

# Utilizing Kubernetes as a universal control plane

Dag Bjerre Andersen  
daga@itu.dk

IT University of Copenhagen  
15-12-2022  
KIP22501PE

## Abstract

Control planes are a new paradigm in DevOps and infrastructure management. This paper explains the concept of control planes and how to utilize Kubernetes as a universal control plane for handling both internal resources (e.g., containers) and external resources (e.g., cloud resources) and discusses the challenges and implications. This paper is done in collaboration with the DevOps consultancy Eficode, and the goal of the paper is to provide them with a better understanding of the challenges and implications of transitioning to control plane-based infrastructure-management. A prototype of a universal control plane is designed and developed. To demonstrate the prototype's multi-cloud multi-environment capabilities, an application developed by Eficode is deployed on infrastructure managed by the universal control plane. The paper discusses some of the challenges and limitations observed when designing, developing, and using Kubernetes as a universal control plane. The conclusion is that control plane-based infrastructure-management comes with some great benefits (e.g., a highly streamlined configuration management) but also introduces significant downsides (e.g., limited testing options) depending on the specific implementation and tools chosen.

## Table of Contents

<i>Abstract</i>	<i>1</i>
<i>1. Dictionary and abbreviations</i>	<i>3</i>
<i>2. Introduction</i>	<i>3</i>
<i>3. Motivation</i>	<i>4</i>
<i>4. Defining Control Plane</i>	<i>4</i>
<i>5. Declarative vs Imperative</i>	<i>5</i>
5.1. Terraform	5
5.2. From Terraform to control planes	7
<i>6. Demonstration Application</i>	<i>7</i>
<i>7. Implementation</i>	<i>8</i>
7.1. Technologies and tools used	9
7.2. ArgoCD and Crossplane together	10
7.3. Managing internal state with ArgoCD	11
7.4. Managing External State with Crossplane	13
7.5. Distributing Secrets	14
<i>8. Demonstration Application running in Google Cloud</i>	<i>16</i>
8.1. Use in practice	18
<i>9. Observations</i>	<i>21</i>
9.1. State management with Crossplane compared to Terraform	21
9.2. No preview with Crossplane	22
9.3. Multiple Core Clusters	23
9.4. Single interface	24
9.5. Platform Engineering	24
9.6. Modifying and extending the Core Cluster	24
9.7. Better visibility and confidence in the system	25
9.8. Declaring the entire infrastructure	25
9.9. Bootstrapping Problem	26
9.10. Additional Costs	26
9.11. Build pipelines	27
9.12. Maturity level	27
<i>10. Conclusion</i>	<i>28</i>

## 1. Dictionary and abbreviations

List of the 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<sup>1</sup>.
- **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 as a single source of truth for managing infrastructure and application deployment<sup>2</sup>.
- **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<sup>3</sup>.
- **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<sup>4</sup>.
- **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<sup>5</sup>.
- **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<sup>6</sup>. One state exists per Kubernetes cluster.
- **Internal State:** In this paper Internal state refers to declared Kubernetes objects reflecting objects managed inside Kubernetes (e.g., *Pods*, *Service*, and *Ingress*).
- **External State:** In this paper External state 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 become the desired state<sup>7</sup>.

## 2. Introduction

Since the release of Kubernetes in 2014<sup>8</sup>, an increasing number of CNCF projects built as control planes (FluxCD, ArgoCD, Crossplane, etc.) have emerged. The concept of a control plane is built on the idea of a service that watches a declared state and makes 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 of Kubernetes objects, and different services (control plane components) watch this state and make sure the actual state reflects the declared state. Kubernetes can be extended and used as a platform for control planes because control planes can call Kubernetes' API server to manage the state of their own custom Kubernetes objects (CRD)<sup>9</sup>. 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)

<sup>1</sup> Red Hat, "What is Infrastructure as Code (IaC)?", May 11, 2022, Retrieved: 02/12/2022, <https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac>

<sup>2</sup> GitLab, "What is GitOps?", <https://about.gitlab.com/topics/gitops>

<sup>3</sup> Oracle, "What is Cloud Native?", June 18, 2021, Retrieved: 05/12/2022, <https://www.oracle.com/cloud/cloud-native/what-is-cloud-native/>

<sup>4</sup> Cloud Native Computing Foundation's website, Retrieved: 06/12/2022, <https://www.cncf.io/>

<sup>5</sup> Google Cloud, "A scalable and automated Kubernetes service", Retrieved: 05/12/2022, <https://cloud.google.com/kubernetes-engine>

<sup>6</sup> Matthew Palmer, "How Does Kubernetes Use etcd?", Retrieved: 12/12/2022, <https://matthewpalmer.net/kubernetes-app-developer/articles/how-does-kubernetes-use-etcd.html>

<sup>7</sup> Gaurav Yadav, "How Kubernetes works on reconciler pattern", February 21, 2020, Retrieved: 06/12/2022, <https://www.learnsteps.com/how-kubernetes-works-on-a-reconciler-pattern>

<sup>8</sup> Kubernetes Blog, "The History of Kubernetes & the Community Behind It", July 20, 2018, Retrieved: 13/12/2022, <https://kubernetes.io/blog/2018/07/20/the-history-of-kubernetes-the-community-behind-it/>

<sup>9</sup> Kubernetes Docs, "Extending Kubernetes", October 30, 2022, Retrieved: 12/12/2022, <https://kubernetes.io/docs/concepts/extend-kubernetes/>

have emerged. This means that control planes like Crossplane can be used to replace infrastructure tools like Terraform by controlling the full life cycle of cloud resources from within 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 implements a universal control plane for handling both internal state (e.g., containers) and external state (e.g., cloud resources) and discusses the challenges and implications.

This paper is done in collaboration with Eficode, and the goal of the paper is to provide them with a better understanding of the challenges and implications of transitioning to control plane-based infrastructure-management.

It is assumed that the reader is familiar with Kubernetes, its concepts, and its terminology. Furthermore, it is assumed that the reader is familiar with Terraform, how it manages its state, and how Terraform providers work.

### 3. Motivation

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<sup>10</sup>.

Eficode has stated their concern about how Terraform does not scale well for big organizations in their blog post: “Outgrowing Terraform and adopting control planes”<sup>11</sup>. 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<sup>11</sup>. Many of the technologies/tools leveraging the concept of control planes (e.g., Crossplane) 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.

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

### 4. 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 watch 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<sup>12</sup>. Control planes follow the Reconciler pattern. Control planes are self-healing, and they automatically correct drift<sup>13</sup>.
- **Control plane (as a Kubernetes controller)** refers to an application implemented as a control plane (the concept). Examples of this are Kubernetes' kube-scheduler (that schedules *Pods* based on a declared state)<sup>14</sup> and ArgoCD (a GitOps tool introduced in this paper).

---

<sup>10</sup> Eficode's website, Retrieved: 06/12/2022, <https://www.eficode.com>

<sup>11</sup> Michael Vittrup Larsen, Eficode, “Outgrowing Terraform and adopting control planes”, February 9, 2022, Retrieved: 06/12/2022, <https://www.eficode.com/blog/outgrowing-terraform-and-adopting-control-planes>

<sup>12</sup> FluxCD Docs, “Reconciliation”, Last modified: 30/08/2022, Retrieved: 06/12/2022, <https://fluxed.io/flux/concepts/#reconciliation>

<sup>13</sup> Crossplane's website, Retrieved: 13/11/2022, <https://crossplane.io/>

<sup>14</sup> Kubernetes Docs, “kube-apiserver”, Retrieved: 06/12/2022, <https://kubernetes.io/docs/concepts/overview/components/#kube-apiserver>

- **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<sup>15</sup>. This node is often referred to as the control plane (node).

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 a platform for hosting multiple control planes (as Kubernetes Controllers) that manage 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 do not refer to a control plane *as a node*.

## 5. Declarative vs Imperative

An automated system (like the universal control plane built in this paper) can be designed and implemented in two different paradigms: Declarative and imperative<sup>16</sup>. In the declarative paradigm, developers focus on *what* the desired state of the system should be. In the imperative paradigm, developers focus on *how* the desired state is created.

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"*<sup>16</sup>. The developer must carefully plan every step and the sequence in which they are executed. Suppose a change in the setup must 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"*<sup>16</sup>

### 5.1. Terraform

A very popular tool that is based on the idea of IaC and the declarative paradigm is Terraform. Terraforms' popularity started in 2016-2017 and has been growing ever since<sup>17</sup>.

Terraform let users define both cloud and on-premises resources in configuration files that they can version, reuse, and share<sup>18</sup>. Terraform is responsible for handling the entire lifecycle of the resources: From creation to deletion<sup>18</sup>.

One of the main use cases is often provisioning infrastructure on cloud providers<sup>19</sup>. Terraform is cloud-agnostic and can provision cloud resources across all the big cloud providers (e.g., AWS, Microsoft Azure, and GCP)<sup>19</sup>. 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"*<sup>18</sup>.

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.

<sup>15</sup> Kubernetes Docs, "Kubernetes Components", Retrieved: 06/12/2022, <https://kubernetes.io/docs/concepts/overview/components/>

<sup>16</sup> Tytus Kurek, Ubuntu Blog, "Declarative vs Imperative: DevOps done right", August 6, 2019, Retrieved: 06/12/2022, <https://ubuntu.com/blog/declarative-vs-imperative-devops-done-right>

<sup>17</sup> HashiCorp, "The Story of HashiCorp Terraform with Mitchell Hashimoto", July 13, 2021, Retrieved: 06/12/2022, <https://www.hashicorp.com/resources/the-story-of-hashicorp-terraform-with-mitchell-hashimoto>

<sup>18</sup> Terraform, "What is Terraform?", Retrieved: 06/12/2022, <https://www.terraform.io/intro>

<sup>19</sup> David Harrington, Inside Out Security Blog, "What is Terraform: Everything You Need to Know", March 30, 2022, Retrieved: 06/12/2022, <https://www.varonis.com/blog/what-is-terraform>

### 5.1.1. 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<sup>20 21 22</sup>. 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<sup>23</sup>.

