existing abstract intents.

While we have experimented with more abstract intents, the bulk of the existing work has focused on the most specific intents, the YANG models. The abstractions we have experimented with, as a proof of concept (POC), express network fabric semantics to which end user hosts as well as intermediate network elements may be included as part of the fabric. While these POC abstractions successfully proved the point of intent decomposition, we have not formally defined a set of more abstract intents for our system

4.2. Intent Relationships

When considering decomposition, it is important to understand the relationship between the more abstract and the less abstract. In its simplest form this relationship could be considered a 1:N, creator to created relationship and is important during several lifecycle stages. For example, if resource A creates resource B, then when resource A is deleted, it is likely that resource B should be deleted. Additionally, if an attempt is made by an operator to directly delete a resource created by another resource (i.e., B), then this operation should fail because the resource that created B (i.e., A) should control the lifecycle of B. In short, when one resource creates another as part of a decomposition it has a responsibility towards the created resource's lifecycle.

Complicating this relationship model is the fact that when dealing with network resources it is possible that more than one higher level abstraction may decompose and influence the configuration of more specific abstraction. So the relationship between an abstraction and its decomposition is not necessarily 1:N, nor a tree dependency relationship, as depicted in Figure 6(A), but could be N:N and a graph, as depicted in Figure 6(B). Thus, the relationship between intents (abstract to specific) might better be described as "interest" rather than ownership or creator to created.

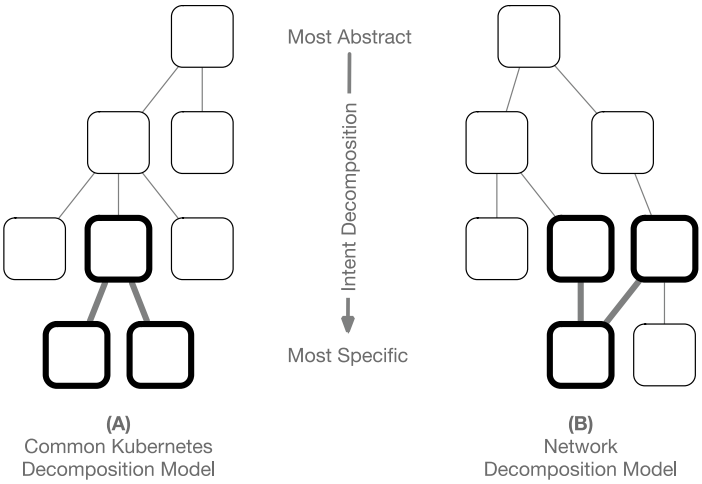

Figure 6 - Abstraction / Intent Decomposition Models

With an interest relationship, the more specific intent cannot be removed until all more abstract intents are removed. At the same time, when a more abstract intent is removed, the more specific intent may be modified. Additionally, a more specific intent can be influenced by changes in more than a single more abstract intent, as such when an abstract intent changes, all "peer" abstract intents must be evaluated to determine the values of the shared, more specific intent.

While the Kubernetes framework does not prohibit the modeling of an acyclic dependency graph, it is not a common usage of the capabilities and may require special considerations when implementing such relationships. This pattern of reconciliation is not currently, to our knowledge, implemented as part of the Kubernetes framework. As this work continues, introducing this relationship will be avoided, if possible, as it significantly increases the complexity of the overall system.

One alternative to allowing an acyclic dependency graph may be to allow each more abstract intent to decompose into its own more specific intent and then predictably and consistently merge those intents when they are being applied to a single TUC.

4.3. Intent Specification and Alternative Tooling

The system being described utilizes YANG models to define the most specific intents as Kubernetes CRDs. Within this system, higher level intents can also be specified as CRDs, but can also be expressed using alternative tools including Helm [6] or Kustomize [7]. The result of these alternative methods is the generation of a set of KRM/CRD instances that represent some abstract intent. When using these alternative tools, there is no Kubernetes intrinsic relationship between the KRM/CRD resources deployed to Kubernetes and originating intent specification, i.e., Helm chart or Kustomize overlay.

