Utilizing Kubernetes as a universal control plane

Dag Bjerre Andersen

15-12-2022

daga@itu.dk

# 1. Abstract

<!--

Infrastructure management is in constant development.

Cloud Computing brings complete new abstraction layers for interacting with cloud resources.

Since the introduction of Kubernetes a new paradigm have emerged: Control Planes. Kubenretes is built on the control planes and the reconsiler pattern, and works as a echo system for control planes.

Terraform

Control Planes

... missing

A testing application will be described and used to demonstrate the unversial contol planes capabilities.

This explains the concept of control planes and how to build kubernetes as an universal control plane for handling internal resources (e.g. \_pods\_) and external resources (e.g. databases) outside Kubernetes.

<!-- what is control planes -->

<!-- The problem -->

<!-- demonstration app -->

<!-- the findings -->

<!-- further research

-->

Table of Contents

[1. Abstract 1](#_Toc121040944)

[3. Dictionary and abbreviations 3](#_Toc121040945)

[4. Introduction 3](#_Toc121040946)

[5. Motivation 4](#_Toc121040947)

[6. Defining Control Plane 5](#_Toc121040948)

[7. Declarative vs Imperative 5](#_Toc121040949)

[7.1. Terraform 6](#_Toc121040950)

[7.2. From Terraform to control planes 8](#_Toc121040951)

[8. Demonstration Application 9](#_Toc121040952)

[9. Implementation 10](#_Toc121040953)

[9.1. Technologies and tools used 11](#_Toc121040954)

[9.2. ArgoCD and Crossplane together 12](#_Toc121040955)

[9.3. Managing internal state with ArgoCD 13](#_Toc121040956)

[9.4. Managing External State with Crossplane 16](#_Toc121040957)

[9.5. Distributing Secrets 17](#_Toc121040958)

[10. Demonstration Application running in Google Cloud 19](#_Toc121040959)

[10.1. Use in practice 21](#_Toc121040960)

[11. Discussion and evaluation of the implementation 25](#_Toc121040961)

[11.1. Build pipelines 25](#_Toc121040962)

[11.2. Additional costs 26](#_Toc121040963)

[11.3. No preview with Crossplane 26](#_Toc121040964)

[11.4. Single interface 27](#_Toc121040965)

[11.5. Platform Engineering 27](#_Toc121040966)

[11.6. Pets to Cattle: Denne title passer bedre til eliminating state. 28](#_Toc121040967)

[11.7. Declaring the entire infrastructure 28](#_Toc121040968)

[11.8. Bootstrapping Problem 29](#_Toc121040969)

[11.9. Multiple core-clusters 29](#_Toc121040970)

[11.10. Maturity level 30](#_Toc121040971)

[11.11. Comparing it to Terraform 31](#_Toc121040972)

[11.12. Stateless Infrastructure 31](#_Toc121040973)

[12. Conclusion 31](#_Toc121040974)

[13. References 33](#_Toc121040975)

# 3. Dictionary and abbreviations

It is assumed that the reader is familiar with Kubernetes, its concepts, and its terminology. Furthermore, it assumed that the reader is familiar with Terraform, how it manages its state, and how Terraform providers work. In addition, there is a list of the additional terms and aberrations used in this paper.

* **Infrastructure as Code (IaC)**: IaC is the concept of managing and provisioning of infrastructure through code instead of through manual processes[[1]](#footnote-1)
* **GitOps**: GitOps is an operational framework that takes DevOps best practices used for application development, such as version control, collaboration, compliance, and CI/CD, and applies them to infrastructure automation. GitOps is a branch of DevOps that focuses on using git repositories to manage infrastructure and application code deployments. The Git repository is the source of truth for the application configuration and code[[2]](#footnote-2)
* **Cloud Native**: The term cloud native refers to the concept of building and running applications to take advantage of the distributed computing offered by the cloud delivery model. Cloud native apps are designed and built to exploit the scale, elasticity, resiliency, and flexibility the cloud provides[[3]](#footnote-3)
* **Cloud Native Computing Foundation (CNCF)**: CNCF is the open source, vendor-neutral hub of cloud native computing, hosting projects like Kubernetes and Prometheus to make cloud native universal and sustainable[[4]](#footnote-4)
* **Amazon Web Services (AWS)**: Cloud computing platform provided by Amazon.
* **Google Cloud Platform (GCP)**: Cloud computing platform provided by Google.
* **Google Kubernetes Engine (GKE)**: Google scalable and automated Kubernetes service[[5]](#footnote-5)
* **State (in relation to Kubernetes)**: Refers to the collection of Kubernetes objects declared in manifests and stored in Kubernetes' etcd-database. The state both contains the *actual* state of the system and the *desired* (declared) state of the system[[6]](#footnote-6). One state exists per Kubernetes cluster.
* **Internal State**: Internal state in this paper refers to declared Kubernetes Objects reflecting objects managed inside Kubernetes (e.g., *Pods*, *Service*, and *Ingress*)
* **External State**: External state in this paper refers to declared Kubernetes Objects reflecting objects managed outside Kubernetes (e.g., a database running on a cloud provider)
* **Reconciler pattern**: Reconciler pattern is the concept of checking the difference between the desired state and the actual state, and if there is a difference, an action is taken to make the actual state becomes the desired state[[7]](#footnote-7).

# 4. Introduction

Since the creation of Kubernetes, we are seeing an increasing number of CNCF projects built as control planes (FluxCD, ArgoCD, Crossplane, etc.). The concept of a control plane is built on the idea of a service that watches a declared state and make sure that a system's actual state reflects the declared state. If the desired state changes, the control plane ensures that the newly declared state is automatically reflected in the actual state.

At its core, Kubernetes stores a declared state in its etcd-database, and different services (control plane components) watch this state and make sure the actual state reflects the declared state. Kubernetes can be used as a platform for control planes because Kubernetes' API server can be queried to manipulate the state of its declared Kubernetes Objects. Control planes can call the API server and manage the life cycles of its custom Kubernetes Objects (CRD[[8]](#footnote-8)). Control planes in Kubernetes have proven to be good at managing internal state (e.g., scheduling and deploying containers), but during the last years, new control planes like Crossplane that manage the state of cloud resources (e.g., databases and Kubernetes clusters) have emerged. This means control planes like Crossplane can be used to replace infrastructure tools like Terraform by controlling the cloud resources' life cycle from inside Kubernetes.

Now the question is if multiple control planes can be combined to create a universal control plane for handling all kinds of state, all managed from within Kubernetes. This paper will implement a universal control plane for handling both internal state (e.g., containers) and external state (e.g., cloud resources) and discuss the challenges and implications.

# 5. Motivation

This paper is written in collaboration with Eficode. Eficode is a consulting company that specializes in DevOps. Eficode offers consultancy, training, and managed services that enable organizations to develop quality software faster, using the leading methods and tools available. Eficode is committed to keeping up with the latest trends and technologies in DevOps, so they can give their customers the best advice and training[[9]](#footnote-9)

Eficode has stated their concern about how Terraform does not scale well for big organizations in their blogpost: [Outgrowing Terraform and adopting control planes][[10]](#footnote-10). Terraform has proven to be a stable and reliable infrastructure tool for many years now, but it may not always be the best solution. New technologies get showcased, and new paradigms emerge.

Control planes are a new paradigm in DevOps and infrastructure management[[11]](#footnote-11). Many of the technologies/tools leveraging the concept of control planes are still new and do not have many years of proven use. Even though such tools/systems can look promising, it can be difficult to justify the investment in transitioning a DevOps infrastructure to this new paradigm.

In order for Eficode to recommend customers to transition infrastructure managed by control planes instead of tools like Terraform, it is essential to know the implications of such changes and what kind of challenges such a change might bring.

The design and implementation presented in this paper are created and written by myself, and Eficode has not been part of it. The prototype presented and discussed in this paper allows Eficode to get a better understanding of the implications of transitioning to control-plane-based infrastructure-management.

<!-- arguments for control planes / Crossplane ------------------------------------------------------

"the very essence of this new paradigm called ‘Control planes’ and which is poised to replace more traditional Infrastructure-as-Code tools like Terraform." [[source](https://www.eficode.com/blog/outgrowing-terraform-and-adopting-control-planes)]

\_Michael Vittrup Larsen\_ from Eficode puts it like this:

> "Using control planes means relinquishing control. Therefore seasoned SRE teams may feel a reluctance in trusting the control plane. While we still need to see how, e.g., Crossplane fairs in the heat of the battle, we will eventually adopt control planes also for cloud. Not being distracted by details and instead abstracting them away and letting the ‘machines’ handle the details is a natural evolution in tech.

>

> "We have seen this all the way from operating systems, compilers, and container platforms like Kubernetes. Knowing that this is the inevitable evolution, we should embrace control planes. If Crossplane does not strike the right balance and abstraction level, the next control plane will." [[source](https://www.eficode.com/blog/outgrowing-terraform-and-adopting-control-planes)]

Control planes are self-healing, and they automatically correct drift.[[source](https://crossplane.io/)]

Consumers can self-service fast because control planes offer a single point of control for policy and permissions, and control planes integrate easily with other systems because they expose an API, not just a command-line." [[source](https://crossplane.io/)]

-->

# 6. Defining Control Plane

Control plane means something different depending on the context.

* **Control plane (as a concept/paradigm)** refers to the idea of services that watches a declared state and make sure that a system's actual state reflects the declared state. A control plane reconciles the current state to match the desired state. Control planes follow the Reconciler pattern. Control planes are self-healing, and they automatically correct drift[[12]](#footnote-12)
* **Control plane (as a Kubernetes Controller)** refers to an application/service implemented as a control plane (the concept). An example of this is Kubernetes' kube-scheduler (that schedules pods/containers based on a declared state)[[13]](#footnote-13) or ArgoCD (a GitOps tool introduced in this paper).
* **Control plane (as a node)** refers to a node where all Kubernetes default control plane components run (e.g., the kube-scheduler that schedules pods). Kubernetes' default control plane components can be run on any node in the cluster, but they often run on the same node, and user containers do not run on this node[[14]](#footnote-14). This node is often referred to as the control plane (node).
* **Control plane**\*\*: Control planes is based on the Reconciler pattern. A control plane is a service that continuously checks the difference between the desired state and the current state, and if there is a deviation, it tries to take action to make the current state reflect the desired state. -->

When I refer to building a system that utilizes Kubernetes as a *universal control plane*, I refer to a control plane as a *concept*. I think about Kubernetes as an platform for hosting multiple control planes (as applications) that manages all kinds of state (both external and internal state). The universal control plane stores a single state (stored in Kubernetes’ etcd-database) where multiple implementations of control planes reconcile the current state to match the desired state.

<!--

In this paper, I build an implementation of a universal control plane in Kubernetes, where the user can manage both internal state (e.g., *Pods*) and external state (e.g., databases and other Kubernetes clusters).

-->

# 7. Declarative vs Imperative

<!--

When building a universal control plane for handling all our infrastructure, we need to base it on some core design ideas. First of all, we need to want to build our infrastructure as IaC using as much declarative configuration as possible. We want to limit the number of imperative commands and long scripts of sequential steps as much as possible.-

Declarative and imperative are two different DevOps paradigms [[s]](https://ubuntu.com/blog/declarative-vs-imperative-devops-done-right). In the declarative paradigm, developers focus on *what* the desired state of the system should be. In the imperative paradigm, the developer focuses *how* a desired state is created.