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.

### 5.1.2. Challenges with managing Terraform state

Terraform state is inherently difficult to manage<sup>20</sup>. Just because Terraform state is stored in a remote place does not mean multiple people can work on it simultaneously.

When storing the state in a remote place, users need to specify a so-called *Backend Configuration*<sup>24</sup>, 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)<sup>20 25</sup>. This, again, just adds to the complexity and creates an even bigger barrier for getting started with Terraform if users want to make sure their IaC configuration is safe to use.

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 a user must `force-unlock` the state<sup>26</sup>. This gets even worse if a `force-unlock` is executed while another process is in the middle of applying a new Terraform configuration<sup>27</sup>. This can result in multiple simultaneous writers, which can result in the state being corrupted<sup>26</sup>.

Furthermore, even though there may be a lock that makes sure that there is no race condition while applying changes, 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<sup>28</sup>. During this time, no other entity can apply changes to the configuration<sup>29</sup>. This is an even bigger problem if a company has a monolithic infrastructure configuration with a lot of dependencies and all components are stored in the same Terraform state. So, for example 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.

Another challenge with Terraform is that Terraform's state can easily go out of sync. This is called *configuration drift* 20<sup>30</sup>. If the `terraform apply`-command is not run regularly, the actual state can drift away from the declared 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 can cause issues when later mutating the Terraform state because Terraform will then revert the manually made changes<sup>20</sup>.

<sup>20</sup> Adam Brodziak, Adam on DevOps, "Terraform is terrible", May 27, 2021, Retrieved: 06/12/2022, <https://adambroziak.pl/terraform-is-terrible>

<sup>21</sup> Yevgeniy Brikman, Medium, "How to manage Terraform state", Oct 3, 2016, Retrieved: 06/12/2022, <https://blog.gruntwork.io/how-to-manage-terraform-state-28f5697e68fa>

<sup>22</sup> Jack Roper, spacelift, "Managing Terraform State – Best Practices & Examples", August 11, 2022, Retrieved: 06/12/2022, <https://spacelift.io/blog/terraform-state>

<sup>23</sup> PMG, "A beginner's experience with terraform", December 20, 2018, Retrieved: 06/12/2022, <https://www.pmg.com/blog/a-beginners-experience-with-terraform>

<sup>24</sup> Terraform, "Backend Configuration", Retrieved: 06/12/2022, <https://developer.hashicorp.com/terraform/language/settings/backends/configuration>

<sup>25</sup> Angelo Malatucca, Medium, "AWS Terraform S3 and dynamoDB backend", July 20, 2020, Retrieved: 06/12/2022, <https://angelo-malatucca83.medium.com/aws-terraform-s3-and-dynamodb-backend-3b28431a76c1>

<sup>26</sup> Terraform, "Recovering from State Disasters", Retrieved: 06/12/2022, <https://developer.hashicorp.com/terraform/cli/state/recover>

<sup>27</sup> Terraform, "State Locking", Retrieved: 06/12/2022, <https://www.terraform.io/language/state/locking>

<sup>28</sup> Terraform, "Provision a GKE Cluster (Google Cloud)", Retrieved: 06/12/2022, <https://learn.hashicorp.com/tutorials/terraform/gke>

<sup>29</sup> Nic Cope, Crossplane's Blog, "Crossplane vs Terraform", March 2, 2021, Retrieved: 06/12/2022, <https://blog.crossplane.io/crossplane-vs-terraform>

<sup>30</sup> Bassam Tabbara, Upbound, "Outgrowing Terraform — and Migrating to Crossplane", June 16, 2021, Retrieved: 06/12/2022, <https://blog.upbound.io/outgrowing-terraform-and-migrating-to-crossplane>

On the other hand, 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 because a service needed to use a new IP address, and it needed to be fixed immediately. This would not be possible if Terraform did not allow *configuration drift*.

## 5.2. From Terraform to control planes

Configuration drift with Terraform can be avoided by using an automated tool (e.g., Atlantis<sup>31</sup>) 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, 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 continuously reconcile its current state to match the desired state. 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 inside Kubernetes. A control plane running on Kubernetes could automatically synchronize the actual infrastructure with the declared state.

*Crossplane*, Google's *Config Connector*, and AWS' *Controllers for Kubernetes* are control planes that reconcile 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. Crossplane will be described in greater detail in section: 7.4. *Managing External State with Crossplane*.

## 6. Demonstration Application

To verify and demonstrate the capabilities of the universal control plane that I have built (presented in section: 7. *Implementation*), I need a demonstration application that runs on infrastructure managed by the universal control plane.

To do this, I have used Eficode's public `quotes-flask`<sup>32</sup> application that they use for educational purposes. It is a simple application consisting of a frontend, a backend, and a database. I refer to this application as "Quote App".

The 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, 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 (as seen in Figure 1). Besides that, I have not changed the overall architecture.

---

<sup>31</sup> Atlantis' website, Retrieved: 06/12/2022, <https://www.runatlantis.io/>

<sup>32</sup> Eficode Academy, GitHub, "quotes-flask", Retrieved: 06/12/2022, <https://github.com/eficode-academy/quotes-flask>



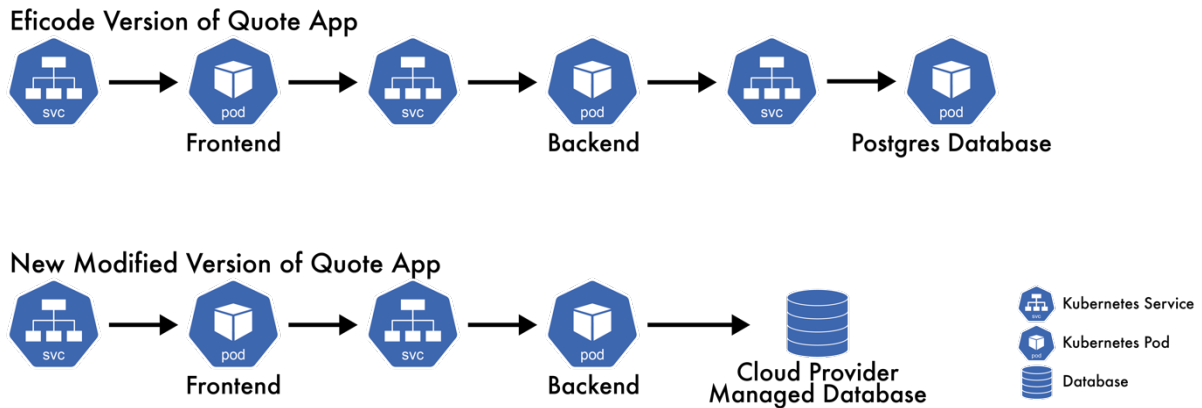


Figure 1: A diagram showing Eficode's version of Quote App and my modified version of Quote App.

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

The company may want multiple environments like *production*, *staging*, and *development*, and they may leverage cloud services across multiple cloud providers. Therefore, the Quote App runs on multiple isolated environments on one cloud provider and accesses a database on another cloud provider (visualized in Figure 2).

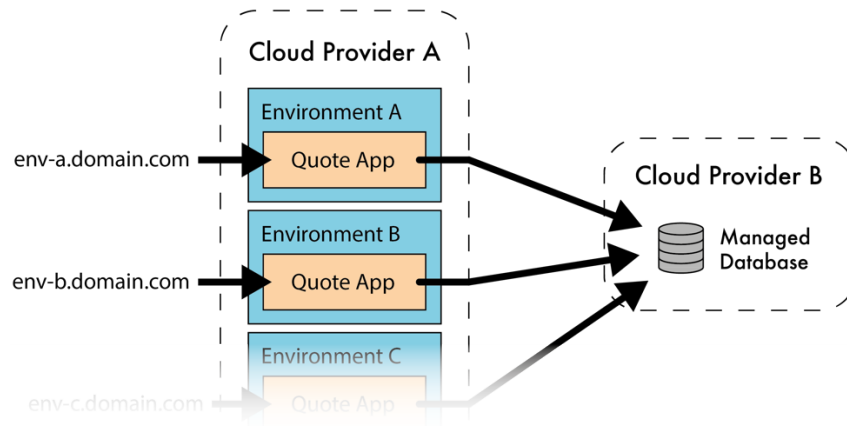


Figure 2: A visualization of the Quote App (demonstration application) running in multiple environments.

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.

## 7. Implementation

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.

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 App Cluster environments.



