1. Utilizing Kubernetes as an universal control plane

2. Abstract

3. Table of Contents

4. Dictionary and abbreviations

- **Infrastructure as Code (IaC)**: IaC is the concept of managing and provisioning of infrastructure through code instead of through manual processes [s].
- **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 [s]
- **Reconciler pattern**: Reconciler pattern is the concept of checking the difference between the desired state and the current state, and if there is a difference, an action is taken to make the current state into the desired state [s].
- **Control plane**: A control plane reconciles the current state to match a desired state. Control planes follow the Reconciler pattern. An example of this is Kubernetes' internal control plane that schedules pods/containers based on a declared state.
- **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 [s]
- **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 [s].
- **Google Cloud Platform (GCP)**: Cloud computing platform provided by Google.
- **Amazon Web Services (AWS)**: Cloud computing platform provided by Amazon.

5. Introduction

With the creation of Kubernetes, we are seeing a rise in popularity of Control Planes. The concept is built on the idea that you declare a state and services, then watches the state and make sure that the system's actual state reflects the state declared. If the desired state changes, the control plane ensures that the new state is automatically reflected in the actual state. Control planes are self-healing, and they automatically correct drift.

Kubernetes is a perfect ecosystem fundament for control planes because most events that happen inside is controlled/managed by control planes. At its core, Kubernetes stores a state declared in YAML, and different services watch this state and make sure the actual state reflects the declared state. Kubernetes can easily be extended by creating new services that watch the state and act accordingly. During the last years, new control planes have emerged that also manage external resources. It works more or less the same way, but now the services instead ensure that some external state corresponds to the declared state.

Now the question is if we can combine these tools in order to create a universal control plane for handling all kinds of states, all managed from within Kubernetes? How would such a setup look? How would you interact with it? And what concerns and implications may such a setup bring?

6. Motivation

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

Eficode has stated their concern about how Terraform does not scale well for big organizations in their blogpost: Outgrowing Terraform and adopting control planes. 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 is a new paradigm in the realm of DevOps and infrastructure management. Many of the technologies/tools leveraging the concept of control planes are still considered new and do not have many years of proven use. Even though many of the tools/systems look promising, it can be difficult to justify the investment in transitioning a DevOps infrastructure to this new paradigm.

In order 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. This paper will implement a universal control plane for handling both software deployment and cloud resources and discuss the implications.

The design and implementation presented in this paper is created and written by myself, and Eficode has not been part of it.

7. Declarative vs Imperative

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

Declarative and Imperative are two different DevOps Paradigms. Declarative Programming specifies *what* final state we want, while imperative programming specifies *how* we want to get to a given state.

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

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

7.1. Terraform

A very popular tool that is based on the idea of IaC and declarative configuration is Terraform. Terraforms' popularity started in 2016-2017 and has been growing ever since. Now Terraform has integration with more than 1700 providers [s]. So the natural question is now, why bother with control planes when tools like Terraform exist? This section describes some of the issues related to using Terraform and why it might be better to use control planes for handling external resources.

Terraform is an Infrastructure as Code tool that lets you define both cloud and on-prem resources in configuration files that you can version, reuse, and share. You can then use a consistent workflow to provision and manage all of your infrastructure throughout its lifecycle. Terraform can manage all kinds of resources, from low-level components to huge cloud provider-managed services [s]. Terraform has the concept of terraform-providers, where service providers 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" [s].

7.1.1. Barrier of entry

The only practical way of using Terraform in teams is to store the state in some shared place. This is commonly done on a cloud provider in some kind of *bucket/Object storage*. Setting this up can be a big hurdle to overcome if it is your first time or you are not an experienced developer. So before you can solve *whatever your code is supposed to solve*, you first need to solve the problem of *how and where to store the Terraform state*, before you can even start developing anything in your team.

Removing this step by switching to control planes is a selling point on its own. Lowering a barrier to start up projects is always desirable.

7.1.2. Challenges with managing state

Distributed teams must store the Terraform state in some remote place. But just because it is shared in a remote place does not mean two people can work on it at the same time.

When storing the state in a remote place, you need to specify a so-called `backend`, but not all `backend`s support locking. This means that in some cases, race conditions can still happen (if two people run `terraform apply` at the same time). An example of this is that there is no lock on the state if stored on AWS S3. A solution to this is to store lock file somewhere else (e.g. AWS DynamoDB) [s] and [s]. This again just adds to the complexity and creates an even bigger barrier if you want to make sure your Terraform is safe to use.

Furthermore, if a `terraform apply` goes wrong because the process is interrupted for whatever reason, the state can end up not being unlocked, and you have to `force-unlock` the state [s]. This is, of course, a great feature to solve the issue - but the main problem here is that this can happen to begin with.
Things can get even worse if a `force-unlock` is executed while another process is in the middle of applying [s]. Managing a mutual state in a distributed state is difficult at its core.

But even though there may be a lock that makes sure that there is no race condition while applying, only one person/process can work on the state simultaneously. Updating a Terraform state can take minutes - e.g., it will take around 10 minutes to spin up a GKE cluster on GCP [s]. "During this time no other entity - no other engineer - can apply changes to the configuration." [s]. This is an even bigger problem when you have a monolithic setup with a lot of dependencies. So if one developer is updating the GKE cluster, then another developer may be blocked from updating a database or a networking rule.

Another challenge with Terraform state is that Terraforms state can easily go out of date with reality. This is called *configuration drift* [source and source]. If the `terraform apply` -command is not run regularly, the actual state can drift away from the declared/desired state. This means the state no longer reflects the real world. This can create issues when later mutating the stored state, which can make the stored state unusable because it is so far from the actual state.

This can both be a gift and a curse. If it is an emergency, then it is possible to modify the terraform-created-resource manually through some other service (e.g., a person could change a network rule manually through Google Cloud Platform) without it being reverted by some automated tool.

So the Terraform state can either be updated by a manual task (e.g., a developer manually creating a database), a triggered automated task (e.g., a deployment pipeline that applies the newly changed Terraform files - e.g. Atlantis or Terraform Cloud), or some GitOps tool that continuously synchronizes the actual state with the declared state stored in a repo (or elsewhere).

7.1.3. State automation

You could install an automated tool or script that simply does this `terraform apply` -step on a regular basis to ensure that there is no *configuration drift*. This is a reasonable approach, but then you are essentially creating a system that works just like a control plane. So instead of using a tool with all its issues and then patching some of the issues by wrapping it in some automation tool/script... why not just use a control plane that was built to solve exactly that?

Kubernetes stores desired state, and the internal components try to keep the actual state as close as possible to the desired state. The state is stored as Kubernetes objects definitions declared in YAML. Why not store the same information usually stored in Terraform state inside Kubernetes instead? A control plane running on Kubernetes could automatically sync (similar to continuously running `terraform apply`) the actual infrastructure with the declared state in YAML.

This is where tools like *Crossplane*, Google's *Config Connector*, and AWS' *Controllers for Kubernetes* come into play. This paper will focus on Crossplane because it is built as a foundation for control planes in general and not only focuses on a single cloud provider.

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

7.1.4. Crossplane as an alternative

The most popular cloud native control plane tool for handling external state is *Crossplane*. Crossplane is a CNCF project that tries to streamline cloud providers' APIs.

Crossplane tries to follow the success of Terraform and applies some of the same concepts to Kubernetes. Terraform has the concept of *providers*. A Terraform provider is a Terraform plugin that allows users to manage an external API. "Terraform uses providers to provision resources, which describe one or more infrastructure objects like virtual networks and compute instances. Each provider on the Terraform Registry has documentation detailing available resources and their configuration options." [source]. Crossplane mimics this concept, so external organizations can create a plugin that integrates with Crossplane providing the user the ability to provision external resources.

Crossplane can be used to provision resources for all cloud providers if the cloud provider has created a Crossplane provider for it. As a company/organization, it can be beneficial to have the option to be multi-cloud because different cloud providers have different offerings.

Terraform uses HashiCorp configuration language (HCL), while Crossplane uses `YAML` and follows the same structure conventions as Kubernetes manifests.

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

Crossplane will be described in greater detail in the *Managing External State*-section.