-->

An automation system (like the universal control plane) can be designed and implemented in two different paradigms: declarative and imperative[[15]](#footnote-15). In the declarative paradigm, developers focus on *what* the desired state of the system should be. In the imperative paradigm, the developer focus on *how* the desired state is created.

<!-- imperative setup scales poorly -->

The imperative paradigm scales poorly in large software environments. "While using an imperative paradigm, the developer is responsible for defining exact steps which are necessary to achieve the end goal, such as instructions for software installation, configuration, and database creation"[[16]](#footnote-16). The developer has to carefully plan every step and the sequence in which they are executed. Suppose a change in the setup has to be made. In that case, the developer has to carefully make sure the change doesn't break something else in the sequence of steps - especially if there are conditional statements, meaning there are multiple possible paths through the sequence of steps. "Although suitable for small deployments, imperative DevOps does not scale and fails while deploying big software environments, such as OpenStack"[[17]](#footnote-17)

<!-- Den her block er måske useless -->

<!-- declarative configuration is a higher abstraction -->

Creating a declarative configuration is a higher abstraction than declaring a configuration with sequential imperative steps. Every declarative API encapsulates an imperative API underneath. For a declarative API to work, there needs to be some code behind the scenes that parses the files and acts upon them in an imperative way. Declarative programming cannot stand alone because there will always be a sequence of imperative steps executing some operations on a machine[[18]](#footnote-18) [[19]](#footnote-19). Even though creating a declarative configuration is often time more demanding than creating an imperative configuration, we see it all across software development, from CSS in web development[[20]](#footnote-20) to Terraform in infrastructure management.

## 7.1. Terraform

A very popular tool that is based on the idea of IaC and declarative configuration is Terraform. Terraforms' popularity started in 2016-2017 and has been growing ever since[[21]](#footnote-21).

Terraform lets you define both cloud and on-premises resources in configuration files that you can version, reuse, and share[[22]](#footnote-22).

One of the main use cases is often provisioning infrastructure on cloud providers[[23]](#footnote-23).

Terraform is responsible for handling the entire lifecycle of the resources: from creation to deletion[[24]](#footnote-24).

Terraform is cloud-agnostic and can provision cloud resources across all the big cloud providers (e.g., AWS, Azure, and GCP)[[25]](#footnote-25).

Terraform has the concept of terraform-providers, where service providers (e.g., GCP) can create integrations with Terraform and let the user manage the providers' services through the HashiCorp Configuration Language (HCL). "*Providers enable Terraform to work with virtually any platform or service with an accessible API*"[[26]](#footnote-26).

<!-- maybe here

Terraform has the concept of \_providers\_.

A Terraform provider is a Terraform plugin that allows users to manage an external API. "Terraform uses providers to provision resources, which describe one or more infrastructure objects like virtual networks and compute instances. Each provider on the Terraform Registry has documentation detailing available resources and their configuration options."[[27]](#footnote-27).

-->

Even though Terraform is popular and versatile, it might not be the best way to manage infrastructure. This section describes some of the issues related to using Terraform and why it might be better to use control planes for handling cloud resources.

### 7.1.1. Barrier of entry

For teams working with Terraform storing the state in some shared place is a must to ensure all team members can access the state[[28]](#footnote-28) [[29]](#footnote-29) [[30]](#footnote-30). This is commonly done on a cloud provider in some kind of *Object storage*. To store the state on a cloud provider, the developer first needs to set up an account on the cloud provider, gain the required roles/permissions and then write the Terraform code. Depending on the cloud provider, this can be a more or less complex process, and it can be a big hurdle to overcome if it is the developer's first time doing it or the developer is new in the field[[31]](#footnote-31).

So, before the developer can start using Terraform, they first need to solve the problem of *how and where to store the Terraform state*.

Lowering a barrier to start up projects by removing the need for storing a terraform state is an argument for switching to control planes.

### 7.1.2. Challenges with managing terraform state

Terraform state is inherently difficult to manage[[32]](#footnote-32). Just because Terraform state is stored in a remote place does not mean multiple people can work on it simultaneously.

<!-- no lock -->

When storing the state in a remote place, you need to specify a so-called Backend Configuration[[33]](#footnote-33), but not all Backend Configurations support locking. This means that in some cases, race conditions can still happen (if two people run terraform apply simultaneously).

An example of this is that there is no lock on the state if stored on AWS S3 (object storage on AWS). A solution to this is to create a lock for the S3 and store it somewhere else (e.g., an AWS DynamoDB table)[[34]](#footnote-34) [[35]](#footnote-35) [[36]](#footnote-36). This again just adds to the complexity and creates an even bigger barrier for getting started with Terraform if you want to make sure your IaC configuration is safe to use.

<!-- force unlocking -->

Even if there is a lock on the Terraform State, it can still get corrupted. If a terraform apply goes wrong because the process is interrupted for whatever reason, the state can end up not being unlocked, and you have to force-unlock the state[[37]](#footnote-37). This gets even worse if a force-unlock is executed while another process is in the middle of applying[[38]](#footnote-38). This can result in multiple simultaneous writers, which can result in the state being unusable/corrupted[[39]](#footnote-39).

<!-- only one person at a time -->

Furthermore, even though there may be a lock that makes sure that there is no race condition while applying, only one person/process can work on the state simultaneously. Updating a Terraform state can take minutes - e.g., it will take around 10 minutes to provision a GKE cluster on GCP[[40]](#footnote-40). During this time, no other entity apply changes to the configuration[[41]](#footnote-41). This is an even bigger problem when you have a monolithic infrastructure configuration with a lot of dependencies, and all components are stored in the same Terraform state. So, if one developer is updating the GKE cluster, then another developer (or automated process) may be blocked from updating a database or a networking rule. So overall, Terraform can end up being a bottleneck if a big company has multiple developers or processes working on the infrastructure at the same time.

<!-- drift -->

Another challenge with Terraform is that Terraform's state can easily go out of sync. This is called *configuration drift* [[42]](#footnote-42) [[43]](#footnote-43). If the terraform apply-command is not run regularly, the actual state can drift away from the declared/desired state. This can, for example, happen if a Terraform-managed database running on a cloud provider is modified manually through the cloud platforms interface and not through Terraform, then the actual state will no longer reflect the terraform state. This means the terraform state no longer reflects the real world. This can create issues when later mutating the stored state, which can make the stored state unusable because it is so far from the actual state.

In some cases, it might be beneficial to modify the terraform-created-resource manually through some other service if an infrastructure-related emergency happens. An example of this would be a person changing a network rule manually through Google Cloud Platform’s web-interface in the middle because a service needed to use a new IP address, and it needed to be fixed as soon as possible. This would not be possible if the IoC configuration tool did not allow *configuration drift*.

## 7.2. From Terraform to control planes

<!--

So the Terraform state can either be updated by:

- A manual task, e.g., a developer manually creating a database

- A triggered automated task, e.g. a deployment pipeline that applies the newly changed Terraform files a tool like [Atlantis](https://www.runatlantis.io/) or Terraform Cloud.

- a GitOps tool e.g. ArgoCD that continuously synchronizes the actual state with the declared state stored in a repo (or elsewhere).

-->

Configuration drift with terraform can be avoided by using an automated tool (e.g. Atlantis[[44]](#footnote-44) or Terraform Cloud) or a script that simply runs terraform apply on a regular basis. Doing this essentially creates a system that works just like a control plane. So instead of using a tool like Terraform with all its challenges and then patching some of the issues by wrapping it in some automation tool/script then, it may be better to use a control plane-based tool that was built to solve exactly that.

Kubernetes stores a desired state, and the internal components try to keep the actual state as close as possible to the desired state <maybe use reconcile here>. The state is stored as Kubernetes objects definitions declared in YAML (also known as manifests). The information stored in the Terraform state can instead be stored as Kubernetes Objects stored inside Kubernetes. A control plane running on Kubernetes could automatically sync the actual infrastructure with the declared state.

<!-- Control plane-based infrastructure management can automatically correct drift and can "self-heal" if something goes down [[s]](https://crossplane.io/), meaning the -->

*Crossplane*, Google's *Config Connector*, and AWS' *Controllers for Kubernetes* are control planes that reconciles a declared state (stored inside Kubernetes’ etcd-database) with resources managed by a given cloud provider. This paper will focus on Crossplane because it is built as a framework for control planes in general and not only focuses on a single cloud provider.

<!-- skal det her ned til implemenation eller diskussions afsnitttet? -->

Even though the paper highlights Crossplane as a tool, the question is not so much if Crossplane specifically is a great tool or not - but more about whether the paradigm of control planes is good in general. As Eficode states: "If Crossplane does not strike the right balance and abstraction level, the next control plane will."[[45]](#footnote-45)

<!-- Crossplane will be described in greater detail in the \_Managing External State\_-section. -->

# 8. Demonstration Application

In order to verify and demonstrate that the universal control plane I have built (presented in the *Implementation*-section) actually works, I need a demonstration application that runs on infrastructure managed by the universal control plane to demonstrate its capabilities.

To do this, I have used Eficode's public quotes-flask[[46]](#footnote-46) application that they use for educational purposes. It is a simple application consisting of a frontend, a backend, and a database. I will refer to this application as "Quote App".

Quote App's frontend is a website where users can post and read "quotes" from other users. The quotes are posted and sent to the backend-service, which then stores the data in a Postgres database.

The Quote App is built to be run on Kubernetes, and the repository already contains Kubernetes manifests. The system uses a Postgres database running in a standalone *Pod*. To showcase the implementation of a universal control plane's ability to provision database resources on cloud providers, I have replaced the Postgres-database-pod with a managed database running in a cloud provider. Besides that, I have not changed the overall architecture.

A picture containing first-aid kit, soccer

Description automatically generated

Figure 1

This setup is supposed to represent an actual production-ready application that a hypothetical business may want to run on a cloud provider.

The business may want multiple environments like *production*, *staging*, and *development*, and they may leverage cloud services across multiple cloud providers. Therefore, this demonstration application will run on multiple isolated environments on one cloud provider and access a database on another cloud provider (visualized in Figure X).

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure 2: A visualization of the Quote App (demonstration application) running in multiple environments.

This architecture demonstrates the universal control plane's multi-cloud and multi-environment capabilities. The evaluation of this project will partially be based on how well the implemented universal control plane manages to host/deploy/run this demonstration application and what implications and challenges it may result in.

# 9. Implementation

<!-- what will be in this Implementation-section? -->

This section describes how I suggest building a universal control plane within Kubernetes for handling internal and external resources. The implementation strives to imitate a production-ready system for a hypothetical company with a website running in a production and staging environment in the cloud.

<!-- main idea -->

The main design idea of this implementation is to have a single cluster that works as a control plane for managing databases, other clusters, and software deployment. To better understand the design idea, two names are introduced: Core Cluster and App Cluster. The Core Cluster represents the universal control plane for managing both infrastructure and software deployment. The App Clusters is a shared term for all the clusters where business logic is supposed to run. For instance, a company may have two App Clusters in the form of a production cluster and a staging cluster. The Core Cluster hosts all the core infrastructure components (like ArgoCD and Crossplane, introduced in the next section) and shared services between different App Cluster environments.