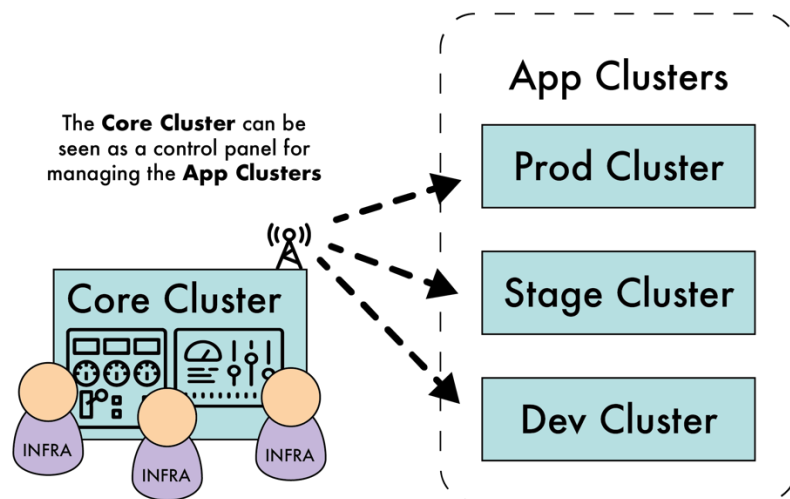


Figure 3: An illustration of how the infrastructure team manages App Clusters from the Core Cluster.

Only the infrastructure teams are supposed to interact with the 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 3.

## 7.1. Technologies and tools used

### Amazon Web Services

AWS was chosen as the cloud provider for a simple managed database. AWS is one of the officially supported providers for Crossplane<sup>33</sup>. Both Microsoft Azure and GCP would be potential alternatives to AWS in this implementation since they are also officially supported providers.

### Google Cloud Platform

GCP was chosen as the main provider for cloud-hosted Kubernetes clusters and Networking because GCP is less complex to use than AWS<sup>34</sup>. GCP is one of the officially supported providers for Crossplane<sup>33</sup>. Both Microsoft Azure and AWS would be potential alternatives to GCP since they are also officially supported providers.

### 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<sup>28</sup>). There are many different tools for running Kubernetes locally, and many of them would probably work for this implementation, but Kind was chosen 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.

<sup>33</sup> Crossplane Docs, "Configure Your Cloud Provider Account", Retrieved: 06/12/2022, <https://docs.crossplane.io/v1.10/reference/configure/>

<sup>34</sup> Fernando Villalba, Medium, "Why I think GCP is better than AWS", May 9, 2020, Retrieved: 06/12/2022, <https://nandovillalba.medium.com/why-i-think-gcp-is-better-than-aws-ea78f9975bda>

## ArgoCD

ArgoCD is a declarative continuous delivery GitOps tool that is built as a control plane and runs inside Kubernetes<sup>35</sup>. 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, or a Helm Chart. Quote App is an example of an Application deployed with Kustomize and managed by an ArgoCD. When changes are made to the ArgoCD Application's configuration in Git, ArgoCD will compare it with the configurations of the running ArgoCD Application to bring the desired and actual state into sync<sup>36</sup>. 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). 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 *Graduated project* maturity level<sup>37</sup>. ArgoCD's logo can be seen in Figure 4.



Figure 4:  
ArgoCD's Logo

## Crossplane

Crossplane is a control plane that runs inside Kubernetes, which ensures 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 resources managed by Crossplane are declared in manifests stored in Kubernetes. Crossplane was accepted to CNCF on June 23, 2020, and is at the *Incubating project* maturity level<sup>38</sup>. Crossplane's logo can be seen in Figure 5.



Figure 5:  
Crossplane's Logo

Just like Terraform, Crossplane has the concept of *providers*<sup>39</sup>. 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 Microsoft Azure as cloud providers. A DigitalOcean provider is also in active development<sup>40</sup>.

## 7.2. ArgoCD and Crossplane together

The Core Cluster uses Crossplane to provision cloud resources and uses ArgoCD to deploy and manage all services 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 state (e.g., provisioning cloud resources) - combined, they can be used as a universal control plane for managing multi-cloud multi-environment infrastructure.

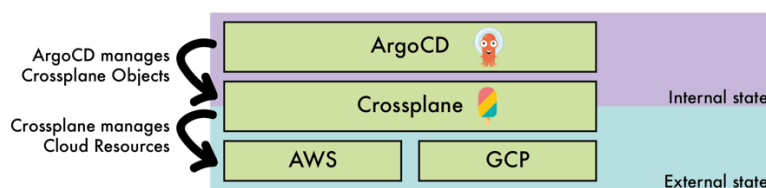


Figure 6: A visualization of how ArgoCD manages Crossplane Objects and Crossplane manages Cloud Resources.

Crossplane itself and the Kubernetes objects used by Crossplane are all declared in manifests, checked into Git, and synced by ArgoCD. Figure 6 is a visualization of how ArgoCD and Crossplane work together to provision and manage cloud resources.

<sup>35</sup> Argo Project, ArgoCD's website, Retrieved: 06/12/2022, <https://argoproj.github.io/cd>

<sup>36</sup> OpsMx, "What is ArgoCD?", Retrieved: 06/12/2022, <https://www.opsmx.com/what-is-argocd>

<sup>37</sup> CNCF, "Argo", Retrieved: 06/12/2022, <https://www.cncf.io/projects/argo/>

<sup>38</sup> CNCF, "Crossplane", Retrieved: 08/12/2022, <https://www.cncf.io/projects/crossplane>

<sup>39</sup> Crossplane Docs, "Providers", Retrieved: 10/12/2022, <https://docs.crossplane.io/v1.10/concepts/providers/>

<sup>40</sup> Kim Schlesinger, DigitalOcean Blog, "Announcing the DigitalOcean Crossplane Provider", May 4, 2022, Retrieved: 06/12/2022, <https://www.digitalocean.com/blog/announcing-the-digitalocean-crossplane-provider>

### 7.3. Managing internal state with ArgoCD

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

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. There would be multiple endpoints and multiple ArgoCD-profiles/-credentials for each ArgoCD instance running in each cluster, which may not be desirable if a company runs 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.

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.

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 and delete the two Applications independently. Based on this idea of separation, I have chosen to structure my Applications as illustrated in Figure 7.

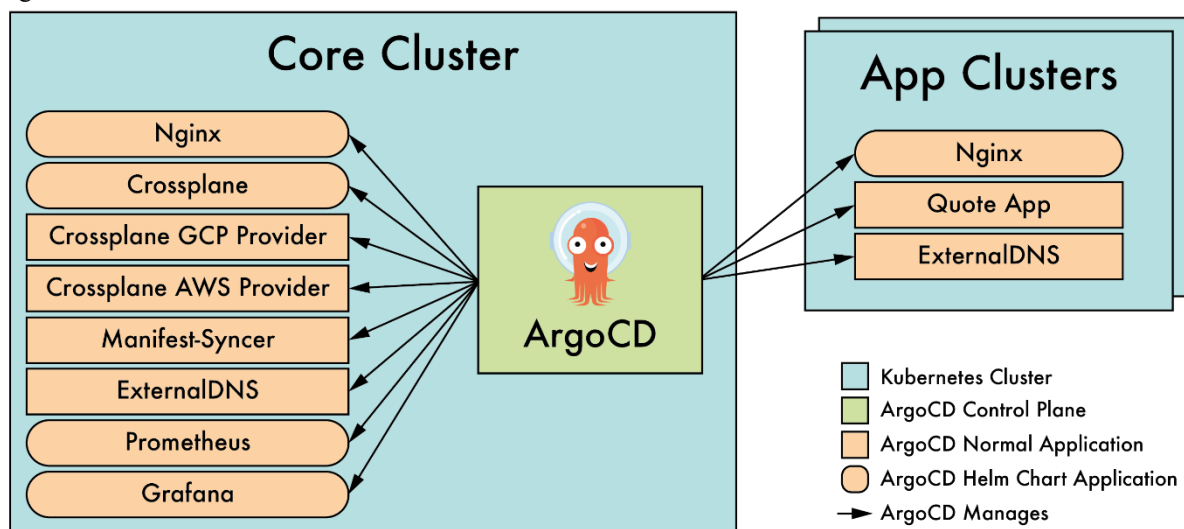


Figure 7: An illustration of what packages and services ArgoCD install and on which cluster.

As seen in Figure 7, besides Crossplane and Quote App, ArgoCD also installs other packages/services:

#### Nginx

To call the Quote App's frontend from outside Kubernetes, I need to set up Nginx. 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 to call the endpoints inside Kubernetes from outside Kubernetes<sup>41</sup>.

#### ExternalDNS

ExternalDNS synchronizes exposed Kubernetes *Services* and *Ingresses* with DNS providers<sup>42</sup>. To automatically create DNS records in GCP when new *Ingress* objects are created in GKE, ExternalDNS needs to be deployed and set up in the cluster<sup>43</sup>.

<sup>41</sup> Ambassador, "Understanding Kubernetes Ingress", Retrieved: 06/12/2022, <https://www.getambassador.io/learn/kubernetes-ingress>

<sup>42</sup> Kubernetes SIGs, GitHub, "ExternalDNS", Retrieved: 06/12/2022, <https://github.com/kubernetes-sigs/external-dns/tree/master/charts/external-dns>

<sup>43</sup> Kubernetes SIGs, GitHub, "Setting up ExternalDNS on Google Kubernetes Engine", Retrieved: 06/12/2022, <https://github.com/kubernetes-sigs/external-dns/blob/master/docs/tutorials/gke.md>

### Prometheus and Grafana

Prometheus and Grafana is an open-source monitoring stack. The stack makes it possible to observe, e.g., the memory usage of the Core Cluster. 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.

### 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: 7.5. *Distributing Secrets*.

#### 7.3.1. Eventual consistency

At its core, all ArgoCD does is that it reads from a given branch on a given Git 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*).

Provisioning and deploying infrastructure through sequential steps in a script can take a long time because it runs sequentially and not in parallel. With eventual consistency in Kubernetes, multiple steps can run simultaneously, and they will be executed eventually when the steps 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 scheduled eventually when the right conditions are present (e.g., enough CPU).