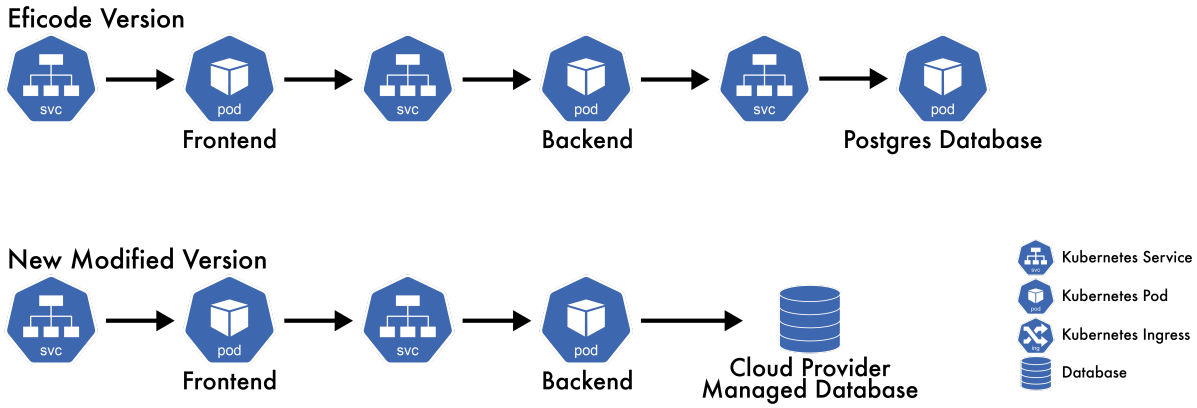
8. Demonstration Application

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

To do this, I have used Eficode's public `quotes-flask` application that they use for educational purposes. It is a simple application consisting of a frontend, backend, and a database.

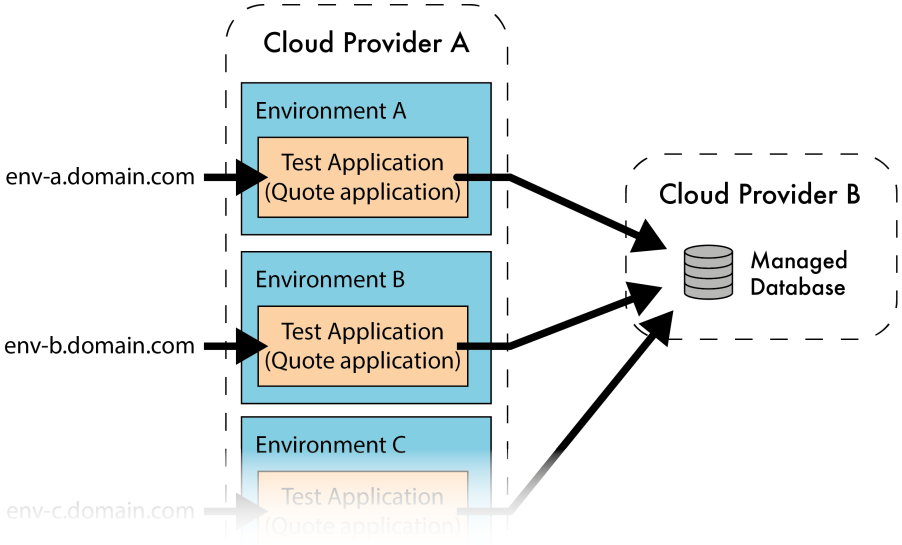
The frontend is a website where you can post and retrieve "quotes". The quotes are posted and sent to the backend-service, which then stores the data in a Postgres database.

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



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

The business may want multiple environments like *production*, *staging*, and *development*, and they may leverage cloud services across multiple cloud providers. Therefore the aim of the system is to look something like what can be seen on Figure X.



> Figure X: A visualization of the demonstration application running in multiple environments.

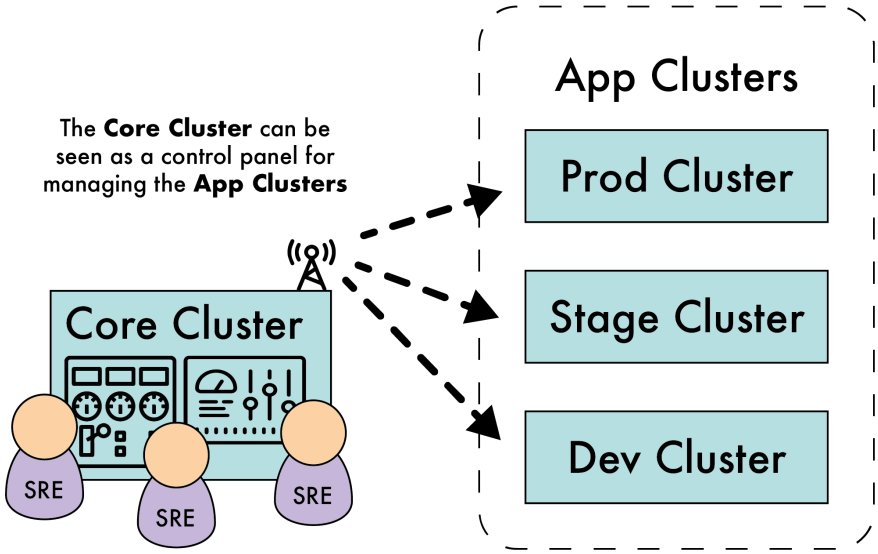
This architecture will demonstrate the universal control plane's multi-cloud and multi-environment capabilities. The evaluation of this project will partially be based on how well the implemented universal control plane manages to host/deploy/run this demonstration application and what implications/challenges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

9. 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 small 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. This cluster will provision other clusters and deploy software to them.

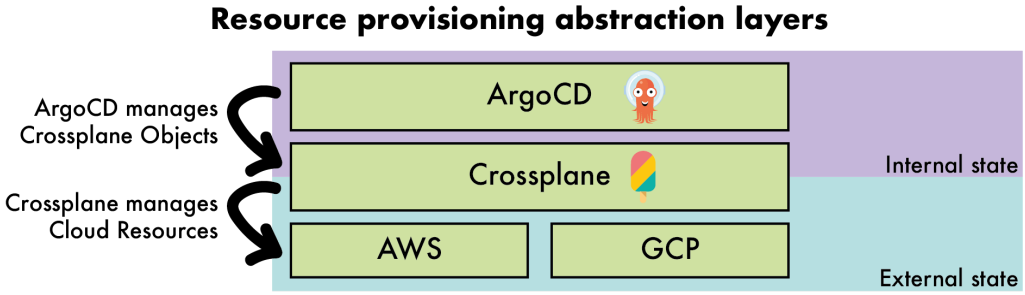
To better understand the design idea, two names are introduced: `core cluster` and `app cluster`. The `core cluster` is the control plane for managing your infrastructure and software environment. The `app clusters` is a shared term for all the clusters where business logic is supposed to run. For instance, a company may have two `app cluster`s in the form of a production cluster and a staging cluster. The `core cluster` hosts all the shared services between different `app cluster` environments.



> Figure X: An visualization/drawing of how the SRE/infrastructure team manages `app clusters` through the `core cluster`.

Only the infrastructure teams are supposed to interact with `core cluster` directly - while the application developers are supposed to only care about getting their workload running on the `app cluster`s.

The `core cluster` use Crossplane for provisioning cloud resources and use ArgoCD to deploy and manage all services that are running in the core cluster and the app clusters. Crossplane and ArgoCD are both open-source tools funded by the CNCF. They are created as Control planes for Kubernetes, and together they can be used to create a control plane for managing multi-cloud multi-environment infrastructure.



> Figure X: An visualization of how ArgoCD and Crossplane work together to provision cloud resources.

ArgoCD handles all internal state, while Crossplane handles all external states - combined, they can be used as a universal control plane.

This setup can easily be extended by other tools. Crossplane is currently focused on cloud providers (but nothing prevents it from expanding to new areas in the future), so if you want to manage external resources that are not cloud-related, you could simply install a control plane for that as well. The main point is that only ArgoCD and Crossplane are necessary for the use cases presented in this paper, but other control planes could easily be added if more features were needed.