Graphical user interface

Description automatically generated with low confidence

Figure 3: An illustration of how the infrastructure team manages App Clusters from the Core Cluster.

Only the infrastructure teams are supposed to interact with Core Cluster directly - while the application developers are supposed to only care about getting their workload running on the App Clusters. The design is visualized in Figure X.

## 9.1. Technologies and tools used

#### Amazon Web Services

AWS was chosen as the cloud provider for a simple managed cloud database. AWS is one of the officially supported providers for Crossplane[[47]](#footnote-47). Both Azure and GCP would be potential alternatives to AWS in this implementation.

#### Google Cloud Platform

GCP was chosen as the main provider for cloud-hosted Kubernetes clusters and Networking because GCP is less complex[[48]](#footnote-48) to use than AWS. GCP is one of the officially supported providers for Crossplane[[49]](#footnote-49).

<!-- dics? -->

Another choice would be DigitalOcean, but I experienced some issues with using Crossplane with DigitalOcean (also described in the *Maturity level*-section). I didn't experiment with Microsoft Azure since I had no prior experience working with them as a cloud provider.

Both AWS and Azure would be potential alternatives to GCP in this implementation.

#### Kind

Kind is used for running the Core Cluster locally. When developing and experimenting with the Core Cluster, it can be beneficial to run the cluster locally because it can take a long time to provision clusters on cloud providers (e.g., it often takes 10 minutes on GCP).

<!-- dics? -->

There are many different tools for running Kubernetes locally, and many of them would probably work for this implementation, but the choice ended with Kind because it is easy to set up and simple to use. Other alternatives could be *MicroK8s* or *Docker Desktop*.

#### Gum

Gum is a simple command line tool for making interactive scripts. Gum is used to run the scripts starting the Core Cluster and interactively picking a configuration (e.g., if it should run on GCP or locally with Kind)

#### Helm and Kustomize

Helm and Kustomize are used to template and install Kubernetes resources. Helm is a package manager for Kubernetes, and ArgoCD installs software packages (like Crossplane) using Helm. Kustomize is used for handling templating of my own Kubernetes manifests. One could choose not to use Kustomize and instead put everything into Helm charts as an alternative to this implementation.

#### ArgoCD

ArgoCD is a declarative, GitOps continuous delivery tool that is built as a control plane and runs inside Kubernetes[[50]](#footnote-50). ArgoCD groups manifest into an abstraction called *Applications*. An ArgoCD Application is a Kubernetes object that contains a path to a resource that needs to be deployed, a destination cluster, and some configuration parameters. Applications can deploy raw manifests, a Kustomize manifest, and a Helm Chart. Each Application can be synced independently, and the developer can declare a custom sync-policy for each (e.g., if an application should be synced automatically or not). An example of this is that the Quote App is deployed and managed by an ArgoCD Application. When changes are made to the application configuration in Git, Argo CD will compare it with the configurations of the running application to bring the desired and actual state into sync[[51]](#footnote-51). ArgoCD is used for this implementation because it has more advanced UI features compared to similar tools. ArgoCD was accepted to CNCF on April 7, 2020 and is at the Incubating project maturity level[[52]](#footnote-52).

<!-- dics? -->

The two biggest GitOps-tools are FluxCD and ArgoCD. This implementation could also be built using FluxCD. FluxCD and ArgoCD cover most of the same features, but the way you structure code looks a bit different. Both tools would be good candidates for building a universal control plane for Kubernetes.

Crossplane

Crossplane is a control plane that runs inside Kubernetes that makes sure that the external resources running in the cloud provider are in sync with the state declared in Kubernetes. Crossplane manages the entire lifecycle of the resources declared. All resource managed by Crossplane is declared in manifests stored in Kubernetes. Crossplane was accepted to CNCF on June 23, 2020, and is at the Incubating project maturity level[[53]](#footnote-53)

<!-- providers -->

Just like Terraform, Crossplane has the concept of *providers*[[54]](#footnote-54). Crossplane-providers work similarly to how Terraform-providers work. Service providers can create a plugin that integrates with Crossplane providing the user the ability to provision external resources on their infrastructure. It is now up to the service provider to manage and ensure that the state running on their infrastructure matches the desired state declared in the Kubernetes cluster.

Using Crossplane for infrastructure management makes it possible to provision resources on multiple cloud providers at once, which can be beneficial because different cloud providers have different offerings. Currently, Crossplane supports AWS, GCP, and Azure as cloud providers. A DigitalOcean provider is also in active development[[55]](#footnote-55).

<!-- discussion? -->

If a multi-cloud architecture is not needed, one could instead opt for Google's Config Connector, or AWS' Controllers for Kubernetes.

<!-- move to discussion? -->

<!-- Transitioning -->

It can be a big jump to rewrite your entire infrastructure to use control planes instead of Terraform, which is why tools like *Kubeform[[56]](#footnote-56)* and the Terraform provider for Crossplane[[57]](#footnote-57). Kubeform provides auto-generated Kubernetes Custom Resource Definitions (CRDs) for Terraform resources so that you can manage any cloud infrastructure in a Kubernetes native way. This requires you to rewrite your HCL to Kubernetes CRDs, so if that is too time-consuming, you can instead use Crossplane's Terraform provider. This provider lets you copy-paste your Terraform syntax directly into a CRD, and Crossplane will, in concept, run terraform apply automatically. This could be an intermediate step before doing a complete transition from Terraform to Crossplane.

## 9.2. ArgoCD and Crossplane together

<!-- intro to ArgoCD and Crossplane -->

The Core Cluster uses Crossplane for provisioning cloud resources and uses ArgoCD to deploy and manage all services that are running in the Core Cluster and the App Clusters. Crossplane and ArgoCD are both open-source control plane-based tools funded by the CNCF. ArgoCD handles all internal state (e.g., deploying containers and configuration), while Crossplane handles all external states (e.g., provisioning cloud resources) - combined, they can be used as a universal control plane for managing multi-cloud multi-environment infrastructure.

As seen in Figure X, ArgoCD is responsible for applying the Crossplane resource manifests.

Diagram

Description automatically generated

Figure 4

Crossplane itself and the Kubernetes manifests/objects used by Crossplane are all declared in manifests, checked into git, and synced by ArgoCD. Figure X is a visualization of how ArgoCD and Crossplane work together to provision cloud resources.

<!-- står allerede i State automation. Måske det skal flyttes her til.

As mentioned previously, the question is not whether Crossplane or ArgoCD are great tools or not - but more about whether the paradigm of control planes is good in general. As Eficode states: "If Crossplane does not strike the right balance and abstraction level, the next control plane will." [[s](https://www.eficode.com/blog/outgrowing-terraform-and-adopting-control-planes)]

-->

<!-- move to discussion? -->

This setup can be extended by other tools. Crossplane focuses currently on cloud providers, so if a user wants to manage external resources that are not cloud-related, the user could simply install a control plane for that as well. The main point is that only ArgoCD and Crossplane are necessary for the use cases presented in this paper, but other control planes could easily be added if more features were needed.

## 9.3. Managing internal state with ArgoCD

<!-- intro -->

Everything deployed to the Core Cluster (besides ArgoCD itself) and the App Clusters are declared in manifests, checked into git, and synced by ArgoCD.

<!-- one or more clusters -->

ArgoCD can either be installed on each cluster individually (only controlling the local state) or on a single shared cluster which then handles the deployment to multiple clusters.

Installing ArgoCD on each cluster means there is no shared interface of all the infrastructure running across clusters. You would have to have multiple endpoints and multiple ArgoCD-profiles/-credentials for each ArgoCD instance running in each cluster, which may not be desirable if you run infrastructure on a large scale. Furthermore, it also consumes more resources to run all ArgoCD’s components on each cluster (vs. only on a single cluster), which may be a consideration if your company's budget is tight.

The implementation presented in this paper has a single instance of ArgoCD running on the Core Cluster, and it deploys and manages all the Kubernetes Objects running in both the Core Cluster and the App Clusters.

<!-- Grouping vs. separation -->

ArgoCD applications can be nested and grouped arbitrarily. My experience when developing the system is that smaller groupings are desirable because they can be synced independently with more fine-grained control. For example, grouping the Applications with the Crossplane-provisioned database separately from the Application with Crossplane-provisioned Kubernetes App Clusters. This way, I can create/deploy and delete the two Applications independently. Based on this philosophy of separation, I have chosen to structure my applications as seen in Figure X.

Diagram

Description automatically generated

Figure 5: An illustration of what packages/services ArgoCD installs and on which cluster.

As seen in Figure X, besides Crossplane and Quote App, ArgoCD also installed other packages/services:

Nginx

In order to call the *demonstration application*'s frontend from outside Kubernetes, we need to set up ingress. Nginx is an ingress controller that acts as a reverse proxy and load-balancer and handles all external traffic into the cluster. A cluster needs an ingress controller in order to call the endpoints inside Kubernetes from outside Kubernetes.

ExternalDNS

ExternalDNS synchronizes exposed Kubernetes Services and Ingresses with DNS providers[[58]](#footnote-58). In order to automatically create DNS records in GCP when new Ingress objects are created in GKE, ExternalDNS needs to be set up and deployed in the cluster[[59]](#footnote-59).

#### Prometheus and Grafana

Prometheus and Grafana is an open-source monitoring stack. This stack is not strictly needed to run the Quote App, but it is used to resemble a realistic infrastructure setup seen in a company. It is installed on both the core clusters and the app clusters, so it is possible to observe, e.g., resource usage of all clusters.

#### Manifest-Syncer

manifest-syncer is a custom service I developed to sync secrets between the Core Cluster and the App Clusters. This service will be described in detail in section: XXXX.

### 9.3.1. Eventual consistency

At its core, all ArgoCD does is that it reads from a given branch on a given repository and applies all the resources that it finds to Kubernetes. By default, there is no order to this process, but ArgoCD will simply apply all the manifests, and then Kubernetes will handle the rest (like scheduling *Pods* and *Jobs*).

<!-- how Argo handles eventual consistency -->

In any modern software environment, there exist dependencies. The number of dependencies depends on how well a system is decoupled. If a company had to spin up its entire infrastructure from scratch, it would probably include a lot of sequential steps in a specific order because some of its services need other services to run. Doing sequential deployments through a script often takes a long time because it runs sequentially and not in parallel. With eventual consistency in Kubernetes, multiple steps/jobs can run simultaneously, and they will be executed eventually when the steps/systems they depend on are done. In Kubernetes, there is no order of when which resources/events are created/handled. For example, if a Pod is created in the cluster, it won't necessarily be scheduled immediately. Instead, it will be created eventually when the right conditions are present (e.g., enough CPU).

Applying Kubernetes resources with ArgoCD works the same way. If ArgoCD can't deploy an ArgoCD Application (because some dependency may be missing), it will just automatically try again a minute later. This means I can apply our entire infrastructure at once with ArgoCD, and ArgoCD will make sure everything will be deployed with eventual consistency even though there are broken dependencies temporarily in the process. This also applies every time you update something in the infrastructure. Configuration and workload will be applied and scheduled whenever ArgoCD eventually syncs its state with what is stored in the chosen git repository. An example of this is that the quote-app-frontend will fail to deploy if Nginx is not installed in the cluster at deployment time. ArgoCD will keep trying to deploy the quote-app-frontend' ingress configuration until Nginx eventually exists in the cluster.