As the Kubernetes operating model represents the state of the KRM, it is recommended that CRDs are used to define mature abstractions. We have found that tools such as Helm and Kustomize are valuable when prototyping abstractions, but as the abstraction matures being able to take full advantage or the KRM for implementation is advantageous and allows users to interact with the abstraction using the full ecosystem of Kubernetes based tooling.

It is important to note that while we do not recommend tools such as Helm or Kustomize for the modeling of abstract intents, other organizations may differ in opinion and nothing in the system being described prohibits from taking advantage of the full toolsets available through the Kubernetes communities.

Additionally, these tools can often be integrated into the GitOps processing pipelines, as described in section 7.

4.4. Unified Intents

Because the network intent model is built as CRD extensions to the KRM and it is instrumented via the Kubernetes controller framework, it is possible to create and deploy intents that include compute, network, and storage via a single model and toolchain. This allows infrastructure operators to define intents that are composed of these, and potentially other, resource types to ensure reconciliation across the resource domains, as depicted in Figure 7.

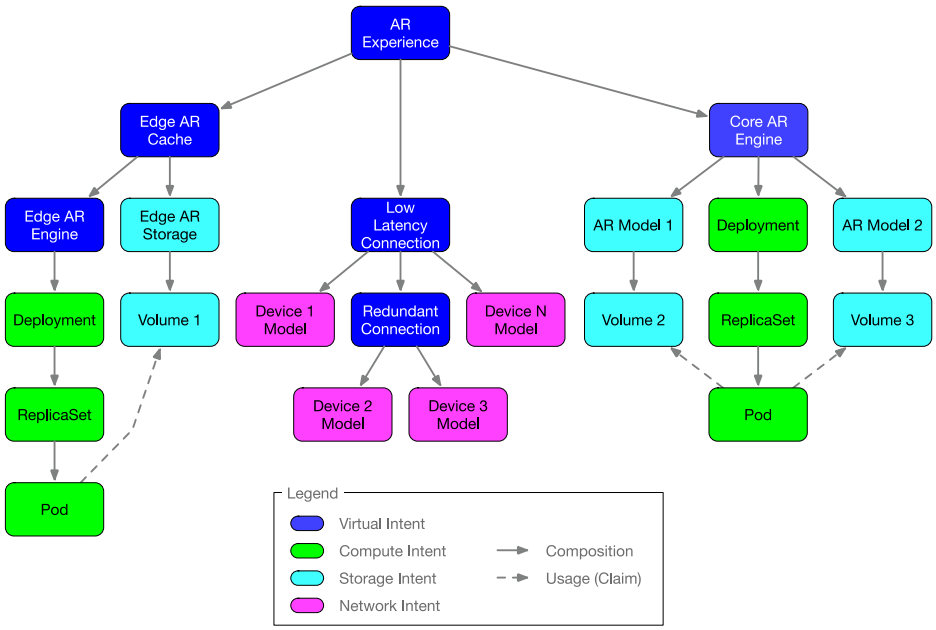

Figure 7 - Multi-Resource Domain Intent Decomposition

Again, while it is recommended that abstract intents be defined as extensions to the KRM as CRDs, this is not required, and they can be defined using many Kubernetes ecosystem tools including Helm and Kustomize. While GitOps was originally designed around the concept of directly deploying KRM resources into a cluster, many GitOps tools chains support tools like Helm and Kustomize directly so that, from the GitOps tool chain perspective, they are first class citizens. Integration of these tools into the GitOps tool chain extends the unification of the resource domains into the GitOps paradigm.