Applying Kubernetes resources with ArgoCD works the same way. If ArgoCD cannot deploy an ArgoCD Application (because some dependency may be missing), it will just automatically try again a minute later. This means infrastructure teams can apply their entire infrastructure at once with ArgoCD, and ArgoCD will make sure everything will be deployed with eventual consistency, even though there may be broken dependencies temporarily in the process. An example of this is that the Quote App's 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's frontend's Ingress configuration until Nginx eventually exists in the cluster and the deployment succeeds.

#### 7.3.2. 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 (`k8s-ucp-core-gitops`), a second ArgoCD synced repository for syncing with the App Clusters (`k8s-ucp-app-gitops`), and finally, a general repository with code and scripts for bootstrapping the system (`k8s-ucp-bootstrap`).

The repositories can be found on GitHub:

- `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>
- `k8s-ucp-bootstrap`: <https://github.com/dag-andersen/k8s-ucp-bootstrap>

“`k8s-ucp`” stands for *Kubernetes Universal Control Plane*

The bootstrapping repository mainly contains scripts for starting the Core Cluster. The two other repositories only contain manifests synced with ArgoCD.

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

```
organization-root
├── k8s-ucp-bootstrap # Repository with bootstrapping scripts
├── k8s-ucp-gitops    # Repository containing all ArgoCD synced resources
│   ├── app-cluster
│   └── core-cluster
```

On the other hand, a company may not want everyone in the organization to have read-access to the configuration of all clusters and, therefore, may prefer the configuration of the Core Cluster and App Clusters to

be separate. The write-access would not be a problem because it can be solved by using, e.g., [CODEOWNERS](#) on GitHub<sup>44</sup>.

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

The folder structure of the Core Cluster and App Cluster repositories can be seen in Figure 8. Most resources that are applied by ArgoCD are built with Kustomize using the *base-overlay-pattern*<sup>45</sup>, which is why all the *base-* and *overlays-*folders exist. I use *App of Apps Pattern*<sup>46</sup> 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 (named *argo-bootstrap* in the code).

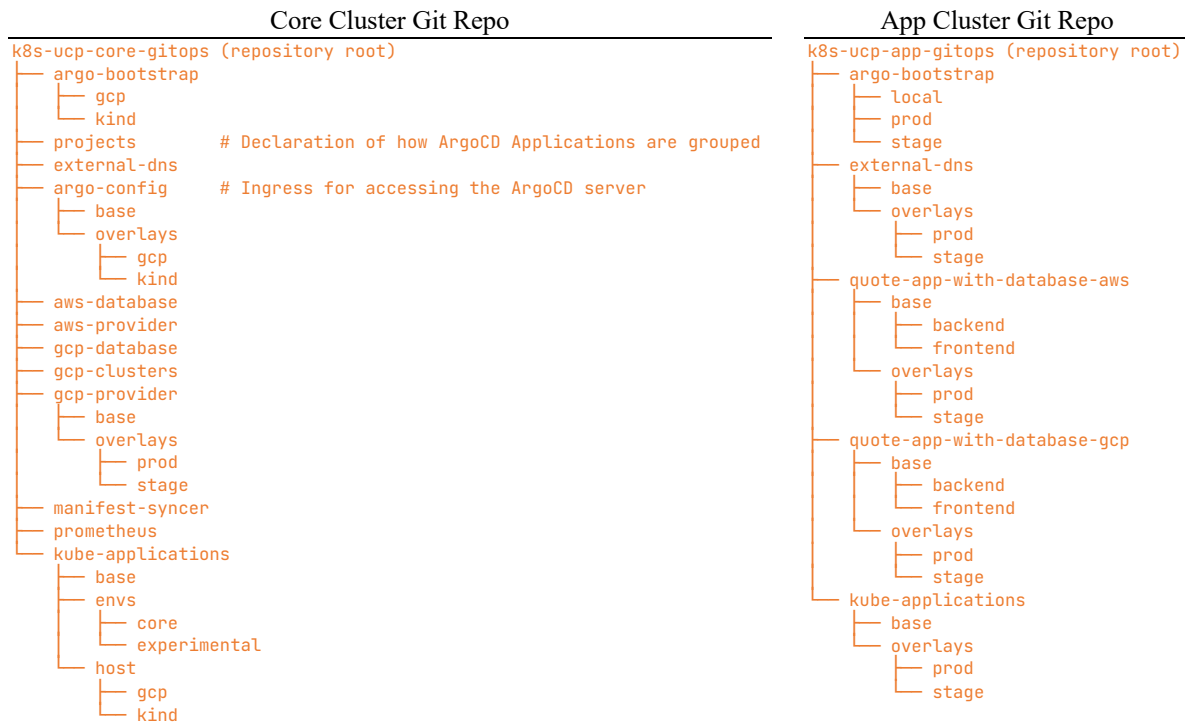


Figure 8: The folder structure of the Core Cluster and the App Cluster repositories

## 7.4. 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).

To authenticate with the cloud provider API, Crossplane needs to have access to credentials. In my case, it would be an *IAM User* for AWS and a *Service Account* for GCP<sup>47</sup>. How to generate the credentials is described in Crossplane's documentation<sup>48</sup>.

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.

<sup>44</sup> GitHub Docs, "About code owners", Retrieved: 02/12/2022, <https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners>

<sup>45</sup> Kubernetes SIGs, GitHub, "Create variants using overlays", Retrieved: 04/12/2022, <https://github.com/kubernetes-sigs/kustomize#2-create-variants-using-overlays>

<sup>46</sup> ArgoCD, "App Of Apps Pattern", Retrieved: 02/12/2022, <https://argo-cd.readthedocs.io/en/stable/operator-manual/cluster-bootstrapping/#app-of-apps-pattern>

<sup>47</sup> Crossplane, GitHub, "Configuring Providers", Retrieved: 10/12/2022, <https://github.com/crossplane/crossplane/blob/master/docs/concepts/providers.md#configuring-providers>

<sup>48</sup> Crossplane Docs, "Install & Configure", Retrieved: 11/12/2022, <https://docs.crossplane.io/v1.10/getting-started/install-configure/>

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), a `ProviderConfig` (specifying to use `aws-creds-secret` as authentication), and a `RDSInstance` (specifying the database properties) and apply them to a Kubernetes cluster with Crossplane installed<sup>39</sup>. The three resources can be seen in this code snippet:

```
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.

Just like with ArgoCD, a developer 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 that shows all the provisioned external resources.

Running Crossplane on a Core Cluster also decouples the external resources from the App Clusters. If Crossplane ran on each App Cluster and managed its own associated external resources, e.g., the staging cluster managing its own staging database, the developers would not be able to manage the staging database when the staging cluster was taken down temporarily. Therefore, it is better to manage all external infrastructure resources from a cluster that is always up and running.

An important detail is that when Crossplane creates a resource (e.g., a database instance), it stores the connection details in the cluster on which Crossplane is running within. The problem here is that credentials are often needed in the App Clusters (e.g., services running in the production environment need to connect to the production database). There are many ways to handle *Secrets*/credentials, but more on this in section: 7.5. *Distributing Secrets*.

## 7.5. Distributing Secrets

This section will describe in technical detail how I close the gap between ArgoCD and Crossplane. This section can be skipped if the reader 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, there exists 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`-GitHub Organization<sup>49</sup>, 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 using many different tools from the Kubernetes ecosystem that are not

---

<sup>49</sup> GitHub Issue, Retrieved: 01/12/2022, <https://github.com/crossplane-contrib/provider-argocd/issues/13>



necessarily meant to be used in combination with each other and do not have a native integration between them. Infrastructure teams may have to close these gaps themselves if they can't find an off-the-shelf component online (e.g., GitHub) that solves their 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<sup>50</sup> or GCP's Secret Manager<sup>51</sup>), where the credentials are stored when the database is created. 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 nonetheless, it 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 its default configuration and is controlled through *CustomResourceDefinitions*. If developers want a *Secret* to be automatically mirrored/copied from the Core Cluster to, e.g., the production cluster, they just create a manifest describing exactly that (see the following code snippet) and apply it to the Core Cluster.