### 9.3.2. Repository structure

The structure for my implementation of a universal control plane is split up into 3 repositories: One ArgoCD synced repository for the Core Cluster, a second ArgoCD synced repository for syncing with App Cluster, and finally, a general repository with code and scripts for bootstrapping the system.

The repositories can be found here:

* k8s-ucp-bootstrap: https://github.com/dag-andersen/k8s-ucp-bootstrap
* k8s-ucp-core-gitops: https://github.com/dag-andersen/k8s-ucp-core-gitops
* k8s-ucp-app-gitops: https://github.com/dag-andersen/k8s-ucp-app-gitops

The bootstrapping repository mainly contains scripts for starting Core Cluster. The GitOps synced repositories do not contain anything other than manifests synced with ArgoCD.

root

├── k8s-ucp-bootstrap # Repository with bootstrapping scripts

├── k8s-ucp-app-gitops # Repository only containing app-cluster manifests synced with ArgoCD

└── k8s-ucp-core-gitops # Repository only containing core-cluster manifests synced with ArgoCD

('k8s-ucp' stands for 'Kubernetes Universal Control Plane'.)

The k8s-ucp-app-gitops and k8s-ucp-core-gitops repositories could be merged into a single repository and store every ArgoCD-application for every cluster in a single repository (as shown below). It may even be preferred because it will be easier to update the structure for a system requiring changes in multiple clusters simultaneously.

On the other hand, you may not want everyone in the organization to have read-access to all infrastructure. The write-access would not be a problem since you can, e.g., use CODEOWNERS on GitHub[[60]](#footnote-60).

root

├── k8s-ucp-bootstrap # Repository with bootstrapping scripts

└── k8s-ucp-gitops # Repository containing all ArgoCD synced Resources

├── app-cluster

└── core-cluster

The main point here is that there are many ways to structure your GitOps synced repositories, and it all depends on what kind of needs you have in your organization.

The folder structure of the Core Cluster and App Cluster can be seen in Figure X. Most resources that are applied by ArgoCD are structured/built with Kustomize using the *base-overlay-pattern*[[61]](#footnote-61), which is why all base- and overlays-folders exist.

<!-- maybe delete? -->

I use *App of Apps Pattern*[[62]](#footnote-62) for bootstrapping the ArgoCD Applications, which means I have a single ArgoCD Application that creates all the other ArgoCD applications. This makes it easier to deploy because I only deploy a single bootstrapping application. This ArgoCD Application is named kube-applications in the code.

App Cluster Git Repo | Core Cluster Git Repo

k8s-ucp-core-gitops (repository root)

├── projects # Declaration of how Argo Applications are grouped

├── argo-bootstrap

│ ├── gcp

│ └── kind

├── argo-config # Ingress configuration for accessing the argo server on GCP or kind.

│ ├── base

│ └── overlays

│ ├── gcp

│ └── kind

├── aws-provider

├── aws-database

├── gcp-provider

├── gcp-database

├── gcp-clusters

│ ├── base

│ └── overlays

│ ├── prod

│ ├── prod-pre

│ ├── stage

│ └── stage-pre

├── manifest-syncer

└── kube-applications

├── base

├── envs

│ ├── core

│ └── experimental

└── host

├── gcp

└── kind

k8s-ucp-app-gitops (repository root)

├── argo-bootstrap

│ ├── prod

│ └── stage

├── quote-app-with-database-aws

│ ├── base

│ │ ├── backend

│ │ └── frontend

│ └── overlays

│ ├── prod

│ └── stage

├── quote-app-with-database-gcp

│ ├── base

│ │ ├── backend

│ │ └── frontend

│ └── overlays

│ ├── prod

│ └── stage

└── kube-applications

├── base

└── overlays

├── prod

└── stage

## 9.4. Managing External State with Crossplane

In my implementation of a universal control plane, I use Crossplane for managing external resources (e.g, databases on AWS and Kubernetes clusters on GCP).

In order to authenticate with the cloud provider API, Crossplane needs to have access to credentials. In this case, it would be an IAM User for AWS and a Service Account for GCP[[63]](#footnote-63).

I inject the credentials into the cluster as Secrets by running:

$ kubectl create secret generic gcp-creds -n crossplane-system --from-file creds=../creds/creds-gcp.json

$ kubectl create secret generic aws-creds -n crossplane-system --from-file creds=../creds/creds-aws.conf

Where ../creds/creds-gcp.json and ../creds/creds-aws.conf point to the credentials stored on my local machine.

The Crossplane provider is specified in a manifest of type Provider, while the credentials are referenced in a manifest of type ProviderConfig. When these are deployed to the cluster, I can start provisioning resources.

As an example, to provision a database resource on AWS, I need to create a Provider (specifying AWS), ProviderConfig (specifying to use aws-creds-secret as authentication), and a RDSInstance (specifying the database properties) and apply it to a Kubernetes cluster with Crossplane installed[[64]](#footnote-64).

apiVersion: pkg.crossplane.io/v1

kind: Provider

metadata:

name: aws-provider

spec:

package: crossplane/provider-aws:v0.30.1

---

apiVersion: aws.crossplane.io/v1beta1

kind: ProviderConfig

metadata:

name: aws-provider-config

spec:

credentials:

source: Secret

secretRef:

namespace: crossplane-system

name: aws-creds

key: creds

apiVersion: database.aws.crossplane.io/v1beta1

kind: RDSInstance

metadata:

name: postgres-instance

spec:

forProvider:

region: eu-central-1

dbInstanceClass: db.t2.small

masterUsername: masteruser

engine: postgres

engineVersion: '12.10'

skipFinalSnapshotBeforeDeletion: true

publiclyAccessible: true

allocatedStorage: 20

providerConfigRef:

name: aws-provider-config

writeConnectionSecretToRef:

namespace: crossplane-system

name: aws-database-conn

The AWS Crossplane-provider will read the above RDSInstance and check that such an instance exists on the AWS account. This is how Crossplane is able to create VPCs, Subnets, Node Pools, Kubernetes Clusters, and databases needed in the demonstration setup to run the Quote App. All the manifests for resources needed in GCP can be found in /gcp-clusters[[65]](#footnote-65) in the core-cluster-argo-repo-repository.

<!-- new providers - discussion?-->

Crossplane is built to be highly extendable (just like Terraform), making it easy to create new providers. Currently, not many providers exist, but I could imagine, for example, Datadog could create a Crossplane-provider (equal to their Terraform provider integration), where the user could declare their Datadog dashboard in manifests and apply it to the cluster. With Terraform, they would have to store the Datadog dashboard terraform in for example, a bucket on a cloud provider. This works fine in practice, but one could argue that we don't need to store that state in a bucket. Instead, we could simply store the declared state directly in Kubernetes together with the services you are monitoring.

<!-- One instance or multiple instances of Crossplane? -->

Just like ArgoCD, you can either install Crossplane on each cluster or install it in a shared cluster. Just like with ArgoCD, it provides much better visibility only to have a single instance running, making it easier to see which external resources are running outside Kubernetes.

Running Crossplane on a shared/core cluster also decouples the external resources from the actual clusters. This means that you don't lose the connection with the staging-database just because you close down the staging-cluster temporarily. You rather want your external infrastructure to be managed from a cluster that you know will remain up and running.

<!-- connection details -->

An important detail is that when Crossplane creates a resource (e.g., database instance), it stores the connection details in the cluster on which Crossplane is running within. The problem here is that you often need the credentials in app clusters (e.g., you want your services running in the production environment to connect to the production database). There are many ways to handle secrets/credentials, but more on this in the *Distributing secrets*-section.

## 9.5. Distributing Secrets

This section will describe a technical detail how I close the gap between ArgoCD and Crossplane. The reader can skip this section if he/she is mainly interested in the overall design of the system.

When Crossplane creates a resource (e.g., Kubernetes cluster or database) on a cloud provider, it stores the connection details (e.g., access credentials) in the cluster where Crossplane is installed. This is a problem since the connection details are needed in App Clusters, where all the business logic is running. So far, no automated native way of making the secret available in the App Clusters.

The challenge is also described as an issue on the crossplane-contrib-GitHubOrganisation[[66]](#footnote-66), and currently, no easy solution exists.

This shows how popular tools like ArgoCD and Crossplane do not necessarily integrate well together natively. These small gaps can easily occur when we are using many different tools from the Kubernetes ecosystem that were not necessarily meant to be used in conjunction with each other and do not have a native integration between them. Infrastructure may have to close these gaps themselves if they can’t find an off-the-shelf component online (e.g., GitHub) that solves your problem. Many of these small gaps can be solved with a few scripts, a cronjob running a script, or a small standalone service.

There are a few ways of overcoming this secret-distribution challenge. The most naive one would be to create a manual step where the infrastructure team needs to somehow copy the credentials to the production cluster when a new cluster is created. For example, running this line of code for each secret they want to be copied every time a new cluster is created: kubectl get secret my-secret-name --context core-cluster --export -o yaml | kubectl apply --context new-app-cluster -f -

Another way of doing this is using some kind of secret-vault (like HashiCorp Vault[[67]](#footnote-67)) where the credentials are stored at the creation of the database. Each cloud environment can then read the credentials directly from the vault when needed. This may be considered a better solution and may come with some great benefits (which are beyond the scope of this paper) - but it nonetheless introduces even more tools/concepts to the infrastructure, which may put even more workload on an infrastructure team.

Therefore, I have created a much simpler automated solution.

I have created a fully declarative solution with eventual consistency. I have implemented a service named manifest-syncer that runs as a container inside Kubernetes. The purpose of the manifest-syncer is to mirror secrets from its host cluster to target clusters. The manifest-syncer is simply deployed to the cluster with default configuration and is controlled through CustomResourceDefinitions. If a developer wants a Secret to be automatically mirrored/copied from the Core Cluster to e.g., the production cluster, he/she just creates a manifest describing exactly that and apply it to the Core Cluster. An example of such a manifest can be seen in Figure X. In this example, it is specified that Secrets named gcp-database-conn, in namespace crossplane-system, should be copied to namespace default on the cluster named gcp-cluster-prod.

apiVersion: dagandersen.com/v1

kind: Syncer

metadata:

name: secret-syncer

annotations:

argocd.argoproj.io/sync-options: SkipDryRunOnMissingResource=true

spec:

data:

- sourceName: gcp-database-conn

sourceNamespace: crossplane-system

kinds: secret

targetCluster: gcp-cluster-prod

targetNamespace: default

Note: argocd.argoproj.io/sync-options: SkipDryRunOnMissingResource=true is added to ensure that ArgoCD does not fail the deployment because Syncer does not exist as a custom resource at deployment time. This can happen when Syncer-manifest is applied before the manifest-syncer is deployed. ArgoCD will fix the failing resources with eventual consistency.

### 9.5.1. Getting access to the App Clusters

In order for the manifest-syncer to have access to the App Clusters, it needs a kubeconfig. I do not want to provide or generate this kubeconfig manually each time I create a new cluster. Instead, I want the manifest-syncer to fix this automatically without having to change other services.