9.1. List of tools/technologies used in this implementation

ArgoCD
ArgoCD is a GitOps tool that is built as an control planes that runs inside Kubernetes. ArgoCD was accepted to CNCF on April 7, 2020, and is at the Incubating project maturity level [s]. ArgoCD is used for this implementation because it has more advanced UI features compared to similar-tools. The two biggest GitOps-tools are FluxCD and ArgoCD. This implementation could also be built using FluxCD. FluxCD and ArgoCD cover most of the same features, but the way you structure code looks a bit different. Both tools would be good candidates for building a universal control plane for Kubernetes.

Crossplane
Crossplane is used in this implementation because it is the only option if you want a control plane that works with multiple cloud providers (and is not just a Terraform synchronizer service that is moved into Kubernetes). If a multi-cloud architecture is not needed, one could instead opt for Google's *Config Connector*, or AWS' *Controllers for Kubernetes*. Crossplane was accepted to CNCF on June 23, 2020, and is at the Incubating project maturity level [s].

Google Cloud Platform
GCP was chosen as the main provider for cloud-hosted Kubernetes clusters because it is less complex to use than AWS but advanced and mature enough to work well with Crossplane. Another choice would be DigitalOcean, but I experienced some issues with using Crossplane with DigitalOcean (also described in the *Maturity level*-section). I didn't experiment with Microsoft Azure since I had no prior experience working with them as a cloud provider. Both AWS and Azure would be a potential alternatives to GCP in this implementation.

Kind
Kind is used for running the Kubernetes cluster locally. When developing and experimenting with this setup, it can be beneficial to run the setup locally because it can take a long time to provision clusters on cloud providers (e.g., it often takes 10 minutes on GCP). There are many different tools for running Kubernetes locally, and many of them would probably work for this implementation, but the choice ended with Kind because it is easy to set up and simple to use. Other alternatives could be *MicroK8s* or *Docker Desktop*.

Gum
Gum is a simple command line tool for making interactive scripts. When you want to start any type of custom-made system from a terminal, it usually involves creating multiple scripts for doing some of the initial bootstrapping. Gum was chosen as a simple tool for interactively picking a configuration on runtime instead of having to remember command parameters or creating static config-script.

Helm and Kustomize
Helm and Kustomize are used to template and install Kubernetes resources. Helm is a package manager for Kubernetes. 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.

9.2. Managing internal state

Everything deployed to the clusters are declared in YAML, checked into git, and read by ArgoCD.

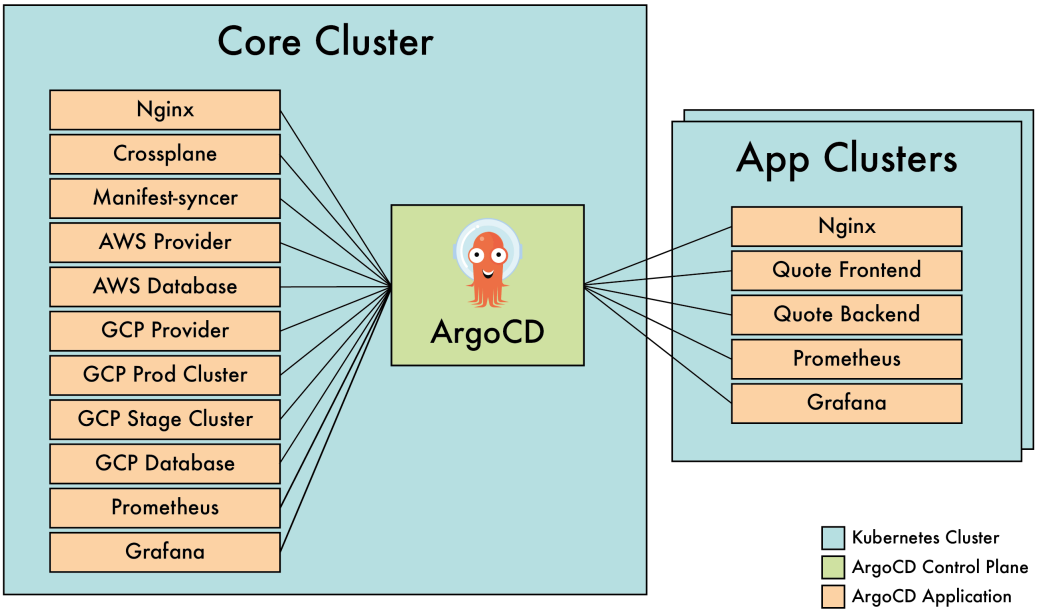
ArgoCD can either be installed on each cluster individually (only controlling the local state) or on a single shared cluster which then handles the deployment to multiple clusters.

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

The implementation presented in this paper has a single instance of ArgoCD running on the `core cluster` and it deploys and manages all the Kubernetes Objects running in both the `core cluster` and the `app clusters`.

ArgoCD groups YAML-files/folders into an abstraction called *Applications*. An Application is a Kubernetes object that contains a path to a resource that needs to be deploy, a destination cluster, and some configuration parameters. You can structure (and nest) these groupings however you want, but you usually want smaller groupings, so you manage/sync them individually with more fine-grained control. e.g., deploying Crossplane-provisioned database separately from the Crossplane provisioned Kubernetes Cluster. This way, you don't have to use the same sync-policy for both groupings of resources. e.g., you may have pruning (auto-delete) disabled on the database to ensure databases are not deleted by mistake.

Based on this philosophy of separation, I have chosen to structure my applications like this:



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

9.2.1. Other applications/packages

Besides Crossplane, ArgoCD also installed other packages/services:

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

Prometheus and Grafana
Prometheus and Grafana is a famous open-source monitoring stack. This stack is not strictly needed to run the demonstration application, but it is used to resemble a realistic setup you would see in a company. It is installed on both the core clusters and the app clusters, so it is possible to observe resource usage of all clusters.

9.2.2. Eventual consistency

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

In any modern software environment, there exist dependencies. The amount of dependencies depends on how well your system is decoupled. If a typical company had to spin up its entire infrastructure from scratch, it would probably include a lot of sequential steps in a specific order because some of its services need other services to run. Doing sequential deployments through a script often takes a long time because it runs sequentially and not in parallel. With Eventual consistency, you can much easier run multiple steps at the same time. Kubernetes is full of eventual consistency. There is no order of when which resources/events are created/handled - instead, everything will just eventually be set up/running. Applying Kubernetes resources with ArgoCD works the same way. If ArgoCD fails to deploy one of these `Applications` (because some dependency may be missing), it will just automatically try again a minute later. This means I can apply our entire infrastructure at once with ArgoCD, and ArgoCD will make sure everything will be deployed with eventual consistency even though there are broken dependencies temporarily in the process. This also applies every time you update something in the infrastructure. Configuration and workload will be applied and scheduled whenever ArgoCD eventually syncs its state with whatever is stored in the chosen git repository.

9.2.3. Repository structure

The structure for this implementation is split up into 3 repositories: One ArgoCD Synced repository for the `core-cluster`, a second ArgoCD synced repository for syncing with `app-cluster`, and finally, a general repository with code and scripts for bootstrapping the system.

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

> `k8s-ucp` stands for *Kubernetes Universal Control Plane*.

The GitOps synced repositories do not contain anything other than resources synced with ArgoCD. Only automated tools and scripts commit changes to the repositories.