```
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
```

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*.

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*-manifests are applied before the *manifest-syncer* is deployed. ArgoCD will fix the failing resources with eventual consistency.

### 7.5.1. Getting access to the App Clusters

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 containing kubeconfigs. This alone is not enough because it only gives read-access to the clusters. To gain write-access, it scans its host cluster for *Secrets* generated by ArgoCD with the label: *argocd.argoproj.io/secret-type=cluster*, and from these, it retrieves ArgoCD's access tokens to the App Clusters. The *manifest-syncer* combines the kubeconfigs and access tokens and gets write-access to the App Clusters. The *manifest-syncer* repeats this process every 10 seconds to continuously detect when new App Clusters are created.

<sup>50</sup> HashiCorp Vault's website, Retrieved: 12/12/2022, <https://www.vaultproject.io>

<sup>51</sup> Google Cloud, "Secret Manager", Retrieved: 12/12/2022, <https://cloud.google.com/secret-manager>



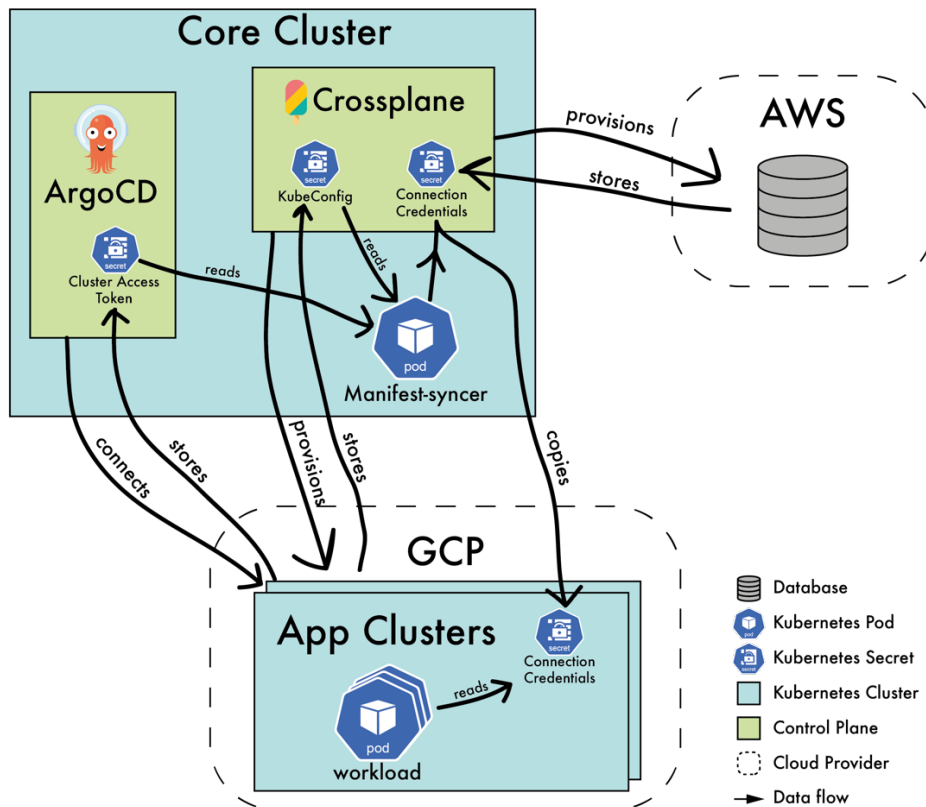


Figure 9: A visualization of how credentials are generated and copied from the Core Cluster to the App Clusters.

In Figure 9, 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) from the Core Cluster to the App Clusters.

One could argue that it is bad practice for a small infrastructure team to build their own services like **manifest-syncer** because they need to maintain them themselves - but since this 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).

## 8. Demonstration Application running in Google Cloud

Continuing from section 6. *Demonstration Application*, I 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 run in GCP while the managed database runs in AWS to show that this kind of setup works across different cloud providers. In GCP, two environments are 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.

The cloud resources needed for this setup are provisioned through Crossplane and can be seen in Figure 10. Crossplane does not have a UI but can be managed through **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** which shows if the resource's actual state in the cloud provider is in sync with the declared state in Kubernetes.

```
> kubectl get managed --context core-cluster
```

NAME		READY	SYNCED						
network.compute.gcp.crossplane.io/vpc-prod		True	True						
network.compute.gcp.crossplane.io/vpc-stage		True	True						

NAME		READY	SYNCED	STATE	ENDPOINT	LOCATION	AGE
subnetwork.compute.gcp.crossplane.io/vpc-subnetwork-prod		True	True	RECONCILING	34.163.175.15	europa-west9-a	29m
subnetwork.compute.gcp.crossplane.io/vpc-subnetwork-stage		True	True	RECONCILING	34.163.38.254	europa-west9-a	29m

NAME		READY	SYNCED	STATE	ENDPOINT	LOCATION	AGE
cluster.container.gcp.crossplane.io/gcp-k8s-prod		True	True	RECONCILING	34.163.175.15	europa-west9-a	29m
cluster.container.gcp.crossplane.io/gcp-k8s-stage		True	True	RECONCILING	34.163.38.254	europa-west9-a	29m

NAME		READY	SYNCED	STATE	CLUSTER-REF	AGE
nodepool.container.gcp.crossplane.io/gcp-k8s-np-prod		True	False	RUNNING	gcp-k8s-prod	29m
nodepool.container.gcp.crossplane.io/gcp-k8s-np-stage		True	False	RUNNING	gcp-k8s-stage	29m

NAME		READY	SYNCED	STATE	ENGINE	VERSION	AGE
rdinstance.database.aws.crossplane.io/aws-database-postgres-instance		True	True	available	postgres	12.10	29m

Figure 10

Figure 10 shows two VPCs (`network.compute.gcp`), two subnets (`subnetwork.compute.gcp`), two Kubernetes clusters (`cluster.container.gcp`), and two node pools (`nodepool.container.gcp`) running on GCP, and a single database instance (`rdinstance.database.aws`) running on AWS.

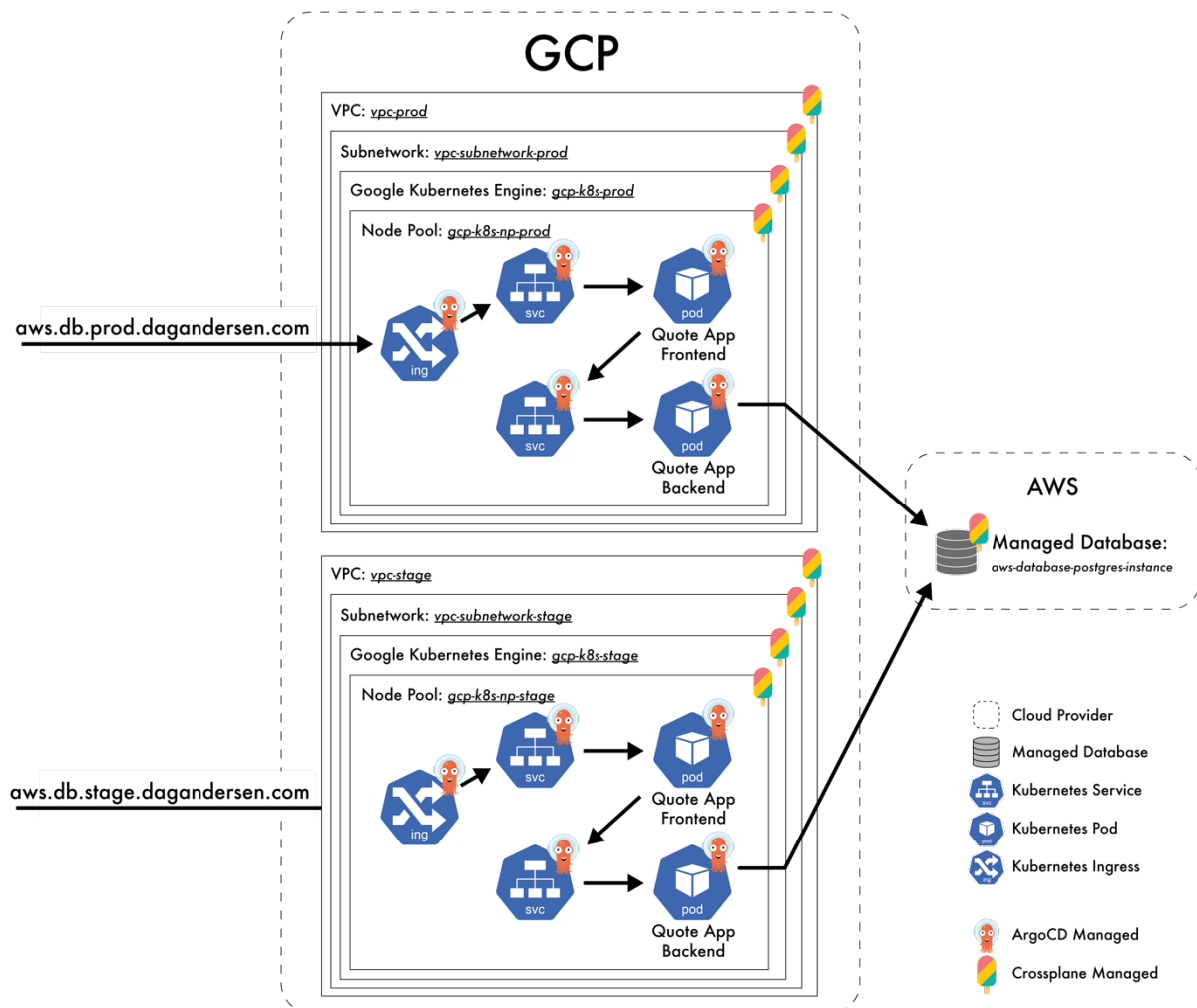


Figure 11: An illustration of how the demonstration application (Quote App) runs on GCP and accesses a database in AWS

Figure 11 illustrates how the demonstration application (Quote App) runs on GCP and accesses a database in AWS. All resources with the Crossplane logo next to them are provisioned by Crossplane, while all Kubernetes objects with the ArgoCD logo next to them are resources deployed and managed by ArgoCD. The text written in *italics* is the name of the Crossplane-object managed in Kubernetes, and they match the objects printed in Figure 10.

The production and staging environments run completely separately on GCP. This design makes it possible to scale the number of workload-environments (App Clusters) linearly from the Core Cluster.

## 8.1. Use in practice

The following section describes how to use this implementation of the universal control plane (the Core Cluster) and how software teams would deploy and update applications in the App Clusters. The section goes through some use case examples of how the Core Cluster works in practice and how a company could choose to build a workflow around it.

The use cases are:

- Spinning up the clusters from scratch
- Developers creating a new service with an associated database
- Deploying a new service version to the multiple software environments

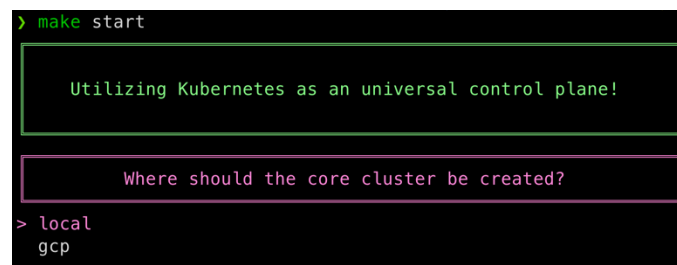
These use case examples assume that the folder structure is the same as described in section: 7.3.2. *Repository structure*.

### 8.1.1. Spinning up the clusters from scratch

Starting the Core Cluster, provisioning the App Clusters, and deploying every service and configuration described in this paper can be done in 4 steps.

1. Pull down the `k8s-ucp-bootstrap`-repository<sup>52</sup>.
2. Run `make install-tools` in the repository-root to install the dependencies.
3. Add your Cloud Provider and Git repository 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 Figures 12 and 13.

A terminal window with a black background. The prompt is '> make start'. Below it, a green-bordered box contains the text 'Utilizing Kubernetes as an universal control plane!'. Below that, a purple-bordered box contains the text 'Where should the core cluster be created?'. At the bottom, the user has entered '> local' and 'gcp' on separate lines.