5. Reconciliation

An intent, as previously stated, declares the desired outcome, including capability and performance characteristics. To realize this outcome an intent is decomposed into more and more specific intents until the point where the intent is equivalent to a TUC configuration, including any configuration or modification to telemetry collection that is required to maintain performance characteristics of the intent. However, a TUC does not always maintain the state to which it was set. The difference between and intent's intended states and its actual or observed state is referred to as drift and can be caused by many factors, including errors, failures, or configuration change originating outside standard management practice, either intentional or malicious.

Drift in the system indicates that an intent or desired outcome is likely no longer being met, and thus, the drift must be addressed. When drift is encountered in a reconciling system, the system attempts to correct the drift in several ways, from the least disruptive to the most disruptive and from the most specific intent to the most abstract intent.

5.1. Drift

Within the system there are two types of drift which must be considered: static and constraint. Static drift (SD) is the difference between a desired configuration and the actual configuration. At the most specific intent level this can be the difference between the configuration set on a device and the device's actual configuration. In a "perfect" world SD is not encountered because all system change is driven through the control systems, but, as the world is not perfect, the control system must account for SD and when it is detected reconcile the desired state back to the actual. Reconciliation, in this context, consists of the control system resetting the TUC's configuration to the desired state.

Constraint drift (CD) is when the performance constraints specified by an intent are no longer being met even though there is no SD. CD is detected by telemetry collection and evaluation. When CD is detected the controller for the intent in which drift is detected re-evaluates how the intent can be realized and may create new more specific intents or modify/delete existing intents while attempting to eliminate CD. If a controller cannot eliminate CD, it will update the status of the intent. This status update serves as a trigger/indicator to more abstract intents that they need to be reevaluated to attempt to reconcile the CD. This process is recursive until a more abstract intent can eliminate the CD. If the CD cannot be eliminated then the status of the intents indicates the CD drift and it is up to the operator, or other controlling system to determine how to proceed.

It is important to recognize that CD may be a temporary situation, and that immediately attempting to correct for CD as soon as it is detected may not be the best action, as the effort and time to correct the CD may be more "painful" and take more time than the temporary CD might otherwise exist. For example, if the bandwidth for a connection is desired to be at 500 Mbps and CD is detected at 300 Mbps, the time it takes to reconfigure to a different path to maintain 500 Mbps may be longer than simply waiting for the CD or return to its normal operating state. This is not true in all situations, as in the case of a true failure, but can be true when the cause of the CD is a temporary "blip". Because of this, a waiting period should be observed before any correction of CD is attempted.

5.2. Drift Prediction

Drift prediction (DP) is the process of analyzing desired state, telemetry, and other sources of operation state and status to predict a future time where CD may occur. While our current implementation does not provide DP, it is believed that analysis, including AI/ML analysis, could be leveraged to calculate DP and use these predictions to re-evaluate intents before actual CD occurs, minimizing or eliminating potential issues.

6. Multiple Sites

Even in an organization where the desired state of the infrastructure, compute, network, and storage, is relatively static, the level of communication between the infrastructure TUCs and the reconciling controllers can be significant, particularly when you consider telemetry collection and CD detection. As such, it is prudent to separate an organization into domains each with their own control system. This practice can be seen in the evolution of Kubernetes with respect to workload control. In practice the

controller and the TUC should be “network close”, indicating sufficient bandwidth, connection stability, and reasonable latency.

When architecting multi-site control system, various patterns can be leveraged, such as hierarchical and peering. As our system is leveraging the Kubernetes platform, hierarchical control was a natural fit, where each site is a Kubernetes cluster and the clusters are aggregated via a multi-cluster console, such as provided by various open source and commercial products. A simple aggregation console does not address the issue of cross site intents but does allow administration of each site/cluster directly while providing a global view of a deployment.