The manifest-syncer automatically scans its host cluster for secrets generated by Crossplane with a name that contains: -k8s-. This alone is not enough because it only gives read access to the cluster. To gain write access, it scans its host cluster for secrets generated by ArgoCD with the label: argocd.argoproj.io/secret-type=cluster and then retrieves ArgoCD's access token to the App Clusters. The manifest-syncer combines the kubeconfig and access token and gets access to the App Clusters. The manifest-syncer repeats this process every 10 seconds to continuously detect when new App Clusters are created.

Diagram

Description automatically generated

Figure 6: A visualization of how credentials are generated and copied to the app clusters.

In figure X, it is illustrated how the manifest-syncer running on the Core Cluster reads ArgoCD's access tokens and the kubeconfigs (generated by Crossplane) to copy the database credentials (generated by Crossplane) to the App Clusters.

One could argue that it is bad practice to build your own small services like this because you need to maintain them yourself - but since the service is self-contained and does not directly interact with other services, it can easily be replaced by a better solution, should a company choose to invest in a more mature solution (like installing a secret-vault).

# 10. Demonstration Application running in Google Cloud

Continuing from the *Demonstration Application*-section, we now have all the pieces to run the Quote App in a multi-environment spanning across multiple cloud providers.

For demonstration purposes, the Kubernetes clusters will run in GCP while the managed database will run in AWS to show that this kind of setup works across different cloud providers. On GCP, there will be two environments running: *production* and *staging*. Each environment runs in its own VPN (and subnetwork) and has its own subdomain on GCP. Both environments can connect to the database running on AWS.

<!-- Crossplane resources print -->

The cloud resources needed for this setup are provisioned through Crossplane and can be seen in Figure X. Crossplane does not have a UI, but you can interact with it with kubectl like any other Kubernetes resource. Running kubectl get managed will print a list of all the resources managed by Crossplane together with extra metadata. An example of metadata would be the column SYNCED that shows if the resource's actual state in the cloud provider is in sync with the declared state in Kubernetes.

Graphical user interface, text

Description automatically generated

Figure 7

Figure X shows two VPCs (network.compute.gcp), two subnets (subnetwork.compute.gcp), two k8s clusters (cluster.container.gcp), and two node pools (nodepool.container.gcp) running on GCP, and a single database instance (rdsinstance.database.aws) is running on AWS.

Combining the objects we saw in Figure X (in the *Demonstration Application*-section) and the Crossplane cloud resources in Figure X, we get the following infrastructure and application architecture:

Graphical user interface

Description automatically generated with medium confidence

Figure 8

Figure X: This illustration shows how the demonstration application (Quote App) runs on GCP and accesses a database in AWS. All elements with the Crossplane logo next to them are objects/resources provisioned by Crossplane, while all elements with the ArgoCD logo next to them are objects/resources deployed and managed by ArgoCD. The text written in *italics* is the name of the Crossplane-object managed in Kubernetes. The names match the objects printed in Figure X.

<!-- Scaling -->

The production and staging environments run completely separately on GCP. This design makes it possible to scale the number of app-clusters/workload-environments linearly from the Core Cluster.

<!-- Universal control - Den her virker underlig.-->

All the resources and objects seen in Figure X are managed by ArgoCD and Crossplane, which is running in the Core cluster, acting as a universal control plane for provisioning infrastructure and deploying workloads.

## 10.1. Use in practice

The following section describes how to use this implementation of the universal control plane (the Core Cluster) and how a software team would develop and deploy services running in App Clusters such as production-cluster.

The *implementation*-section described how one instance of a system using Kubernetes and control planes could look, but it but we still need to cover how we use the setup after creation. The setup/system needs to be maintainable and modifiable over time.

There are different methods and strategies when it comes to the deployment of software. This paper gives an example of how to use ArgoCD together with Crossplane, but others may choose to structure their code differently. For example, if a user chooses to use FluxCD (instead of ArgoCD) and Google's Config Connector, the optimal structure may look very different.

<!-- discussion? -->

Furthermore, keep in mind that many of the tools used in this setup are under active development, so the feature set of these tools may change in the future and change the workflows.

<!-- If we want to convince ourselves that this is actually an elegant setup, we first need to envision how it would be used in practice in a company. -->

In this section, I will go through some examples of how such a system would work in practice if implemented in a company.

**The use cases are:**

* Spinning up the cluster from scratch
* Developer creating a new service with an associated database
* Deploying a new service version to multiple cloud environments

These use-case examples assume that the folder structure is the same as described in the *Repository Structure*-section.

### 10.1.1. Spinning up the cluster from scratch

Starting the Core Cluster, multiple App Clusters, and deploying every service/system described in the *managing internal state*-section can be done in 4 steps.

1. Pull down the k8s-ucp-bootstrap-repository[[68]](#footnote-68).
2. Run make install-tools in the root to pull down the dependencies.
3. Add your Cloud Provider and Git-repo credentials to the ./creds/-folder.
4. Run make start in the repository root to start the interactive CLI for choosing what resources to create.

Running make start will show the following terminal printout shown in Figure X.

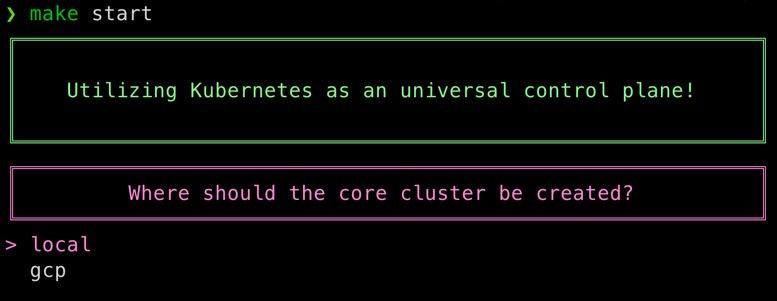


Figure 9

<!-- This interactive-CLI can be replaced with something else, e.g., it would be possible to replace the CLI with a simple UI if that is preferred. -->

First, specify if the Core Cluster should run locally with Kind or on GCP.

<!-- Other options can be added relatively easily by adding scripts to the bootstrap-repo. -->

Choosing local is preferred when developing since it takes around 10 minutes to spin up a cluster on GCP, and it only takes 20 seconds to spin up a local Kind cluster.

Text

Description automatically generated

Figure 10

Secondly, specify what App Clusters should be created (production or staging) and if the Quote App should be deployed together with an associated database. Choosing skip only starts the Core Cluster. To deploy the Quote App at a later point, just run the make start-command again and select quote-app-with-aws-database or quote-app-with-gcp-database. It will detect that you already have a Core Cluster running and deploy the Quote App, and provision a database on AWS.

By choosing skip, the script will spin up the Core Cluster install ArgoCD, and deploy Nginx. ArgoCD will then apply all the Crossplane-related manifests and deploy the manifest-syncer. Now that the core components are running and the Core Cluster can be used as a control plane for spinning up cloud resources and deploying apps to external Kubernetes clusters.

The time it takes from the make start-command is run till the Quote App is running and accessible through a web browser depends on if the Core Cluster locally or on GCP. Provisioning a GKE cluster on GCP takes around 10 minutes[[69]](#footnote-69). So, if the Core Cluster is chosen to run on GCP, then it will take about 10 minutes longer than if local was chosen. Overall, choosing local results in a start time of around 15 minutes, and choosing gcp results in a start time of around 25 minutes.

### 10.1.2. Developer creating a new service with an associated database

If a developer wants to create a deploy an application to, e.g., the staging cluster together with a database instance, the flow would look something like this:

1. The developer creates a *pull request* to the core-cluster-gitops-repo with the manifests describing the database instance and a syncer object for distributing the connection details to the database (as explained in *Distributing Secrets*-section)
2. The developer creates a *pull request* to the app-cluster-gitops-repo with the manifests describing the application. This could, for instance, be manifests describing *Ingress*, *Services*, and *Deployments* as with APP Cluster.

Depending on the size and policies of this imaginary company, it could, for example, be a person from the infrastructure team approving the provisioning of this new database, while it may be someone from the development team approving the normal application-related Kubernetes resources (as illustrated in Figure X).

Diagram

Description automatically generated

Figure 11

### 10.1.3. Deploying a new service version to the multiple cloud environments

If a developer wants to update an application on the production or staging cluster, the flow will look something like this:

Most ArgoCD resources in this setup are generated using Kustomize. Kustomize uses a folder-pattern, where each configuration for a service is stored in the overlay folder. In this case, the configuration *service A* for the production environment would be stored in organization-name/service-a/overlays/prod/.

Let us assume a developer has a containerized service named: service-a. When a new version of service-a needs to be deployed, the developer creates a new *pull request* with the committed changes. If the commits are not on the main/master branch, then the build pipeline builds the image and pushes it to, e.g., Docker Hub with a unique tag. In addition, the pipeline also updates the service version in app-cluster-gitops-repo-repository with the new tags. In this case, the build system updates the kustomization-file in organization-name/service-a/overlays/stage/ with the newest version that should be deployed to the staging cluster.

When the changes have been tested on the staging cluster, and the PR has been merged into master/main, then the same process begins. The only difference is that the build-pipeline this time updates the organization-name/service-a/overlays/prod/ instead.

Diagram

Description automatically generated

Figure 12

The *Argo Project* (the organization behind all Argo tools) does not provide any opinionated standardized way of pushing a new commit with the new image tag/version. So developers have to write this custom code for pushing an update to the GitOps-synced repository themselves.

# 11. Discussion and evaluation of the implementation

I have now explained how control planes work and how to build an entire system based on control planes, and I have shown how such a system could be used in practice.

This section discusses some of the challenges and limitations of using Kubernetes as a universal control plane. These topics will give Eficode a better baseline for discussing the pros and cons with their clients when considering if they should transition control plane managed infrastructure.

Each topic covered in this section will not be described in detail but will mostly be highlighted as my observations and opinion of what to keep in mind when deciding to move to control plane-managed infrastructure and what to keep in mind when designing a production-ready system in a company. It is difficult to give a definite conclusion on each of these topics since it all comes down to the specific tools and the exact implementation a company chooses to implement.

* The discussion topics are:
* Build pipelines
* Additional cost
* No preview with Crossplane
* Single interface
* Platform Engineering
* Pets to Cattle
* Declaring the entire infrastructure
* Bootstrapping Problem
* Multiple core-clusters
* Maturity level
* Stateless Infrastructure

## 11.1. Build pipelines

Switching to a GitOps workflow with tools like ArgoCD still requires using pipelines. With the demonstrated setup, I still need some kind of build-pipeline that runs when a new version of an application is pushed.

Graphical user interface

Description automatically generated with low confidence

Figure 13: This figure shows 3 different ways of building deployment systems

CI/CD pipelines often consist of 4 steps: *pull source code*, *build code*, *test code*, and *deploy* [[70]](#footnote-70). With Kubernetes, that results in *pull source code*, *build and push image*, *test code/image*, and *deploy by applying manifests to Kubernetes* with kubectl. ArgoCD introduces a new step where the GitOps-synced-repository is updated (there is no standardized way of updating this GitOps-synced-repo, but it can be done with a custom script that pushes a new version tag to the repository). So, with ArgoCD, there are still 4 steps in the pipeline: The last step is just replaced with updating a repo instead of applying manifest changes to the cluster directly.