You could choose to store the `core-cluster` and `app-cluster` in the same repository and store every ArgoCD-application for the whole organization in a single repository. From a technical standpoint, there is no problem with this solution. It may even be preferred because it will be easier to update the structure for a system requiring changes in multiple clusters simultaneously.

```
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, you may not want everyone in the organization to have read-access to all infrastructure. The write-access would not be a problem since you can, e.g., use `CODEOWNERS` on github.

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

Below we can see the folder structure of the `Core Cluster` and `App Cluster`. Most resources that are applied by ArgoCD are structured/built with Kustomize using the base-overlay-pattern. This is why we have all the `base` and `overlays` folders.

App Cluster Git Repo	Core Cluster Git Repo
k8s-ucp-app-gitops ├── argo-bootstrap │ ├── prod │ └── stage ├── example-database-eficode-aws │ ├── base │ │ ├── backend │ │ └── frontend │ └── overlays │ ├── prod │ └── stage ├── example-database-eficode-gcp │ ├── base │ │ ├── backend │ │ └── frontend │ └── overlays │ ├── prod │ └── stage └── kube-applications ├── base └── overlays ├── prod └── stage	k8s-ucp-core-gitops ├── projects # Declaration of how Argo Applications are grouped ├── argo-bootstrap │ ├── gcp │ └── kind ├── argo-config # Ingress configuration for accessing the argo server on GCP or kind. │ ├── base │ └── overlays │ ├── gcp │ └── kind ├── aws-provider ├── aws-database ├── gcp-provider ├── gcp-database ├── gcp-clusters │ ├── base │ └── overlays │ ├── prod │ ├── prod-pre │ ├── stage │ └── stage-pre ├── manifest-syncer └── kube-applications ├── base ├── envs │ ├── core

```
│   └── experimental
└── host
    ├── gcp
    └── kind
```

9.3. Managing External State

In this implementation, I use Crossplane for managing the external resources. Crossplane is a control plane that runs inside Kubernetes that makes sure that the external resources described in YAML are in sync with the state declared in Kubernetes.

Here you can think about Terraform and the provider-integrations that exist. The developer creates a provider-configuration and is then able to create resources on that provider's infrastructure. It is now up to the provider to manage and ensure that the state running on their infrastructure matches the desired state declared in your Kubernetes cluster.

Currently, Crossplane supports AWS, GCP, and Azure as cloud providers. A DigitalOcean provider is also in active development [source].

As an example, in order to provision a resource on AWS, you need to create a `Provider`, `ProviderConfig`, and *the resource you want*. So if we want to provision a Postgres database on AWS, we apply the following configuration to Kubernetes (with Crossplane installed):

Provider Configs

```yaml
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
```

Provider Resources

```yaml
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
```

These YAML files are read by a *controller* (which is provided by the Crossplane-provider as a *helm-chart*). As a developer, you do not care how it is implemented behind the scene; you just know that the controller continuously tries to make sure the desired state is upheld.

The controller will read the above `RDSInstance` and check that such an instance exists on the AWS account referenced in the `ProviderConfig` 's `credentials` -section. This is how Crossplane is able to create DNS Records, VPCs, Subnets, Node Pools, Kubernetes Clusters, and databases in this setup. All this can be found in the `/gcp-clusters` in `core-cluster-argo-repo`.

Depending on which cloud provider you are able to create, most cloud resources are offered by that cloud provider.

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

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

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

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

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

9.4. Distributing secrets

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

When Crossplane creates a resource (e.g., Kubernetes cluster or database) on a cloud provider, it stores the connection details (e.g., access credentials) in the cluster where Crossplane is installed. This is a problem since the connection details are needed in `app clusters`, where all the business logic is running. There is no smart, 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 Organisation* [source], and currently, no easy solution exists.

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

There are a few ways of overcoming this secret-distribution challenge. The most naive one would be to create a manual step where the developer needs to somehow copy the credentials to the production cluster whenever he/she decides to create a cluster.

Another way of doing this is using some kind of secret-vault where the credentials are stored during the creation of the database. Each cloud environment can then read the credentials directly from the vault when needed. This may be considered best practice currently, and it comes with some great benefits (which are beyond the scope of this paper) - but it nonetheless introduces even more tools/concepts to our infrastructure, which may put even more workload on a small infrastructure team.

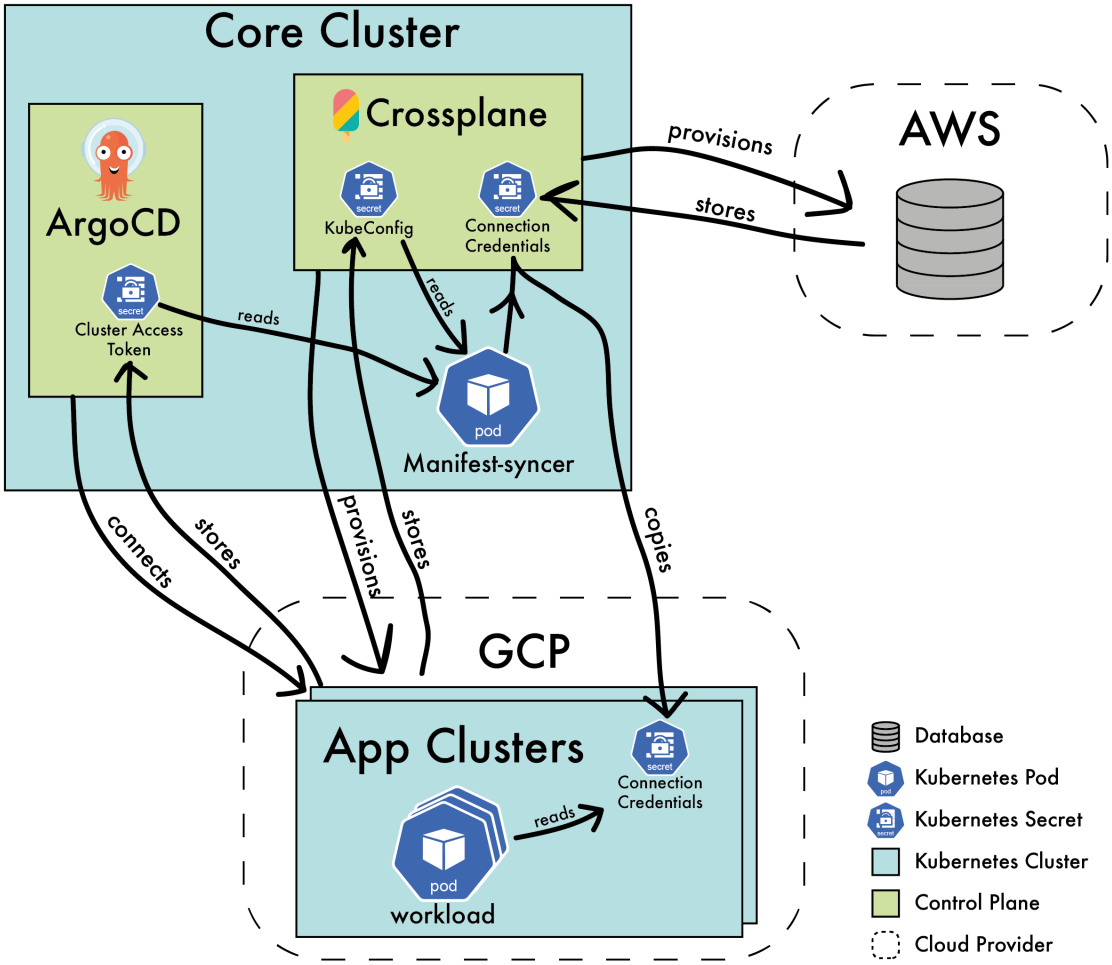
Therefore I have created a much simpler solution.

As usual, there are many ways of closing a gap, but we need to keep in mind that we want a declarative approach, so we want the distribution to happen eventually and not at a specific step in the rollout of a new cluster or new secret. Therefore, I have implemented a service named `manifest-syncer`. The purpose of the `manifest-syncer` is to mirror secrets from their host cluster to target clusters. The service makes use of `CustomResourceDefinitions`, so the user of the service does not need to configure the service. They just deploy the container and create YAML-files describing what they want to be copied. The developer can then declare what he/she wants to be copied and where.

```
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, we can see how we specify that the secret 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` -manifest is *applied* before the `manifest-syncer` is deployed. ArgoCD will fix the failing resources with eventual consistency.



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

9.4.1. Getting access to the App Clusters

In order for the `manifest-syncer` to have access to the App Clusters, it needs a kubeconfig. We don't want to provide/generate this kubeconfig manually each time we create a new cluster. Instead, we want the `manifest syncer` to fix this automatically without having to change other services.

The `manifest-syncer` automatically scans its host cluster for secrets generated by Crossplane with a key named: `kube-config`. . This alone is not enough because it only gives read access to the cluster. To gain write access, it scans, fetches secrets generated by ArgoCD, and "steals" its access token to the clusters it is connected to. The manifest `manifest-syncer` combines the kubeconfig and access token and gets access to the app clusters in that way. This process is done continuously to detect when new app clusters are created automatically.