Supporting cross site intents implies that an intent can be specified to a system such that the realization of the intent deploys and connects resources from multiple sites, while control of those resources are individually controlled by the site or Kubernetes cluster that “owns” them. To date, in our implementation we have not addressed the issue of cross site intents, but it is envisioned that a centralized Kubernetes cluster could house models and controllers that represent cross-site intents. These models, along with information about the remote clusters, could then be leveraged to implement intent-based configuration of our organization’s global infrastructure resources. As we believe our requirements and goals closely align to those of the larger Kubernetes ecosystem, we continue to monitor projects under organizations like the Cloud Native Computing Foundation (CNCF) and the Linux Foundation (LF) that consider cross-site or multi-site control architectures.

7. Source of Truth and the Pipeline

The operational source or truth (SOT) for the system being described is the model states as maintained by the Kubernetes reconciliation engine. These states can be made redundant, highly available, and persistently stored via standard Kubernetes deployment practices. These states represent the desired state toward which the system is working.

When extending the system beyond the operational state to utilize a GitOps environment, the SOT for the organization is maintained and versioned, in the git repository as a set of files that represent the desired states. In practice these states are represented as model instances formatted as YAML documents. These YAML documents, in Kubernetes parlance, are referred to as manifests. Events within the git SOT, such as the modification and commit of a manifest or the releasing (tagging) of a git repository that may contain many manifests, causes the state from the git repository to be applied to the operational system(s). At steady state, the state maintained in git and the operational state are equivalent. Figure 8 depicts a typical GitOps pipeline, including manual merge approval, automated unit testing, and multiple pre-production deployment environments.

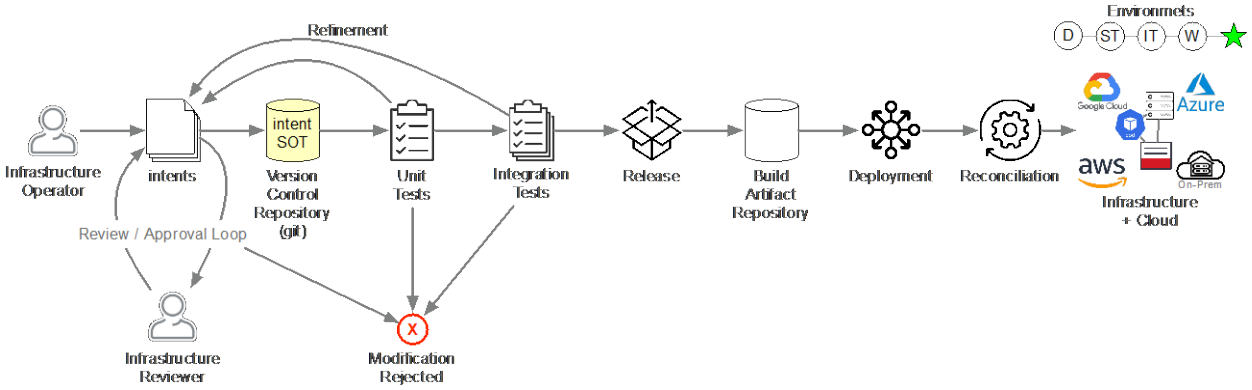

Figure 8 - Representation of GitOps Pipeline

7.1. Test and Digital Twin Environments

As part of GitOps, organizations can establish automated pipelines that verify a change in compute, storage, and network state before it is pushed into production. Minimally, this can include some basic static analysis and regression testing; but it can also include the deployment into a digital twin environment may include full system and failure tests.

The level of testing is up to the organization with the goal of not only protecting the production system from accidental configurations, but also improve the confidence that a new change will not adversely affect the capability or performance of the production system.

7.2. Rollback

Because the state of the environment (compute, storage, and networking) is maintained in a versioned repository and because the environment is controlled by a reconciling controller, it is possible to set the state of the environment back to any previous states by running the GitOps automated pipeline on a previous version of the desired state.

The pipeline will deploy the desired (previous) state to the Kubernetes reconciling control plane and the controllers within the Kubernetes control environment will reconcile the desired (previous) state to the actual state on the TUCs. Thus, rollback becomes the process of identifying the version of the desired state and executing the automated process pipeline, changing rollback from an anomaly in the environment to a normal process that is not to be feared.