So, in that sense, I have not improved much. I still have 4 steps in the deployment pipeline if I use GitOps/ArgoCD, but I have now removed the need for direct access to the cluster from the pipelines. Instead, the pipeline needs access to the repo, where it is supposed to push the changes.

All the steps in the pipelines are so far imperative. To avoid the need to create this custom imperative code that pushes the new configuration to the GitOps-synced-repository, the argocd-image-updater-project was created. argocd-image-updater is part of the *Argo Project* and tries to tackle this challenge by moving the “image-version-update-logic” into a Kubernetes operator. Now the pipeline system should only worry about building and pushing images. I tried the argocd-image-updater but experienced issues connecting to *Docker Hub* (as explained in an issue on GitHub[[71]](#footnote-71)). The argocd-image-updater-project has been in development for at least 2 years without reaching a stable state. So, I will not consider this a reliable option for the time being. A similar tool exists in the FluxCD ecosystem[[72]](#footnote-72). The Flux automated image updater may work, but since we went with the ArgoCD ecosystem, it was not reasonable to try out the flux version in this limited time frame.

*To conclude:* By switching your CI/CD system to a GitOps tool like ArgoCD, we have not removed the need for deployment pipelines. Instead, we have only made the deployment process more complex.

## 11.2. Additional costs

When using Kubernetes as a control plane for managing my infrastructure, I am running an additional Kubernetes cluster (the Core Cluster), which is not free. Since this setup entails that I run a Core Cluster (which does not provide any direct customer value), it will naturally mean that we spend more money on cloud resources than if I did not run a Core Cluster.

When running the demonstrated application (Quote App), more resources were required to run the Core Cluster than the staging and production clusters together. The ratio between the resources required by Core Cluster and App Clusters depends on how much workload you run in the App Clusters. For my small demonstration application, the resources required by the Core Cluster surpassed the other clusters combined, so if a small company/team would adopt this setup, the economic aspect may be relevant.

For the Core Cluster to run properly in GCP without issues, 3 nodes of type "e2-medium" are needed. Monthly, this is $73,38 ($24.46 · 3) for running just the Core Cluster[[73]](#footnote-73). The size of the machines needed will, of course, depend on how much shared infrastructure/how many control planes are running in the Core Cluster.

*To conclude*: Running a Core Cluster is not for free. For small companies with a tight budget, it is something to keep in mind when deciding the implications of switching to control plane-managed infrastructure.

## 11.3. No preview with Crossplane

As I see it, one of the biggest drawbacks of using Crossplane for infrastructure-management compared to Terraform is that there is no option to preview a change before they are applied. With Terraform, the developer can run terraform plan to see a preview of the changes that will happen. This makes it possible to review the changes first before committing to the new configuration.

With Crossplane, there is no such feature. Crossplane cannot show a preview of the resources it is going to create/modify/delete. So, the developers can only apply the manifests and hope that they did it correctly. This makes Crossplane risky to use for critical infrastructure. This issue is also described on GitHub[[74]](#footnote-74).

I don't know if there will be a preview-feature or "dry-run"-feature (where it runs the new configuration but without changing anything) in Crossplane in the future. I don't know how it would work, but I imagine preview-features are inherently difficult to make in control planes because it is not obvious when in the process, a preview of the changes should be reviewed by a developer.

*To conclude*: Crossplane has no preview-feature for reviewing changes applied to the infrastructure configuration. This makes Crossplane risky to use for critical infrastructure, and I would consider this the biggest drawback of using Crossplane for infrastructure-management.

## 11.4. Single interface

With tools like Crossplane, the development teams do not even need to have (write) access to the cloud provider. All external resources can be managed through tools like Crossplane. Crossplane would be the only entity with (write) access to the cloud providers. "*The (cluster) administrator then uses the integrated Role Based Access Control (RBAC) to restrict what people can ask Crossplane to do on their behalf*."[[75]](#footnote-75). All access control can be moved to kubectl - making Kubernetes the only platform developers need access to. This is also covered in Eficode's article: "Outgrowing Terraform and adopting control plane"[[76]](#footnote-76).

Suppose you want to take it one step further. In that case, ArgoCD could be the only entity with write-access to clusters - enforcing that everything is version controlled (checked into git) and reviewed by multiple parties before any change goes into production. With tools like ArgoCD, the development teams do not need to have (write) access to the clusters directly. All Kubernetes objects could be managed through tools like ArgoCD.

*To conclude:* By combining ArgoCD and Crossplane, you can create a workflow where all external resources and application logic are checked into git and can only go into production through *Pull Requests*. Developers only need write access to git and nothing else, which creates a single interface for the developer to use and interact with. How strict you want your permissions all depends on the policies and amount of trust in the organization. This setup with Crossplane and ArgoCD makes it possible to create a very restrictive system if needed.

## 11.5. Platform Engineering

Platform engineering is gaining popularity in the last two years[[77]](#footnote-77). This topic deserves a paper in itself and is therefore not the focus of this paper, but I just shortly want to highlight how control planes like Crossplane can modernize infrastructure-management by embracing Platform Engineering and self-servicing. Crossplane has the concept of *Composite Resources*[[78]](#footnote-78), which works as an abstraction layer between the managed resource running on the cloud provider and the resource offered by the infrastructure team to the development teams. For example, a developer then does not have to worry about where a database runs. The developer just creates a manifest describing a database, and the rest is handled by the abstraction created by the infrastructure/platform team. The abstraction becomes a platform for the developer to use - and they can self-service/provision infrastructure by using the provided abstraction. Developers will only interact directly with Kubernetes and not any other cloud platform/portal.

Kelsey Hightower from Google puts it like this: *"This conversation is less about Crossplane vs. terraform. This is more about using Crossplane to build your own kind of API server, your own control plane that highlights and exposes only the resources and the properties that you want people in your organization using. This is a little bit different than saying: »Hey, here is access to GCP. knock yourself out until we get the bill«"* [[79]](#footnote-79). In other words, the developers can view the platform team as the cloud provider instead of seeing GCP as their cloud provider. Everything the developers need is exposed through the abstraction provided by the platform team[[80]](#footnote-80).

*To conclude:* Crossplane enables an infrastructure team to build an engineering platform where the developers can self-service cloud resources provided by the infrastructure team. The infrastructure team has full control over what resources are available in the organization and how they are configured behind the abstraction layer.

## 11.6. Pets to Cattle: Denne title passer bedre til eliminating state.

One challenge a company can have when their developers have direct access to the cloud resources is that cloud services get created and forgotten about[[81]](#footnote-81). These resources may have been created accidentally or just used for a quick experiment. Other reasons could be that the resource is irrecoverable because the Terraform state was lost. The company ends up being charged for these unutilized resources each month because it lacks the knowledge if the services are actually used or not, and the company is too scared of deleting the resources because they may be in use. Tools like Crossplane and ArgoCD can limit or mitigate this risk.

<!-- orphan -->

ArgoCD has a feature to display all "orphan" resources not handled by ArgoCD (meaning the resource is no longer or never was checked into git). This is great for getting an overview of the resources not stored as IaC. Finding these cases can be essential in eliminating a false sense of security of the system being re-creatable should it go down. If these cases are not detected, an infrastructure team may think that they can re-create their production cluster without any issues, but in reality, their services depend on a resource that was created manually and never checked into git.

<!-- Pets to Cattle -->

If you lose trust in the reproducibility of your infrastructure, you start treating your infrastructure as pets that you have to protect at all costs. Having a Core Cluster that manages many App Clusters helps you go from *Pets to Cattle[[82]](#footnote-82)*. The easier it is to spin up new clusters, the less we will treat our infrastructure as "pets"[[83]](#footnote-83). This paper's implementation can manage an infinite number of disposable App Clusters.

*To conclude:* Provisioning resources with Crossplane ensures visibility of which cloud resources exist in the organization. Combining it with ArgoCD will create visibility of which resources are not checked into git. Overall, Crossplane and ArgoCD build confidence in the reproducibility of the system by creating visibility of what cloud resources and what software is provisioned/deployed from git.

## 11.7. Declaring the entire infrastructure

The setup described in this paper is built using only 2 file types: YAML and makefiles. YAML is used for declaring the state of the entire infrastructure-configuration, while makefiles are only for the initial bootstrapping. All resources required to run this implementation are declared as *Infrastructure as Code* and checked into git.

"*Since Crossplane makes use of Kubernetes, it uses many of the same tools and processes, enabling more efficiency for users - without the need to learn and maintain entirely new tools*"[[84]](#footnote-84). This creates a highly streamlined infrastructure because it does not require knowledge about, e.g., Terraform, Ansible, or multiple scripting languages. I consider this huge benefit of this setup.

<!--

Furthermore, it can be demanding and expensive to modernize a company to the newest technologies because the employees may not have the required skills and knowledge to utilize all the latest tools. Therefore it is beneficial to limit the number of languages and systems introduced in the company. Instead of needing to learn the syntax and mental model of many different tools, then it would be advantageous to only need the knowledge about Kubernetes. Having a single abstraction for everything.

-->

*To conclude:* Building your infrastructure using control planes in Kubernetes (like Crossplane and ArgoCD) ensures your entire infrastructure is declared in manifests. An infrastructure team can define the entire infrastructure in Kubernetes manifests and does not need to know any other tool-specific languages.

## 11.8. Bootstrapping Problem

Every automatic process requires an initial command to start the process. This initial task/command cannot be declarative since a command is, by definition, imperative[[85]](#footnote-85).

In order to simplify the setup process as much as possible, I aimed to make the bootstrapping as simple, clean, and error-safe as possible. The only bootstrapping done in this implementation is starting a cluster (used as Core Cluster) and then installing ArgoCD on it. Installing all other components (e.g., Prometheus, Nginx, ExternalDNS, Crossplane, etc.) in both the Core Cluster and the App Clusters is handled automatically by ArgoCD eventual consistency.

*To conclude:* I can't remove bootstrapping entirely - but I can try to reduce it as much as possible. I would argue that the bootstrapping done in this paper’s implementation is fairly minimal since it only does 2 steps. *1*: Spin up core-cluster (locally or on cloud provider), *2*: Install and set up ArgoCD.

## 11.9. Multiple core-clusters

Just like it can be beneficial for a company to have a staging cluster to test its software before it goes into the production cluster, it would be beneficial to test new software before it goes into the Core Cluster. E.g., it would probably be a good idea to test a new version of Crossplane before its upgrade in the Core Cluster. This would indicate that a company may want a staging/testing/experimentation version of the Core Cluster as well. Here the company could test and experiment with software before it goes onto the stable version of the Core Cluster.

Running multiple instances of the Core Cluster simultaneously does not work well. Multiple Core Clusters can easily be spun up at the same time - E.g., running a Core Cluster on GCP and running another instance locally. The problem is that they each have their own internal desired state and may work against each other. One cluster may want to provision a given resource on a cloud provider, while another may want to delete that resource. This results in race conditions and unpredictable behavior.