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

10. Putting it all together

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

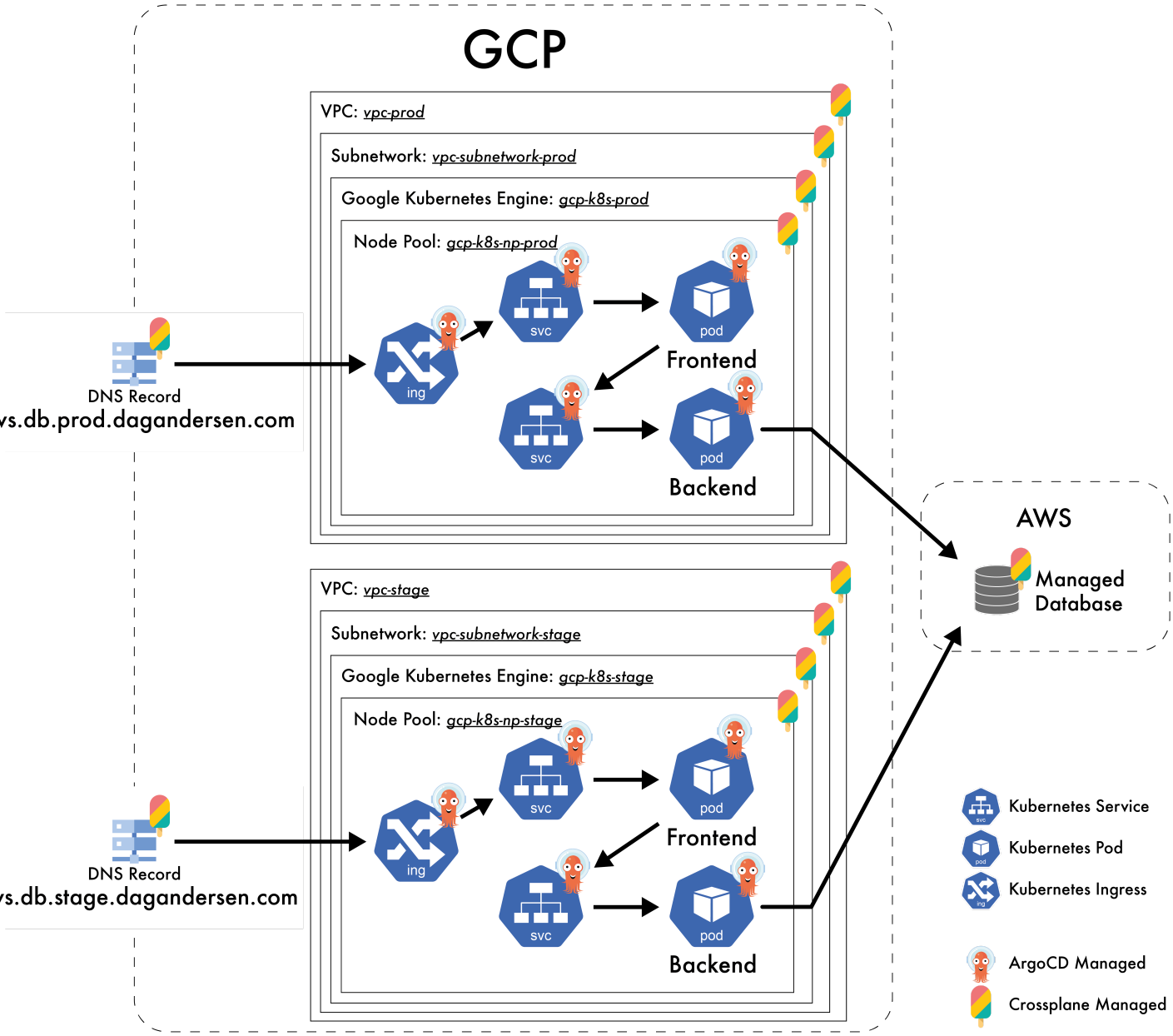
For demonstration purposes, the Kubernetes clusters will run in GCP while the managed database will run in AWS to show that this kind of setup works across different cloud providers. On GCP, there will be two environments running: *production* and *staging*. Each environment runs in its own VPN (and subnetwork) and has its own subdomain on GCP. Both environments can connect to the database running on AWS. This architecture is only for demonstration purposes. This paper is not arguing that this is good software architecture.

The cloud resources needed for this setup are provisioned through Crossplane and can be seen in Figur X. Crossplane does not have a UI, but you can interact with it with `kubectl` like any other Kubernetes resource. Running `kubectl get managed` will print a list of all the resources managed by crossplane.

```
❯ kubectl get managed --context core-cluster
NAME                                                    READY   SYNCED
subnetwork.compute.gcp.crossplane.io/vpc-subnetwork-prod    True    True
subnetwork.compute.gcp.crossplane.io/vpc-subnetwork-stage   True    True

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

NAME                                                          READY   SYNCED   DNS NAME
resourcerecordset.dns.gcp.crossplane.io/aws.db.prod.dagandersen.com    True    True
resourcerecordset.dns.gcp.crossplane.io/aws.db.stage.dagandersen.com   True    True
resourcerecordset.dns.gcp.crossplane.io/dagandersen.com                True    True
resourcerecordset.dns.gcp.crossplane.io/gcp.db.prod.dagandersen.com    True    True
resourcerecordset.dns.gcp.crossplane.io/gcp.db.stage.dagandersen.com   True    True
resourcerecordset.dns.gcp.crossplane.io/prod.dagandersen.com           True    True
resourcerecordset.dns.gcp.crossplane.io/stage.dagandersen.com          True    True

NAME                                            READY   SYNCED   STATE     ENDPOINT         LOCATION         AGE
cluster.container.gcp.crossplane.io/gcp-k8s-prod    True    True     RUNNING   34.163.165.111   europe-west9-a   40m
cluster.container.gcp.crossplane.io/gcp-k8s-stage   True    True     RUNNING   34.163.30.244    europe-west9-a   40m

NAME                                               READY   SYNCED   STATE     CLUSTER-REF     AGE
nodepool.container.gcp.crossplane.io/gcp-k8s-np-prod    True    True     RUNNING   gcp-k8s-prod    40m
nodepool.container.gcp.crossplane.io/gcp-k8s-np-stage   True    True     RUNNING   gcp-k8s-stage   40m

NAME                                                          READY   SYNCED   STATE       ENGINE     VERSION   AGE
rdsinstance.database.aws.crossplane.io/aws-database-postgres-instance   True    True     available   postgres   12.10     10m
```

> Figure X: Here we can see that two VPCs (`network.compute.gcp`), two subnets (`subnetwork.compute.gcp`), six DNS records (`resourcerecordset.dns.gcp`), two k8s clusters (`cluster.container.gcp`), and two node pools (`nodepool.container.gcp`) are running on GCP, and a single database instance (`rdsinstance.database.aws`) is running on AWS.

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



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

The production and staging environment run completely separately on GCP. This project makes it possible to (theoretically at least) scale the number of workload-environments to infinity through the `core-cluster` and *IaC* based *GitOps*.

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

The following section will describe how to use this implementation of a universal control plane and how a software team would develop and deploy services running in `app clusters` such as production-cluster.

10.1. Use in practice

So far, this paper has explained what control planes are and how one can build an entire infrastructure setup only using Kubernetes.

The *implementation*-section described how one instance of a system using Kubernetes and control planes could look, but we still need to cover how we use the setup after creation. As we know, systems are not built once and never touched again. The setup/system needs to be maintainable and modifiable over time.

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

If we want to convince ourselves that this is actually an elegant setup, we first need to envision how it would be used in practice in a company.
In this section, I will go through some examples of how such a system would work in practice if implemented in a company.