7.3. Kubernetes Drift from Git

GitOps solutions do not typically monitor Kubernetes deployments for change or attempt to determine drift between the Git SOT and the operational SOT in Kubernetes. This is because it can be expensive and it is not required because of the Kubernetes reconciliation model.

When a state is applied to a Kubernetes control plane it compares values in the new state to values in the current operational state. If the values are equivalent, then no action is taken, and the application of the state is essentially a no-op. If the values are different, then the associated controllers are invoked, and the updated state is reconciled and propagated throughout the system.

This operational state allows GitOps solutions to simply reapply existing state multiple times with little computational consequence, rendering drift detection between git and Kubernetes unnecessary. Instead,

the GitOps systems re-apply the desired states at a [configurable] interval. Thus, git and Kubernetes are kept synchronized within the time error of this period.

7.4. Lockdown

A key aspect to any controlled environment is to minimize the ability of individuals to introduce unauthorized SD into the system. All changes to the TUCs **must** be handled through the GitOps pipeline, for normal operations as well as in times to failures.

This operational change is another instance of culture change an organization should be willing to accept before moving to GitOps management of their environment. While it may seem counter intuitive to force operators to drive changes through the GitOps pipeline in the presence of a customer affecting failure it is also, precisely at these times when extra care should be taken to remediate the problem to prevent a proposed remediation from exasperating the original issue and consequences.

While exceptions to this rule may be natural to consider, such as when the GitOps pipeline can no longer communicate to the Kubernetes control plane, it is recommended that this communication error between GitOps and Kubernetes be resolved first and then proceed though the GitOps process. One risk of not following this pattern and directly manipulating the configuration of the Kubernetes control plane or the TUCs is that when connectivity is restored, the GitOps pipeline will reconcile its version of the SOT, through the Kubernetes control plane, down to the TUCs, overwriting any direct manual change made and potentially reproduce the original problem.

7.5. Two Sets of Eyes

GitOps evolved from the software development practices related to continuous integration and continuous deployment (CI/CD). One of these practices is related to how code becomes accepted as part of the production code for a project. In this model, a developer develops some code and submits that code to be merged into the production code. Before the code is accepted, or merged, into production product, automated tests are run with the code and an individual with authority must review and approve of the code change. Within some projects, multiple people must review the code and the code is not accepted until a given number those individuals agree and approve of the code change.

With GitOps as part of infrastructure management, this process should be adopted. The equivalent of code when using GitOps for infrastructure management is an intent specification. When an operator makes a change to the desired intent state, this is submitted to the Git repository for automated testing, review, and approval. Only when the change is approved is it merged as part of the production SOT in Git and propagated to the Kubernetes control plane to be deployed in the production environment.

7.6. Cultural Change

GitOps, as previously defined is meant to provide a continuous, controlled, predictable, and stable change process while maintaining the highest level or correctness in the system. Before GitOps can successfully be deployed in an organization the wild west must be sacrificed as must the lone hero. Change process is controlled. Changes are automatically verified and reviewed regardless of the source. Even in the face of customer outages the process is followed. This can be a difficult pattern to follow and enforce when dealing with software development projects as many developers will attest. This can be even more challenging when attempting to apply this to infrastructure management.

8. Conclusions

A reconciliating control plane is an effective control model for network elements and the service across those network elements. When the control plane used for network connectivity can be shared, as is the case with a Kubernetes control plane, the paradigm becomes even more powerful for infrastructure and infrastructure service control.

Adopting the culture and best practices from GitOps adds benefit on top of the core reconciliating control plane and can lead to a more predictable and stable infrastructure change model. With the additions of manual review and automated checks in the GitOps pipeline, as well as system wide pre-deployment test in alternate environments, including digital twins, the incidence of misconfiguration can be minimized.