```
> make start

Utilizing Kubernetes as an universal control plane!

Where should the core cluster be created?

> local
gcp
```

Figure 12

First, the user specifies if the Core Cluster should run locally with Kind or on GCP. Choosing `local` is preferred when developing since it takes around 10 minutes to spin up a cluster on GCP<sup>28</sup>, and it only takes 20 seconds to spin up a local Kind cluster.

<sup>52</sup> Bootstrap Repository, Retrieved: 14/12/2022, <https://github.com/dag-andersen/k8s-ucp-bootstrap>

```
> make start

Utilizing Kubernetes as an universal control plane!

Where should the core cluster be created?

Answer:
local

What should be installed?

○ skip
● prod-cluster
● stage-cluster
○ quote-app-with-gcp-database
> ● quote-app-with-aws-database
```

Figure 13

Secondly, the user specifies which App Clusters should be created (production or staging) and if the Quote App should be deployed together with an associated database.

Regardless of which of the five options is chosen (seen in Figure 13 and in the bulleted list below), the script spins up the Core Cluster and installs ArgoCD. ArgoCD then deploys Nginx, Crossplane, Prometheus, Grafana, ExternalDNS, and the `manifest-syncer`. When these components are running, the Core Cluster can be used as a control plane for spinning up cloud resources and deploying applications to external Kubernetes clusters.

The five options are:

- Choosing `skip` only starts the Core Cluster.
- Choosing `prod-cluster` starts the Core Cluster, provisions the production cluster on GCP, and then deploys Nginx and ExternalDNS to the production cluster.
- Choosing `stage-cluster` starts the Core Cluster, provisions the staging cluster on GCP, and then deploys Nginx and ExternalDNS to the staging cluster.
- Choosing `quote-app-with-gcp-database` starts the Core Cluster, provisions a database on GCP, and then deploys the Quote App to each of the App Clusters (if they exist).
- Choosing `quote-app-with-aws-database` starts the Core Cluster, provisions a database on AWS, and then deploys the Quote App to each of the App Clusters (if they exist).

The time it takes from the `make start`-command to run and till the Quote App is running and accessible through a web browser depends on if the Core Cluster runs locally or on GCP. Provisioning a GKE cluster on GCP takes around 10 minutes<sup>28</sup>. 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.

### 8.1.2. Developers creating a new service with an associated database

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

1. The developer creates a pull request to the `k8s-ucp-core-gitops`-repository with the manifests describing the database instance and a `syncer` object for distributing the connection details to the database (as explained in section: 7.5. *Distributing Secrets*)
2. The developer creates a pull request to the `k8s-ucp-app-gitops`-repository with the manifests describing the application. This could, for instance, be manifests describing *Ingresses*, *Services*, and *Deployments* as with the Quote App.

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 application-related Kubernetes resources (as illustrated in Figure 14).

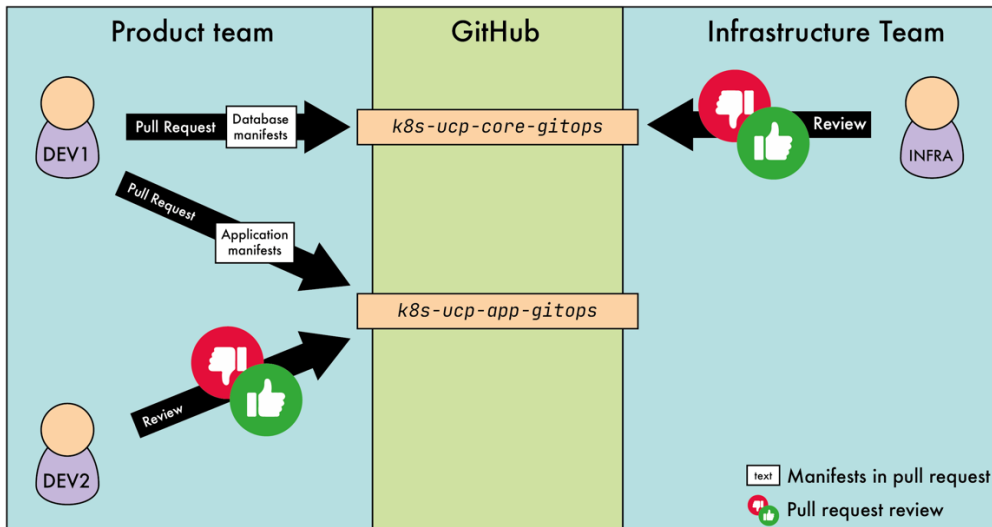


Figure 14: A visualization of the flow when a developer (DEV1) wants to deploy an application to, e.g., the staging cluster together with a database instance

### 8.1.3. Deploying a new service version to the multiple software environments

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

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 to the repository containing the `service-a` code. 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-a`'s version in `k8s-ucp-app-gitops`-repository with the new tags. In this case, the build system updates the Kustomize-file in `k8s-ucp-app-gitops/service-a/overlays/stage/` with the newest version (as illustrated in Figure 15), and ArgoCD will automatically deploy it to the staging cluster.

When the changes have been tested on the staging cluster and the pull request has been merged into master/main, then the same automated process begins. The only difference is that the build-pipeline, this time updates the `k8s-ucp-app-gitops/service-a/overlays/prod/` instead, and ArgoCD will automatically deploy it to the production cluster.

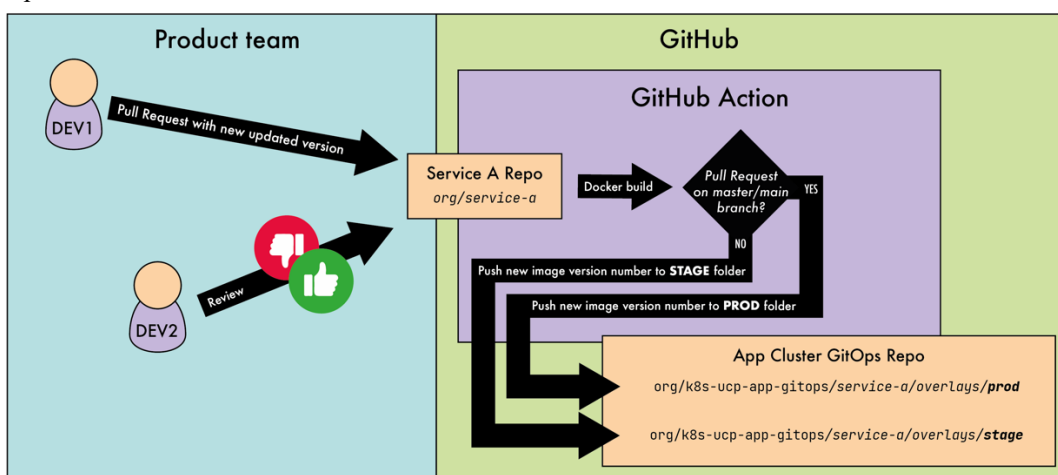


Figure 15: A visualization of the flow when a developer (DEV1) wants to update Service A in the production or staging cluster

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.

## 9. Observations

So far, I have 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 observed when designing, developing, and using Kubernetes as a universal control plane. These topics will give Eficode a better foundation 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 is not described in detail but is instead mentioned as considerations of what to keep in mind when deciding to transition to control plane-managed infrastructure and 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.

Eficode has approved that these topics are all relevant to discuss with clients.

The discussion topics are:

1. State management with Crossplane compared to Terraform
2. No preview with Crossplane
3. Multiple Core Clusters
4. Single interface
5. Platform Engineering
6. Modifying and extending the Core Cluster
7. Better visibility and confidence in the system
8. Declaring the entire infrastructure
9. Bootstrapping Problem
10. Additional cost
11. Build pipelines
12. Maturity level

### 9.1. State management with Crossplane compared to Terraform

The three overall arguments in this paper against Terraforms state are 1) Configuration Drift, 2) only one developer/process can update the state at a time, and 3) the potential for a corrupt or lost state. These will be discussed and compared to how Crossplane manages state.

1) One of the challenges with Terraform is that its state can easily go out of sync with reality – also known as *configuration drift*. This is not possible with Crossplane. Crossplane continuously reconciles the current state to match the desired state. If a Crossplane-managed resource gets deleted or crashes, Crossplane will “self-heal” and recreate the resource<sup>53</sup>. If a Crossplane-managed resource is changed by some other source (like if the node count of a GKE Cluster is manually changed through GCP’s web-interface), Crossplane will simply revert the change. ArgoCD does the same. If a user manually changes one of the ArgoCD-managed manifests with `kubectl`, ArgoCD will revert the change the next time it reconciles with Git to avoid configuration drift. However, the big difference between ArgoCD and Crossplane is that with ArgoCD, a user can disable the automatic syncing temporarily so that manual changes won’t be reverted. This is not possible with Crossplane. This means a user must make changes to the Crossplane-managed resource through Kubernetes and cannot do it through another source (like GCP’s web-interface).

2) Another challenge with Terraform is that multiple developers can’t update the same terraform state simultaneously. This is not a problem with Crossplane (and control planes in general). The declared state stored in Kubernetes is managed through the Kubernetes’ API server. API Server provides the frontend to the cluster’s state through which all other components interact<sup>53</sup>. So, as long as Kubernetes can handle the number of requests produced by the developers, there is no limit to how many developers can update the state simultaneously. The Crossplane-provider will continuously try to reconcile the actual state with the state declared at that given time in Kubernetes. How well each Crossplane-provider handles a frequently updating state depends on the individual implementation.

---

<sup>53</sup> Kubernetes Docs, “kube-apiserver”, Retrieved: 12/12/2022, <https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/>

3) One of the main arguments for moving away from Terraform is the stored Terraform state it creates. The state can be corrupted or lost. If a control plane like Crossplane does not prevent this, I would not consider Crossplane much of an improvement compared to Terraform. When I refer to a “stateless cluster” in this section, I mean that the cluster is deleted and recreated without losing any data or configuration.

When making changes to the Terraform state, Terraform looks at the stored state, the Terraform configuration code, and what is currently running in the cloud. Crossplane only looks at the declared manifests and what is currently running in the cloud. This means there exists no stored state that can get corrupted. 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 resources like VPC-networks on GCP, but with 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 (a kubeconfig) as a *Secret* in the cluster with Crossplane installed. If the GKE cluster already exists, it will not provision anything but simply pull down the connection details (kubeconfig). That means 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 is lost.