The use cases are:

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

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

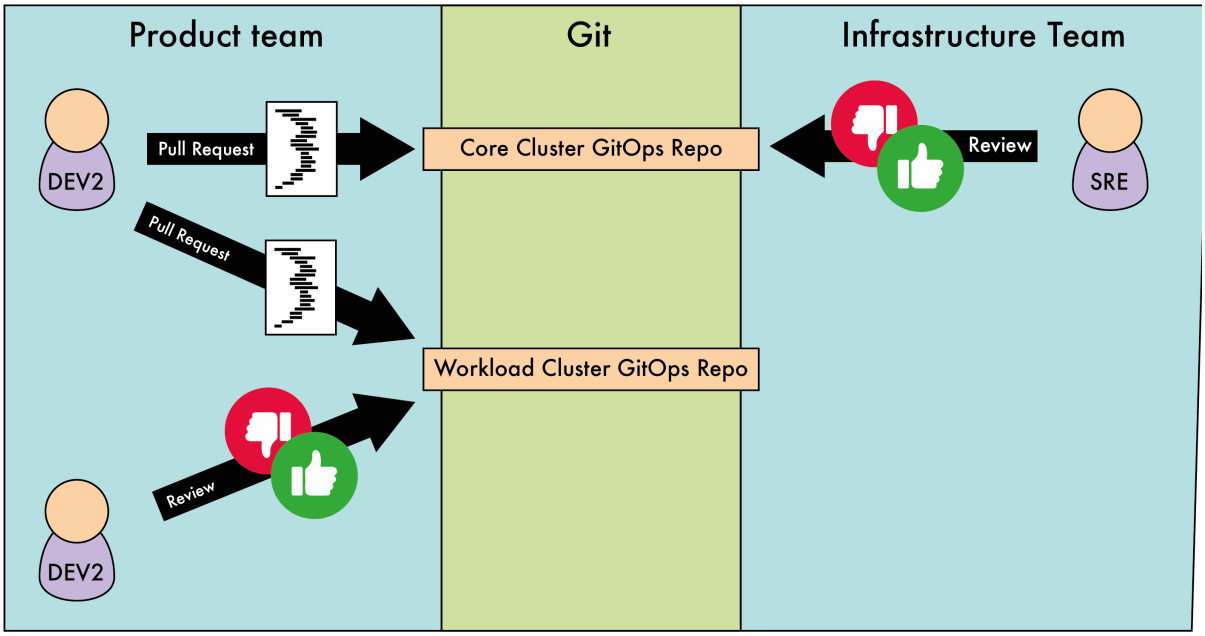
> Keep in mind that many of the tools used in this setup are under active development, so the feature set of these tools may change in the future and change the workflows.

10.1.1. Developer creating a new service with an associated database

So how would the workflow be if a developer wants to create a new service with a new database instance?

1. The developer creates a *pull request* to the `core-cluster-gitops-repo` with the YAML describing the database instance and a `syncer` object for distributing the connection details to the database (as explained in *Distributing secrets*-section)
2. The developer creates a *pull request* to the `app-cluster-gitops-repo` with the YAML describing all Kubernetes objects required. This could, for instance, be YAML describing `ingress` , `service` , and `deployment` as with the demonstration application.

Depending on the policy for this imaginary company, it may be a person from the infrastructure team approving the provisioning of this new database, while it may be someone from the development team approving the normal app-related Kubernetes resources.



> Figure X: This image shows how the developer creates/deploys new services with databases.

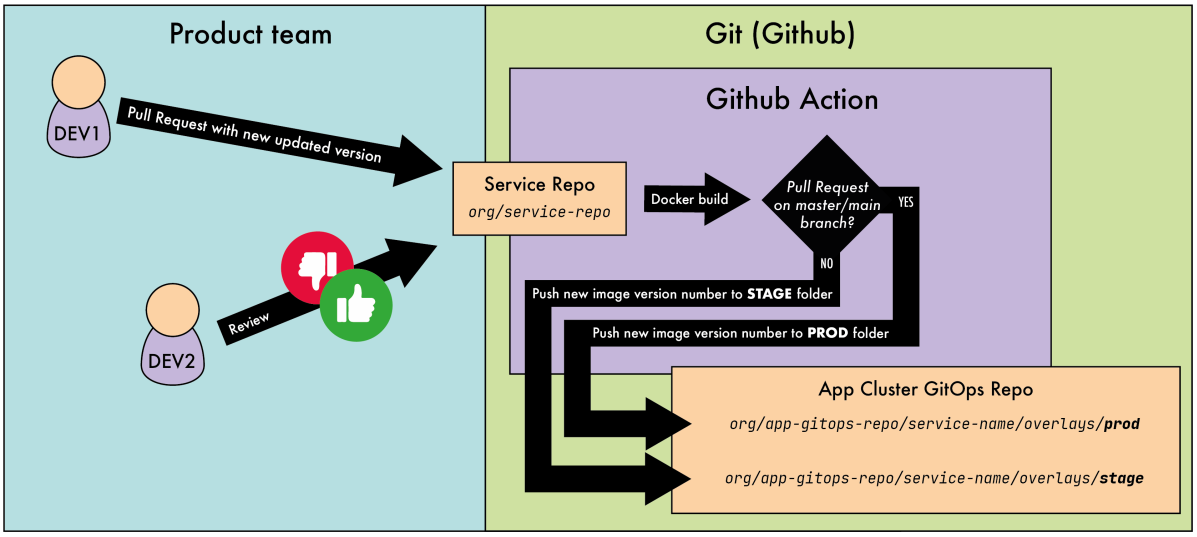
10.1.2. Deploying a new service version to the multiple cloud environments

So, how would the workflow be if a developer wants to update their service on the production/staging cluster?

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

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

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

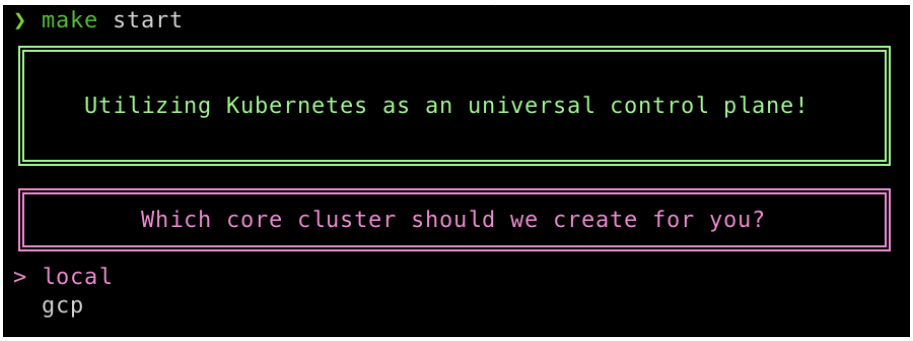


The *Argo Project* (the organization behind all Argo tools) does not provide any opinionated standardized way of pushing a new commit with the new image tag/version. This usually ends up with each organization/team making their own custom code for doing commit push to the GitOps-synced repository.

10.1.3. Spinning up the cluster form scratch

So, how would the workflow be if an infrastructure engineer wants to spin up the entire setup from scratch? Essentially, spinning up a `core cluster` , multiple `app clusters` , and deploying every service/system described in the *managing internal state*-section from nothing.

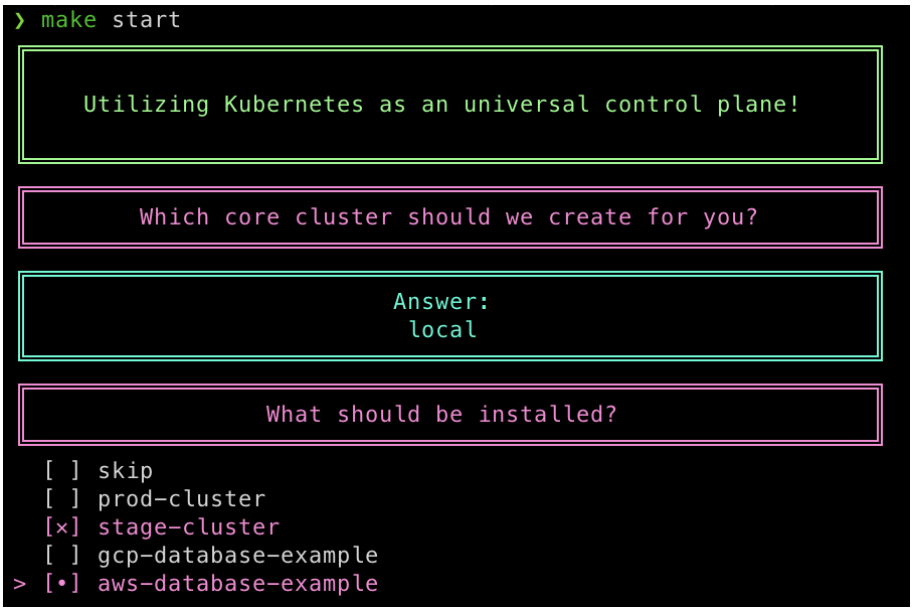
1. Git pull the `scripts-repo` / `bootstrap repo` .
2. Run `make install-tools` in the root to pull down the dependencies.
3. Add your Cloud Provider and Git-repo credentials to the `./creds/` -folder.
4. Run `make start` in the root to start the interactive cli for choosing what resources to create.