The only way to allow multiple Core Clusters to be run simultaneously is to run a complete copy of all your resources. So, the new Core Cluster copy could be named core-cluster-experimental, which then creates, e.g., a prod-cluster-experimental and aws-database-experimental, and so on. It is fairly straightforward code-wise, so it is doable, but this would effectively double the infrastructure costs.

Diagram

Description automatically generated

Figure 14

But the biggest drawback of running a complete copy of your entire infrastructure is that it adds a lot of complexity. Especially when it comes to managing a separate configuration for a duplicate cluster. One has to figure out a proper way of telling prod-cluster-experimental controlled by core-cluster-experimental, to use the IP-address/hostname of the aws-database-experimental and not the normal aws-database. While at the same time being automated and managed with infrastructure as code and checked into git. I imagine this becoming a nightmare to maintain with a lot of small edge cases where things can go wrong. This only gets worse the more infrastructure an organization handles. Solving this issue is beyond the scope of this project.

<!-- If i have to take a guess on how one would handle this it would be that all software in the core cluster should be able to run in "dry-mode". The process would be close down the stable core cluster. Boot up an experimental version where everything run in dry-mode. After seeing that evertyhing works as intendend. Then delete the experimental version and boot up the stabel version, but now with the updated control planes (not running in dry-mode) -->

*To conclude*: It is difficult to test/experiment with the Core Cluster. Running a testing/experimental version of the Core Cluster requires a complete duplication of the entire infrastructure, which may be unfeasible or unmaintainable depending on the size and budget of your organization. There is no good solution for this, and I consider this one of the biggest drawbacks of this paper’s implementation of a universal control plane for managing all infrastructure.

## 11.10. Maturity level

If a company chooses to use Kubernetes as a universal control plane for all of their infrastructure, they rely heavily on the stability and flexibility of the control planes made by big corporations or open-source communities. "*Using control planes means relinquishing control*”[[86]](#footnote-86). When they use ArgoCD, they put all their faith in that its control plane correctly deploys my services and does not randomly delete arbitrary *Pods*. When they use Crossplane, they rely on the controller/providers to correctly provision the requested resource and manage their life cycle. As a user of these control planes, it is out of your hands, and you rely solely on the tools. This is the same limitation that Terraform has. Terraform is only as good as the providers that integrate with Terraform.

Both Crossplane and ArgoCD are marked with maturity level: *Incubating*, which is meant for "*Early Adopters*"[[87]](#footnote-87) So, it is not expected that the tools give a flawless experience.

Examples of observed issues when working with ArgoCD and Crossplane

* Digital Ocean's Crossplane provider (v0.1.0) cannot delete resources on their platform (Issue is reported here[[88]](#footnote-88)). This means that Crossplane can only be used to spin up, e.g., databases and Kubernetes clusters – but not delete them afterward. This makes the provider nearly useless because you cannot control the full life cycle of resources. This will probably be fixed in the future. However, this is a good example of cloud providers not being mature and ready for control planes like Crossplane. I found this issue when I, at the start of this project, wanted to use DigitalOcean instead of AWS.
* ArgoCD cannot natively deploy resources when the generated manifests get too large (the issue is also described on GitHub[[89]](#footnote-89)). So, if a helm-charts generates manifests that is too long for ArgoCD to handle, a developer would need to install it manually through CLI or find a custom workaround online. This can be quite painful since if ArgoCD cannot deploy *every single resource* declared in your infrastructure, you must introduce custom logic for edge cases, which doesn't scale well.
* ArgoCD cannot connect to an external cluster based on data stored as a Secret in Kubernetes (the issue is also described on GitHub[[90]](#footnote-90)). ArgoCD can only connect to external clusters by running argocd add cluster <kubeconfig-context> on a machine with a kubeconfig available. This goes against the idea of declaring everything in manifests by forcing the user to call a shell command imperatively.
* The ArgoCD Server Pod would randomly crash loop for a few minutes and be completely unresponsive. In the meantime, it was not possible to access ArgoCD's UI or access it through ArgoCD's CLI interface, making it impossible to manage the deployment of my applications. This happened mostly when the Core Cluster was hosted on GCP, but I never figured out why I happened.

*To conclude*: Both ArgoCD and Crossplane are good tools with strong support from the community and industry, but they are not flawless, so one should expect to experience small issues with both of them that may be fixed/handled in the future. Especially a user should pick their Crossplane providers with care because some of them are in a very early stage and are not production ready.

## 11.11. Comparing it to Terraform

## 11.12. Stateless Infrastructure

One of the biggest selling points moving away from Terraform is the state you must manage, so if a control plane like Crossplane has not improved that process, then we have not gained much.

With Terraform, a state is stored each time aastore a state each time you provision any cloud resource. The resources must be stored in a shared place (if you are a distributed team), and you must make sure it is up to date and that only the right people have access to it. Crossplane doesn't handle state in the same way.

Terraform looks at the Terraform state, your local code, and what is currently running in the cloud. <!-- check med Zander --> Crossplane only looks at what is currently running in the cloud. If the requested resource does not exist, Crossplane will create it. If the requested resource already exists, it will not create anything. This works well with simple resources like DNS-records on GCP, but if you look at resources that need connection details, like managed databases and Kubernetes clusters, it gets more interesting! What the Crossplane provider does with each resource, if it already exists, depends on the individual resource.

If Crossplane requests a GKE cluster *that does not already exist,* it will provision a cluster and store the connection details (kubeconfig) as a secret in the cluster with Crossplane installed. If the GKE cluster already exists, it will not create anything but simply pull down the connection details (kubeconfig). That meant that if the Core Cluster gets deleted and recreated, the kubeconfig to the App Clusters *will not* be lost. This can also be seen if a user manually deletes a secret generated by Crossplane. Crossplane will simply detect it and reconcile and recreate the secret. No state lost.

This is not the case with database connection details. The password will only be pulled down on creation, but never again. This is the case with both the GCP and AWS Crossplane providers. If Crossplane requests a database *that does not already exist,* it will provision the database and store all the connection details in the cluster. If Crossplane requests a database *that already exists,* it will pull down all connection details besides the password and store it in the cluster. That means that if the Core Cluster gets deleted and recreated, the password to the database *will* be lost. This is intentional behavior by Crossplane for security reasons[[91]](#footnote-91).

This would suggest that a better approach to storing secrets is needed in this implementation if I want the Core Cluster to be stateless. As briefly explained in the *Distributing Secrets*-section, a better way of handling secrets would be installing some sort of secret-vault (like HashiCorp Vault) instead of relying on copying secrets between clusters with my manifest-syncer (as described in *9.4. Distributing Secrets*-section).

*To conclude*: The only thing blocking the entire infrastructure in this implementation from being completely stateless is the database secrets. If the project scope had been bigger, I would have introduced a secret-vault like *HashiCorp Vault* and, that way, made both the App Clusters and the Core Cluster completely stateless, and hence they could have been deleted and recreated arbitrarily. Moving from Terraform to a control plane managed infrastructure with ArgoCD and Crossplane would make the entire infrastructure configuration stateless, without the need to store a state like Terraform.

# 12. Conclusion

<!-- The golden hammer approach -->

<!-- The App Clusters are stateless and can be treated as "cattle". As long as you do not store persistent storage inside them, they are stateless. On the other hand, in this implementation, the Core Cluster is stateful and cannot be deleted because secrets will be lost. If the secrets were stored in some sort of secret-vault instead, the Core Cluster could be stateless as well and could be deleted arbitrarily. -->

<!-- Control planes are gaining popularity. -->

<!-- They are here to stay. If this iteration is not good, they next on will. -->

<!-- This project is done in collaboration with Eficode. In order for Eficode to recommend transitiong to control plane managed infrastrcture, they need to know the benefits and the implcations such a system may bring. -->

<!-- This paper implements a universal control plane for handling both software deployment and cloud resources and discuss the implications. -->

<!-- Terraform state is difficult to manage -->

<!-- configuration drift -->

<!-- This paper presents an implementation of a universal control plane for provisioning cloud resources and deploying software. -->

<!-- It handles the full lifecycle of cloud resources and container workload. -->

<!-- building a demo application to show case how you would run that on a cloud provider through the universal control plane. -->

<!--

Main pros:

- Removed state management

- Pets to Cattle

- No issues with configuration drift.

- Streamned infrastrcture

- Everything moved into Kubenretes.

- Everything checked into Git.

Main Cons:

- Extra complexity

- GitOps is just more complex than using standard deployment pipelines.

- Building a platform for engineers are adding extra abstraction layers and thus creating extra complexity.

- Difficult to test core cluster in practice.

- Experimental. Early days.

- The CNCF projects are in active development, and some features included in the projects are either experimental or not in its early phases.

-->

---------------

Dette skal stå i discussion.

<!-- general Crossplane stuff -->

This paper is not trying to argue that Crossplane is a perfect tool but rather that Crossplane is just an example of a tool that can be used to manage external resources from Kubernetes.

# 13. References

<!--

Why gitOps?

https://www.youtube.com/watch?v=l5pVzP6wsP0&ab\_channel=CNCF%5BCloudNativeComputingFoundation%5D

https://opengitops.dev/

https://www.youtube.com/watch?v=8xhEPPA6XUs&ab\_channel=Weaveworks%2CInc.

https://www.weave.works/technologies/gitops/

https://blog.mergify.com/gitops-the-game-changer/

-->

<!--

IMPORTANT LINK: https://blog.crossplane.io/crossplane-vs-cloud-infrastructure-addons/

IMPORTANT LINK: https://blog.crossplane.io/announcing-terrajet/

https://faun.pub/how-to-build-a-gitops-workflow-with-argocd-kustomize-and-github-actions-f919e7443295

"it’s a loop process created by Kubernetes that takes care of routine tasks to ensure the desired state of an object, or set of objects, running on the cluster matches the observed state." M4\_ Introduction to Kubernetes Workloads v1.7.pdf

- https://blog.upbound.io/outgrowing-terraform-and-migrating-to-crossplane/

- https://www.libhunt.com/compare-terraform-vs-crossplane

- https://www.eficode.com/blog/outgrowing-terraform-and-adopting-control-planes

- https://blog.crossplane.io/crossplane-vs-terraform/

- https://youtu.be/UffM5Gr1m-0

- https://www.youtube.com/watch?v=UffM5Gr1m-0&ab\_channel=Crossplane

- https://betterprogramming.pub/crossplane-and-beyond-36a7cd855ce0

- https://blog.argoproj.io/using-argo-cd-with-vclusters-5df53d1c51ce

https://www.youtube.com/watch?v=JqkwOWYcx2w&ab\_channel=CNCF%5BCloudNativeComputingFoundation%5D

https://www.youtube.com/watch?v=9qRF-BjoAeU&ab\_channel=CNCF%5BCloudNativeComputingFoundation%5D

Why gitops?

https://www.youtube.com/watch?v=l5pVzP6wsP0&ab\_channel=CNCF%5BCloudNativeComputingFoundation%5D

https://opengitops.dev/

"Reconciliation refers to ensuring that a given state (e.g. application running in the cluster, infrastructure) matches a desired state declaratively defined somewhere (e.g. a Git repository)." [[link](https://fluxcd.io/flux/concepts/#reconciliation)]

idempotent: Idempotence, in programming and mathematics, is a property of some operations such that no matter how many times you execute them, you achieve the same result.