This is not the case with database connection details. The database password will only be pulled down on creation, but never again. This is the case with both the GCP and AWS Crossplane providers. This is due to security reasons<sup>54</sup>. 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 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 section 7.5. *Distributing Secrets*, a better way of handling *Secrets* would be installing some sort of secret-vault (like HashiCorp Vault or GCP’s Secret Manager) instead of relying on copying *Secrets* between clusters with my *manifest-syncer*.

This means that the Core Cluster is stateful, and therefore the state can be lost if the cluster is deleted. If the cluster used a secret-vault where the database credentials were stored outside the cluster, then the Core Cluster would be stateless, and I would have eliminated all three issues related to the Terraform state. If the project scope had been bigger, I would have introduced a secret-vault like HashiCorp Vault and, that way, made the Core Cluster completely stateless, and hence it could have been deleted and recreated arbitrarily.

*To conclude:* In contrast to Terraform, Configuration Drift is not possible with Crossplane since it continuously reconciles the current state with the declared state. Furthermore, multiple people can update the Crossplane state at the same time because the declared state is managed by Kubernetes’ API Server. The only thing blocking the entire Core Cluster from being completely stateless is the database credentials/*Secrets* stored in the Core Cluster. Introducing a secret-vault like HashiCorp Vault would make the Core Cluster stateless so that no state could be lost.

## 9.2. 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 changes 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<sup>55</sup>.

<sup>54</sup> GitHub Issue, Retrieved: 28/11/2022, <https://github.com/crossplane-contrib/provider-gcp/issues/397#issuecomment-948758190>

<sup>55</sup> GitHub Issue, Retrieved: 25/11/2022, <https://github.com/crossplane/crossplane/issues/1805>



I do not 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. Also, I do not 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 one of the biggest drawbacks of using Crossplane for infrastructure-management.

### 9.3. 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 it's upgraded 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 into 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 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 (as illustrated in Figure 16). It is fairly straightforward code-wise, so it is doable, but this would effectively double the infrastructure costs.

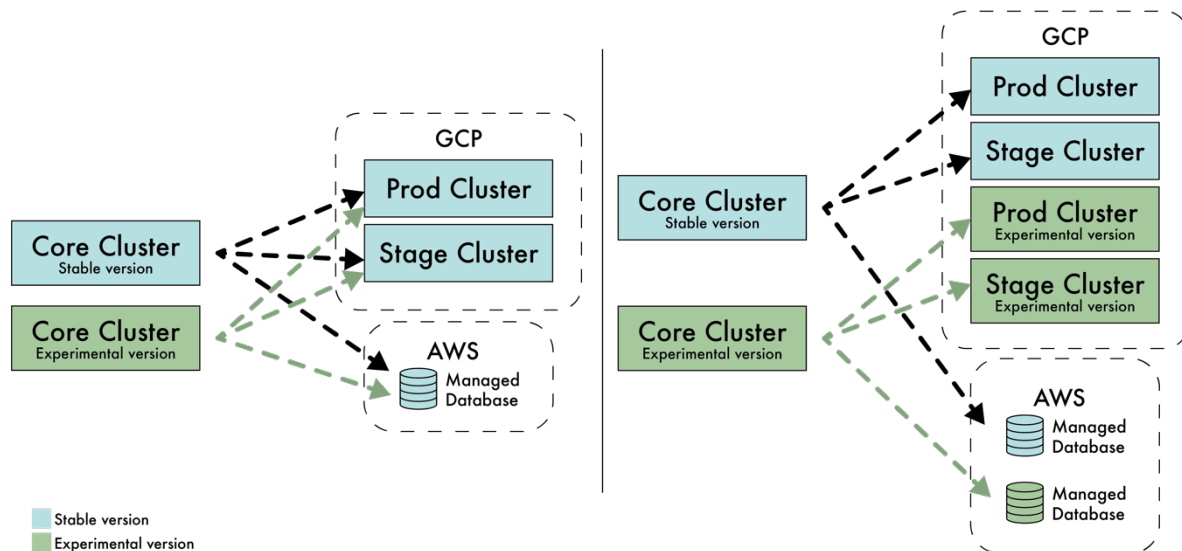


Figure 16: An illustration of how an experimental Core Cluster could either reconcile with the same resources as the stable Core Cluster (left) or with a duplicated version of the entire infrastructure (right).

But the biggest drawback of running a complete copy of the 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 manages. Solving this issue is beyond the scope of this project.

*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 the 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.

## 9.4. 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."*<sup>56</sup>. 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"*<sup>57</sup>.

Suppose a company wants 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, a company 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 a company wants its 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.

## 9.5. Platform Engineering

Platform Engineering has been gaining popularity in the last two years<sup>56</sup>. Control planes like Crossplane can modernize infrastructure-management by embracing Platform Engineering and self-servicing. Crossplane has the concept of *Composite Resources*<sup>57</sup>, 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«"*<sup>58</sup>. 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<sup>59</sup>.

*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.

## 9.6. Modifying and extending the Core Cluster

The Core Cluster can be extended by other control planes. Only ArgoCD and Crossplane are necessary to run the Quote App, but other control planes could easily be added if more features were needed.

Crossplane currently focuses on cloud providers, so if a user, for example, wants to manage external resources that are not cloud-related, the user can simply install a control plane for that as well.

<sup>56</sup> Luca Galante, CNCF, "DevOps vs. SRE vs. Platform Engineering? The gaps might be smaller than you think", July 1, 2022, Retrieved: 28/11/2022, <https://www.cncf.io/blog/2022/07/01/devops-vs-sre-vs-platform-engineering-the-gaps-might-be-smaller-than-you-think/>

<sup>57</sup> Crossplane Docs, "Composite Resources", Retrieved: 23/11/2022, <https://crossplane.io/docs/v1.9/concepts/composition.html#composite-resources>

<sup>58</sup> Kelsey Hightower, Crossplane YouTube Channel, Time: 14:03, Retrieved: 11/11/2022, <https://youtu.be/UfFM5Gr1m-0?t=843>

<sup>59</sup> Kelsey Hightower, Crossplane YouTube Channel, Time: 12:49, Retrieved: 11/11/2022, <https://youtu.be/UfFM5Gr1m-0?t=769>

Crossplane is built to be highly extendable (just like Terraform), making it possible 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<sup>60</sup>), where the user could declare their Datadog dashboard in manifests and apply it to the cluster. That way, users could move their Datadog dashboard configuration into the Core Cluster instead of managing it with Terraform.

Both the major components, ArgoCD and Crossplane, could be replaced by alternatives.

Crossplane could be replaced with Google's *Config Connector* or AWS' *Controllers for Kubernetes* if a multi-cloud architecture is not needed.

ArgoCD could be replaced by FluxCD, which is another popular GitOps-tool for Kubernetes. FluxCD and ArgoCD share most of the same features, but the way users structure their code looks a bit different. Both tools would be good candidates for building a universal control plane for Kubernetes.

A company could even run ArgoCD, FluxCD, Crossplane, Google's *Config Connector*, and AWS' *Controllers for Kubernetes* at the same time. The Core Cluster can run as many control planes as needed.

*To conclude:* Only ArgoCD and Crossplane are necessary to run the Quote App, but other control planes could easily be added if more features were needed. Both ArgoCD and Crossplane can be replaced with alternative tools if needed. The main point of this paper is not whether Crossplane or ArgoCD are great tools or not - but instead whether the paradigm of control planes is good in general.

## 9.7. Better visibility and confidence in the system

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<sup>61</sup>. 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.

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.

*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 are provisioned and what software is managed and deployed from Git.

## 9.8. Declaring the entire infrastructure

The setup described in this paper is built using only 2 file types: manifests (YAML) and makefiles. Manifests are 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"*<sup>30</sup>. This creates a highly streamlined infrastructure because it does not require knowledge about, e.g., Terraform, Ansible, or multiple scripting languages. This is also known as reducing the amount of *Cognitive Overhead*<sup>30</sup>. I consider this a huge benefit of this setup.

---

<sup>60</sup> Terraform's Registry, "Datadog Provider", Retrieved: 14/12/2022, <https://registry.terraform.io/providers/DataDog/datadog/latest/docs>

<sup>61</sup> Wiebe de Roos, Amazic, "Don't waste your money on unused cloud resources", November 4, 2019, Retrieved: 14/12/2022, <https://amazic.com/dont-waste-your-money-on-unused-cloud-resources>

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

## 9.9. 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<sup>62</sup>.

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 the 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 with eventual consistency by ArgoCD.

*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 minimal since it only does 2 steps. 1: Start the Core Cluster (locally or on a cloud provider), 2: Install and set up ArgoCD.