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

First, specify if you want the `core-cluster` to run locally with `kind` or if you want it to run on *Google Cloud Platform*.
Other options can be added relatively easily by adding scripts to the bootstrap-repo.

> choosing `local` is preferred when developing since it takes around 10 minutes to spin up a GCP cluster, and it only takes 20 sec to spin up a local kind cluster.



Secondly, you specify what `App clusters` you want and if you want to spin up the demonstration application. If you choose `skip` only the `core cluster` will be started. You can run the `make start` command as many times as you want. It will detect that you already have a cluster running and ask if you want to delete the currently running version.

By default (by choosing `skip`), the script will spin up a local containerized Kubernetes cluster using *kind* and install ArgoCD and deploy Nginx (so you can connect to the cluster). ArgoCD will then apply all the Crossplane-related setup and apply the `manifest-syncer`. Now that the baseline is set, the `core cluster` can be used as a control plane for spinning up external cloud resources and deploying apps to external Kubernetes clusters.

The time it takes to spin up the whole system depends on if you run the core cluster locally or on GCP. Spinning up a GKE cluster on GCP takes around 10 minutes [source]. So, if you run your `core cluster` locally, it results in a *~15* minutes process to spin up the entire system, of which 10 are only waiting for GCP to create the `app clusters`. If you run your `core cluster` on GCP, you must wait an initial 10 minutes before the core cluster is created, resulting in a combined wait time of *~25* minutes to spin up the entire system.

11. Discussion and evaluation of implementation

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

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

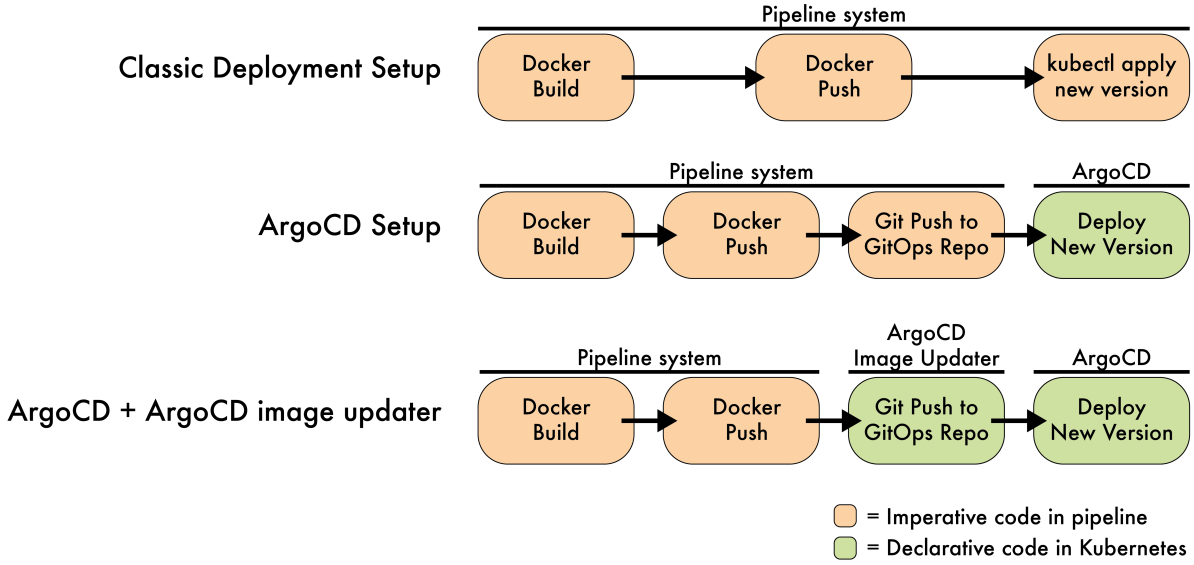
Each topic covered in this section will not be described in detail but will mostly be highlighted as an observation of what to keep in mind when deciding to move to control plane managed infrastructure and what to keep in mind when designing a production-ready system in a company. It is difficult to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

The discussion topics are:

- Build pipelines are still required
- Additional cost
- Not *everything* can be checked into code
- Single surface area
- Platform Engineering
- Pets to Cattle
- Streamlining your codebase
- Bootstrapping Problem
- Multiple core-clusters
- Maturity level
- Eliminating state

11.1. Build pipelines are still required

Switching to a GitOps workflow with ArgoCD or FluxCD will not eliminate the need for pipelines. With the demonstrated setup, you still need some kind of build-pipeline that runs whenever you have a new version of your app.



> Figure X: The image shows 3 different ways of building deployment systems.

Classic build/deploy pipelines with Kubernetes consist of 3 steps: *build image*, *push image*, and *apply YAML changes* to Kubernetes. ArgoCD introduces a new step where the gitops-synced-repo is updated (there is no standardized way of updating this gitops-synced-repo, so it usually ends with a custom script that pushes a new version to the repo). So, with ArgoCD, there are still 3 steps in the pipeline: The last step is just replaced with updating a repo instead of applying YAML changes to the cluster directly.

So, in that sense, we have not optimized much. We still have 3 steps in the deployment pipeline if we use GitOps/ArgoCD, but we have now removed the need for direct access to the cluster from the pipelines. Instead, the pipeline needs access to the repo, where it is supposed to push the changes.

As described in the *Declarative vs. Imperative*-section we prefer declarative configuration when we have the option. 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-repo, `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. We tried the `argocd-image-updater` , but experienced issues connecting to *Docker Hub* (as explained here:). The `argo-image-updater` -project has been going on for at least 2 years without reaching a stable state. So, I will not consider this a reliable option for the time being. The same tool exists in the FluxCD ecosystem [source]. The Flux automated image updater may function as intended, but since we went with the ArgoCD ecosystem, it was not reasonable to try out the flux version in this limited time frame.

So, to conclude: By switching your CI/CD system to a GitOps tool like ArgoCD we have not removed the need for deployment pipelines. Instead, we have only made the deployment process more complex. I would argue the benefits of GitOps still outweigh the extra complexity it creates.

11.2. Additional costs

When using Kubernetes as a control plane, we are running an additional Kubernetes cluster, which is not free. Since this setup entails that we run a `core-cluster` (which does not provide any direct customer value), it will naturally mean that we spend more money on cloud resources than if we did not run a `core-cluster` .

When running the demonstrated setup, more resources were required to run the `core-cluster` than the staging and production cluster combined. The ratio between the resources required by `core-cluster` and `app-cluster` s, will depend on how much workload you run in production. But for my small examples, the resources required by the `core-cluster` surpassed the other clusters combined, so if a smaller 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 only $73,38 (24.46 · 3) [source] for running just the `core-cluster` . 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` .

So, to conclude: Running a `core cluster` is more costly than not running a core cluster. It may not be expensive to run a `core cluster` , but it is something to keep in mind.

11.3. Not *everything* can be checked into code

When creating this project, I wanted to have *everything* declared in code, so I was spinning up everything from scratch with no prior setup.

This was not possible (at least on GCP), because IP-addresses are dynamically assigned to resources unless you reserve the IP-addresses. If you create a Kubernetes cluster on GCP, it will provision a cluster for you and assign it a random available IP-address. This doesn't work well with my setup since creating DNS-records with the GCP-crossplane-provider forces you to hardcode an IP-address into code. So, in this implementation with GCP I needed to reserve 3 IP-addresses. One for each cluster running in GCP.

```
❯ gcloud compute addresses list
NAME                   ADDRESS/RANGE  TYPE      PURPOSE  NETWORK  REGION        SUBNET  STATUS
core-cluster-address   34.155.99.218  EXTERNAL                    europe-west9          RESERVED
prod-cluster-address   34.155.85.111  EXTERNAL                    europe-west9          RESERVED
stage-cluster-address  34.155.198.87  EXTERNAL                    europe-west9          RESERVED
```

> Figure X: xxxx

Those 3 IP-addresses are hardcoded into the DNS records declared in YAML and in the Nginx-Controllers deployed by ArgoCD. Nginx needs to know which reserved IP-address to use, otherwise, it will just pick one at random.