SSO login in argocd.

https://stackoverflow.com/questions/61365202/nginx-ingress-service-ingress-nginx-controller-admission-not-found

`argocd argocd-server-5b8c45c484-bhzmm 0/1 CrashLoopBackOff 2 (2m53s ago) 21m`

"Crossplane is a CNCF project that aims to unify all APIs in platform engineering based on Kubernetes API by leveraging Kubernetes controllers."[[s]](https://cloudnativesimplified.substack.com/p/crossplane-unifying-platform-engineering)

terraform to crossplane https://github.com/upbound/upjet

It is called Kubernetes objects and not Kubernetes resources.

//TODO

- find link related to argo-image-updater

- Skriv Abstract

- Skriv Conclusion

- Overvej om Challenges with managing state with terraform burde skrives om til 3 punkter.

- No preview compared to terraform.

- skriv afsnit om hvor ustabil GKE er.

- Apps of Apps pattern.

- husk link til repo.

1. <https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac)> [↑](#footnote-ref-1)
2. <https://about.gitlab.com/topics/gitops> [↑](#footnote-ref-2)
3. <https://www.oracle.com/cloud/cloud-native/what-is-cloud-native/> [↑](#footnote-ref-3)
4. <https://www.cncf.io/> [↑](#footnote-ref-4)
5. <https://cloud.google.com/kubernetes-engine> [↑](#footnote-ref-5)
6. <https://matthewpalmer.net/kubernetes-app-developer/articles/how-does-kubernetes-use-etcd.html> [↑](#footnote-ref-6)
7. <https://www.learnsteps.com/how-kubernetes-works-on-a-reconciler-pattern> [↑](#footnote-ref-7)
8. <https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#customresourcedefinitions> [↑](#footnote-ref-8)
9. <https://www.eficode.com> [↑](#footnote-ref-9)
10. <https://www.eficode.com/blog/outgrowing-terraform-and-adopting-control-planes> [↑](#footnote-ref-10)
11. <https://www.eficode.com/blog/outgrowing-terraform-and-adopting-control-planes> [↑](#footnote-ref-11)
12. <https://crossplane.io/> [↑](#footnote-ref-12)
13. <https://kubernetes.io/docs/concepts/overview/components/#kube-apiserver> [↑](#footnote-ref-13)
14. <https://kubernetes.io/docs/concepts/overview/components/> [↑](#footnote-ref-14)
15. <https://ubuntu.com/blog/declarative-vs-imperative-devops-done-right> [↑](#footnote-ref-15)
16. <https://ubuntu.com/blog/declarative-vs-imperative-devops-done-right> [↑](#footnote-ref-16)
17. <https://ubuntu.com/blog/declarative-vs-imperative-devops-done-right> [↑](#footnote-ref-17)
18. <https://edward-huang.com/distributed-system/programming/2021/02/28/why-you-should-always-design-your-api-declaratively-instead-of-imperative/> [↑](#footnote-ref-18)
19. <https://ubuntu.com/blog/declarative-vs-imperative-devops-done-right> [↑](#footnote-ref-19)
20. <https://edward-huang.com/distributed-system/programming/2021/02/28/why-you-should-always-design-your-api-declaratively-instead-of-imperative/> [↑](#footnote-ref-20)
21. <https://www.hashicorp.com/resources/the-story-of-hashicorp-terraform-with-mitchell-hashimoto> [↑](#footnote-ref-21)
22. <https://www.terraform.io/intro> [↑](#footnote-ref-22)
23. <https://www.varonis.com/blog/what-is-terraform> [↑](#footnote-ref-23)
24. <https://www.terraform.io/intro> [↑](#footnote-ref-24)
25. <https://www.varonis.com/blog/what-is-terraform> [↑](#footnote-ref-25)
26. <https://www.terraform.io/intro> [↑](#footnote-ref-26)
27. <https://www.terraform.io/cdktf/concepts/providers> [↑](#footnote-ref-27)
28. <https://adambrodziak.pl/terraform-is-terrible> [↑](#footnote-ref-28)
29. <https://blog.gruntwork.io/how-to-manage-terraform-state-28f5697e68fa> [↑](#footnote-ref-29)
30. <https://spacelift.io/blog/terraform-state> [↑](#footnote-ref-30)
31. <https://www.pmg.com/blog/a-beginners-experience-with-terraform> [↑](#footnote-ref-31)
32. <https://adambrodziak.pl/terraform-is-terrible> [↑](#footnote-ref-32)
33. <https://developer.hashicorp.com/terraform/language/settings/backends/configuration> [↑](#footnote-ref-33)
34. <https://adambrodziak.pl/terraform-is-terrible> [↑](#footnote-ref-34)
35. <https://www.linkedin.com/pulse/terraform-state-remote-storage-s3-locking-dynamodb-oramu-/?trk=pulse-article> [↑](#footnote-ref-35)
36. <https://angelo-malatacca83.medium.com/aws-terraform-s3-and-dynamodb-backend-3b28431a76c1> [↑](#footnote-ref-36)
37. <https://www.terraform.io/cli/state/recover> [↑](#footnote-ref-37)
38. <https://www.terraform.io/language/state/locking> [↑](#footnote-ref-38)
39. <https://developer.hashicorp.com/terraform/cli/state/recover> [↑](#footnote-ref-39)
40. <https://learn.hashicorp.com/tutorials/terraform/gke> [↑](#footnote-ref-40)
41. <https://blog.crossplane.io/crossplane-vs-terraform> [↑](#footnote-ref-41)
42. <https://adambrodziak.pl/terraform-is-terrible> [↑](#footnote-ref-42)
43. <https://blog.upbound.io/outgrowing-terraform-and-migrating-to-crossplane> [↑](#footnote-ref-43)
44. <https://www.runatlantis.io/> [↑](#footnote-ref-44)
45. <https://www.eficode.com/blog/outgrowing-terraform-and-adopting-control-planes> [↑](#footnote-ref-45)
46. <https://github.com/eficode-academy/quotes-flask> [↑](#footnote-ref-46)
47. <https://crossplane.io/docs/v1.10/reference/configure.html> [↑](#footnote-ref-47)
48. <https://nandovillalba.medium.com/why-i-think-gcp-is-better-than-aws-ea78f9975bda> [↑](#footnote-ref-48)
49. <https://crossplane.io/docs/v1.10/reference/configure.html> [↑](#footnote-ref-49)
50. <https://argoproj.github.io/cd> [↑](#footnote-ref-50)
51. <https://www.opsmx.com/what-is-argocd> [↑](#footnote-ref-51)
52. <https://www.cncf.io/projects/argo/> [↑](#footnote-ref-52)
53. <https://www.cncf.io/projects/crossplane> [↑](#footnote-ref-53)
54. <https://crossplane.io/docs/v1.10/concepts/providers.html> [↑](#footnote-ref-54)
55. <https://www.digitalocean.com/blog/announcing-the-digitalocean-crossplane-provider> [↑](#footnote-ref-55)
56. <https://kubeform.com> [↑](#footnote-ref-56)
57. <https://github.com/.crossplane-contrib/provider-terraform> [↑](#footnote-ref-57)
58. <https://github.com/kubernetes-sigs/external-dns/tree/master/charts/external-dns> [↑](#footnote-ref-58)
59. <https://github.com/kubernetes-sigs/external-dns/blob/master/docs/tutorials/gke.md> [↑](#footnote-ref-59)
60. <https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners> [↑](#footnote-ref-60)
61. <https://github.com/kubernetes-sigs/kustomize/blob/53cc76fe43c91493d093c2832cc61425aa2972e1/README.md#2-create-variants-using-overlays> [↑](#footnote-ref-61)
62. <https://argo-cd.readthedocs.io/en/stable/operator-manual/cluster-bootstrapping/#app-of-apps-pattern> [↑](#footnote-ref-62)
63. <https://github.com/crossplane/crossplane/blob/master/docs/concepts/providers.md#configuring-providers> [↑](#footnote-ref-63)
64. <https://crossplane.io/docs/v1.10/concepts/providers.html> [↑](#footnote-ref-64)
65. <https://github.com/dag-andersen/k8s-ucp-core-gitops/tree/main/gcp-clusters> [↑](#footnote-ref-65)
66. <https://github.com/crossplane-contrib/provider-argocd/issues/13> [↑](#footnote-ref-66)
67. <https://www.vaultproject.io> [↑](#footnote-ref-67)
68. <https://github.com/dag-andersen/k8s-ucp-bootstrap> [↑](#footnote-ref-68)
69. <https://learn.hashicorp.com/tutorials/terraform/gke> [↑](#footnote-ref-69)
70. <https://www.weave.works/blog/gitops-fully-automated-ci-cd-pipelines> [↑](#footnote-ref-70)
71. <https://github.com/argoproj-labs/argocd-image-updater/issues/496> [↑](#footnote-ref-71)
72. <https://fluxcd.io/flux/components/image/imageupdateautomations/> [↑](#footnote-ref-72)
73. <https://www.economize.cloud/resources/gcp/pricing/e2/e2-medium> [↑](#footnote-ref-73)
74. <https://github.com/crossplane/crossplane/issues/1805> [↑](#footnote-ref-74)
75. <https://blog.upbound.io/outgrowing-terraform-and-migrating-to-crossplane/> [↑](#footnote-ref-75)
76. <https://www.eficode.com/blog/outgrowing-terraform-and-adopting-control-planes> [↑](#footnote-ref-76)
77. <https://www.cncf.io/blog/2022/07/01/devops-vs-sre-vs-platform-engineering-the-gaps-might-be-smaller-than-you-think/> [↑](#footnote-ref-77)
78. <https://crossplane.io/docs/v1.9/concepts/composition.html#composite-resources> [↑](#footnote-ref-78)
79. <https://youtu.be/UffM5Gr1m-0?t=843> [↑](#footnote-ref-79)
80. <https://youtu.be/UffM5Gr1m-0?t=769> [↑](#footnote-ref-80)
81. <https://amazic.com/dont-waste-your-money-on-unused-cloud-resources> [↑](#footnote-ref-81)
82. <https://www.youtube.com/watch?v=Od7e6hqis7A&ab_channel=PrismaCloudbyPaloAltoNetworks> [↑](#footnote-ref-82)
83. <https://www.eficode.com/blog/outgrowing-terraform-and-adopting-control-planes> [↑](#footnote-ref-83)
84. <https://blog.upbound.io/outgrowing-terraform-and-migrating-to-crossplane> [↑](#footnote-ref-84)
85. <https://en.wikipedia.org/wiki/Command_(computing)> [↑](#footnote-ref-85)
86. <https://www.eficode.com/blog/outgrowing-terraform-and-adopting-control-planes> [↑](#footnote-ref-86)
87. <https://www.cncf.io/projects/> [↑](#footnote-ref-87)
88. <https://github.com/crossplane-contrib/provider-digitalocean/issues/55> [↑](#footnote-ref-88)
89. <https://github.com/argoproj/argo-cd/issues/8128> [↑](#footnote-ref-89)
90. <https://github.com/argoproj/argo-cd/issues/4651> [↑](#footnote-ref-90)
91. <https://github.com/crossplane-contrib/provider-gcp/issues/397#issuecomment-948758190> [↑](#footnote-ref-91)