## 9.10. Additional Costs

When using Kubernetes as a control plane for managing all my infrastructure, I am running an additional Kubernetes cluster (the Core Cluster), which is not free.

When running the demonstrated application (Quote App), more resources were required to run the Core Cluster than the staging and production clusters combined. The ratio between the resources required by Core Cluster and App Clusters depends on how much workload runs in the App Clusters. For my small demonstration application, the resources required by the Core Cluster surpassed the App 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 ( $\$24.46 \cdot 3$ ) for running just the Core Cluster<sup>63</sup>. 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.

---

<sup>62</sup> Wikipedia, "Command (computing)", November 19, 2022, Retrieved: 5/12/2022, [https://en.wikipedia.org/wiki/Command\\_\(computing\)](https://en.wikipedia.org/wiki/Command_(computing))

<sup>63</sup> Economize, "e2-medium", Retrieved: 4/11/2022, <https://www.economize.cloud/resources/gcp/pricing/e2/e2-medium>

## 9.11. Build pipelines

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

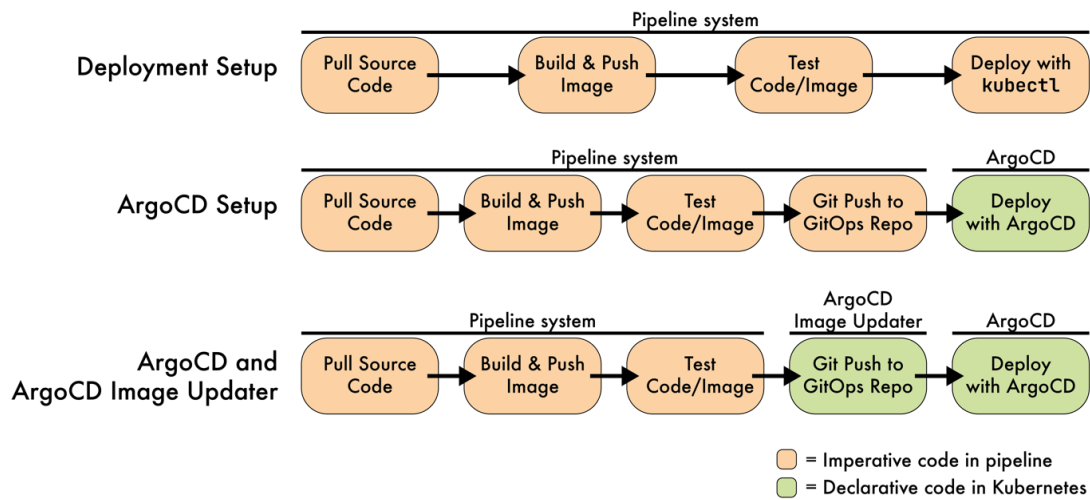


Figure 17: This figure shows 3 different ways of building a deployment system

CI/CD pipelines often consist of 4 steps: *Pull source code*, *build code*, *test code*, and *deploy*<sup>64</sup>. 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 repository instead of applying manifest changes to the cluster directly (as illustrated in Figure 17).

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 repository, 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<sup>65</sup>). 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<sup>66</sup>. The Flux automated image updater may work, but since I went with the ArgoCD ecosystem, it was not reasonable to try out the flux version in this limited time frame.

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

## 9.12. 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*”<sup>11</sup>. When they use ArgoCD, they put all their faith in that ArgoCD correctly deploys their services and does not randomly delete arbitrary *Pods*. When they use Crossplane, they rely on the providers to correctly manage the life cycle of the requested resources. As a user of these control planes, it is out of your hands, and you rely solely on the tools doing their job correctly.

<sup>64</sup> Weaveworks’ Blog, “GitOps - The Path to A Fully-Automated CI/CD Pipelines”, June 09, 2022, Retrieved: 29/11/2022, <https://www.weave.works/blog/gitops-fully-automated-ci-cd-pipelines>

<sup>65</sup> GitHub Issue, Retrieved: 25/11/2022, <https://github.com/argoproj-labs/argocd-image-updater/issues/496>

<sup>66</sup> Flux Docs, “Image Update Automations”, Retrieved: 03/12/2022, <https://fluxcd.io/flux/components/image/imageupdateautomations/>

This is the same limitation that Terraform has. Terraform is only as good as the providers that integrate with Terraform.

#### *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<sup>67</sup>). 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 I 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 became aware of 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<sup>68</sup>). So, if a helm-charts generates manifests that are 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 frustrating since if ArgoCD cannot deploy *every single resource* declared in my infrastructure, I 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<sup>69</sup>). 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 it happened.

*To conclude:* Both ArgoCD and Crossplane are good tools with strong support from the community and industry, but they are not flawless. Both tools are still in active development, so users should expect small issues. 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.

## 10. Conclusion

Control planes are a new paradigm in DevOps and infrastructure management. This paper explains the concept of control planes and how to utilize Kubernetes as a universal control plane for handling both internal resources (e.g., containers) and external resources (e.g., cloud resources) and discusses the challenges and implications.

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 that control planes like Crossplane can be used to replace infrastructure tools like Terraform by controlling the full life cycle of cloud resources from within Kubernetes.

The main design idea of the implementation presented in this paper is to have a so-called “Core Cluster” that works as a universal control plane for managing the full lifecycle of all infrastructure configurations and software deployments. The Core Cluster's two main components are ArgoCD and Crossplane, which are control planes built to run in Kubernetes. ArgoCD handles all internal state (e.g., deploying containers and configuration), while Crossplane handles all external states (e.g., provisioning cloud resources).

The implementation presented in this paper is highly extendable. Only ArgoCD and Crossplane are necessary to run the Core Cluster, but other control planes could easily be added if more features were needed. In addition, both ArgoCD and Crossplane can be replaced with alternative tools if needed. The question is not whether

---

<sup>67</sup> GitHub Issue, Retrieved: 05/11/2022, <https://github.com/crossplane-contrib/provider-digitalocean/issues/55>

<sup>68</sup> GitHub Issue, Retrieved: 23/11/2022, <https://github.com/argoproj/argo-cd/issues/8128>

<sup>69</sup> GitHub Issue, Retrieved: 15/11/2022, <https://github.com/argoproj/argo-cd/issues/4651>



Crossplane or ArgoCD are great tools or not - but instead whether the paradigm of control planes is good in general.

This paper is done in collaboration with Eficode and aims to provide the company with a better understanding of the challenges and implications of transitioning to control plane-based infrastructure-management. To demonstrate the Core Cluster's multi-cloud and multi-environment capabilities, an application developed by Eficode is deployed on infrastructure managed by the universal control plane (the Core Cluster). The application runs in a Kubernetes cluster in production and a staging environment on Google Cloud Platform (GCP) and accesses a managed database on Amazon Web Services (AWS). GCP and AWS could be interchanged or replaced by Microsoft Azure.

The paper discusses some of the challenges and limitations observed when designing, developing, and using the Core Cluster.

Compared to Terraform, Crossplane manages to eliminate issues related to Configuration Drift and allows multiple developers to update the infrastructure state simultaneously. Furthermore, Crossplane does not store a state like Terraform, so it does not have a state that can get corrupted. In my implementation, state can still be lost if the Core Cluster is deleted. The database credentials stored in the Core Cluster are the only data preventing the entire Core Cluster from being completely stateless. Introducing a secret-vault like HashiCorp Vault or GCP's Secret Manager would make the Core Cluster stateless so that no state could be lost.

One of the main benefits of this implementation is that it has a highly streamlined configuration declared in Kubernetes manifests, which is checked into Git as Infrastructure as Code. Another benefit is that the bootstrapping is minimal since it only requires spinning up the Core Cluster and installing ArgoCD, and the rest is handled automatically with eventual consistency. A third benefit is that the universal control plane serves as a good basis for platform engineering, which enables an infrastructure team to build an engineering platform where the developers can self-service cloud resources provided and controlled by the infrastructure team.

One of the main drawbacks of using Crossplane in the Core Cluster for provisioning infrastructure is that Crossplane cannot show a preview of the resources it is going to create/modify/delete. This makes Crossplane risky to use for critical infrastructure, and I would consider this the biggest drawback of using Crossplane for infrastructure-management. Another drawback of this implementation is that 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 the organization.

Overall, the conclusion is that control plane-based infrastructure-management comes with some great benefits but also introduces some significant downsides depending on the specific implementation and tools chosen. I recommend using control planes for managing internal resources. If a company manages to mitigate the drawbacks and limitations mentioned in this paper, I also recommend it for managing external resources.