By introducing abstract intents into the system, the operators are allowed to specify the goals of the system while automated control loops optimize the implementation. It is important with the introduction of intents that the ability to troubleshoot in the context of explicit configuration be supported, including the ability to map from the explicit device configuration back to the intent or intents from which it was derived and why it was derived.

Kubernetes was a good choice for a system as described. It provided much of the "context" required for such a system, the ability to unify control across resource domains (compute, storage, and network), and provided opinionated processes for the definition of models and controllers. This opinionated process allowed most questions about the implementation of the system to be answered based on existing precedent and allowed the development to focus on those aspects that were unique to the solution.

When introducing a system as described into an enterprise it is important that the organization should feel empowered to extend the service intent model in ways that best meet their needs. This includes the development of new or organization specific CRDs that model required services. This is not an "out of the box solution" and an organization should expect and train their employees to be infrastructure model and control developers or rely on 3rd party product and support.

8.1. Future Work

8.1.1. Abstract Intents

As previous stated, the existing implementation has focused on the most specific intents. Work has started on developing more abstract intents. This work needs to continue in cooperation with the organization's IT services groups so that the set of intents instrumented match the infrastructure services offered.

It is expected that as more abstract intents are developed issues may arise with decisions previously made and those decisions may have to be revisited. It is a guiding principle of this work to avoid complexity where possible, even at the cost of operational usability. It is believed that by keeping the underlying technology as simply as possible that the result will be a more stable system. It is also believed that operational usability can be managed through user interaction methods that implemented best practices or common usage patterns.

8.1.2. Troubleshooting

As this system evolves, the team will be investigating how to integrate better troubleshooting, performance monitoring, and predictive techniques. Being able to provide human consumable explanations as to why a system took a specific action or reevaluated an intent can be valuable when

understanding failures as well as it could provide a bases for an automated feedback system to the consumers of the services.

8.1.3. Multi-site Intents

Another area of exploration will be cross-site or multi-site intents and the comparison of a hierarchical solution vs. a peer-based solution. While we will heavily leverage the direction and learnings of the CNCF, there may be something unique with respect to network connectivity and decentralized environments that points us in a different direction. When controlling connectivity, the inter-site connections are as, if not more, important than the intra-site connections. Understanding that, for connectivity, the problem being addressed is not simply distribution of workloads across a decentralized environment. Site inter-connects are imported and must influence how workloads and storage are placed or replicated to optimize the connectivity to meet the desired performance and failure characteristics.

9. Acknowlegements

The author of this paper would like to acknowledge the efforts of Karthick Ramanarayanan and Himani Chawla for their contributions to both the design and implementation of the model-based reconciliation controller. Without their continued hard work this project might not have come to fruition.

The team would also like to show his appreciation to their manager, Marco Naveda, and their peers in the organization for their support, input, and suggestions. While they are not listed as authors on this paper, without their support it would not have been written.

Abbreviations

AI/ML	artificial intelligence/machine learning
CD	Constraint drift
CI/CD	continuous integration/continuous deployment
CLI	command line interface
CNCF	Cloud Native Computing Foundation
CRD	custom resource definition
DevOps	development operations
DP	drift prediction
GitOps	git operations
KRM	Kubernetes resource model
LCD	Least common denominator
LF	Linux Foundation
Mbps	megabits per second
NETCONF	network configuration
NetOps	network operations
POC	proof of concept
SD	static drift
SOT	source of truth
TUC	target under control
XML	extensible markup language
YAML	yet another markup language
YANG	yet another next generation

Bibliography & References

[1] https://en.wikipedia.org/wiki/DevOps

[2] Bass, Len; Weber, Ingo; Zhu, Liming (2015). DevOps: A Software Architect's Perspective. ISBN 978-0134049847.

[3] https://en.wikipedia.org/wiki/DevOps#GitOps

[4] https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/

[5] https://github.com/OAI/OpenAPI-Specification

[6] https://helm.sh/docs/topics/charts/

[7] https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization/