So, to conclude: With GCP and Nginx, you need to reserve IP-addresses beforehand if you want the rest of the system to be fully declared as code. If the DNS records were not needed, then the reserved IP-addresses and hardcode IP-addresses were not needed,

and *everything* could be stored as code.

11.4. Single surface area

Running a business often requires many sets of logins to many different platforms and tools. Not all platforms/tools have a nice way of integrating with each other, so oftentimes, employees must juggle many different sets of credentials. Giving each employee the correct amount of access rights can be a hassle. Very strict rules can result in slow development because you constantly need to request access to new resources should you need them. Very loose rules can result in security vulnerabilities because each employee has access to way more than they need (a violation of the Principle of least privilege).

Terraform on its own does not have a concept of access control [source] - You must manage your access through credentials through the cloud provider. This works fine, but it is an extra set of credentials your organization has to manage for all your developers.

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."[s]. All your access control can be moved to ArgoCD/kubectl - making Kubernetes the only surface requiring access control. This is also covered in Eficode's own article: "Outgrowing Terraform and adopting control plane".

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

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

11.5. Platform Engineering

Platform engineering is gaining popularity in the last two years [s]. This topic is a main topic in itself, and not the focus of this paper, but we just quickly want to highlight how control planes like *crossplane* can modernize your infrastructure, by embracing Platform Engineering and self-service. Crossplane has the concept of Composite Resources, that works as an abstraction layer between the actual managed resource running on the cloud provider and the resource offered by the infrastructure team to the development teams. The developer then does not have to worry about where the database runs. The developer just requests a database in YAML, 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.<<*" [source]. 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 [source].

So, 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 organisation and how they are configured behind the scene.

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

One challenge big companies can have when their developers have direct access to the cloud resources is that cloud services get created and forgotten about. 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 un-utilized 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 it is no longer or never was checked into git). This is great for getting an overview of resources stored as infrastructure as code. 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, you may think that you can re-create your production cluster without any issues, but in reality, your services depend on a resource that was created manually and never checked into git.

If you lose trust in the reproducibility of your infrastructure, you start treating your infrastructure as pets that you have to protect at all costs. Having a `core cluster` that manages many `app clusters` helps you go from *Pets to Cattle*. The easier it is to spin up new clusters, the less we will treat our infrastructure as "pets" [source]. This paper's implementation can manage an infinite number of disposable `app clusters` .

So, 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 builds confidence in the reproducibility of the system, by creating visibility of what cloud resources and what software is provisioned/deployed from git.

11.7. Streamlining your codebase

The setup described in this paper is built using only 2 file types: `.yaml` and `makefiles` . `.yaml` is used for declaring the state of the entire infrastructure-configuration, while `makefiles` are only for the initial bootstrapping. Everything is declared as *Infrastructure as Code* and checked into git (*GitOps*).

"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" [source]. This creates a highly streamlined infrastructure because it does not require knowledge about e.g., Terraform, Ansible, or multiple scripting languages. This is a huge benefit of this setup.

So, to conclude: Building your infrastructure using control planes in Kubernetes (like Crossplane and ArgoCD) ensures your entire infrastructure is declared in YAML. Your infrastructure team can define the entire setup in YAML and does not need to know other tool specific languages.

11.8. Bootstrapping Problem

Every automatic process requires an initial manual task to start the process. This initial task/command cannot be declarative since a command is, by definition, imperative.

In order to simplify the setup process as much as possible, we aim to make the bootstrapping as simple, clean, and error-safe as possible. The only bootstrapping done in this implementation is starting a cluster (used as `core-cluster`) and then installing ArgoCD on it. The rest of the setup is handled automatically by ArgoCD with a declarative/eventual consistency approach.

The bootstrapping in this implementation does not touch logging, monitoring, ingress, Crossplane-related-stuff, Helm Charts, etc., which you could be forced to install sequentially through an imperative script.

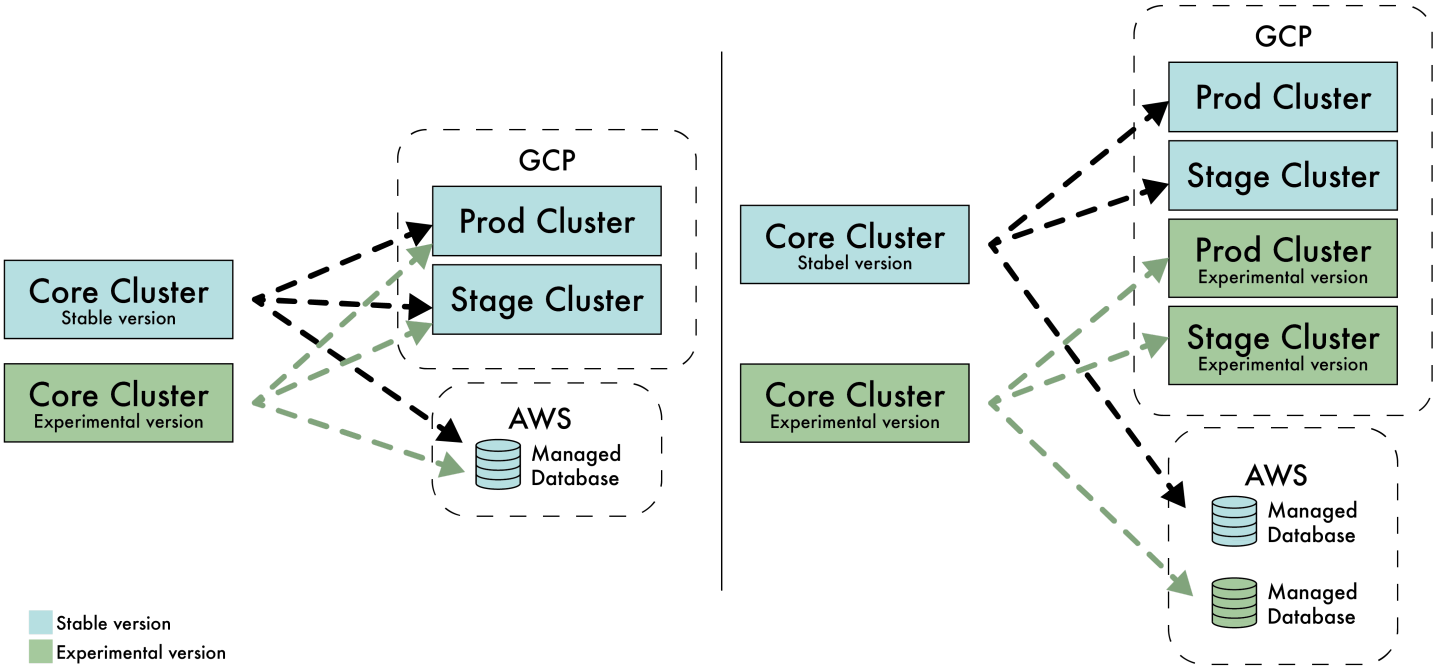
So, to conclude: We can't remove bootstrapping entirely - but we can try to reduce it as much as possible. I would argue that the bootstrapping done in this paper's implementation is fairly minimal since it only does 2 steps. *1*: Spin up core-cluster (locally or on cloud provider), *2*: Install and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

11.9. Multiple core-clusters

Just like you want a production cluster and a staging cluster of the app clusters so you can test software on the staging cluster before it goes into production - you probably also want to test software before it goes into production in the core cluster. E.g., how do you test a new version of Crossplane. It can be risky updating software on the core cluster because the app clusters depend on it. This would indicate that you probably also want a staging/testing/experimentation version of the core cluster, where you can test and experiment with software before it goes onto the stable version of the core cluster.

Running multiple instances of a `core-clusters` simultaneously does not work well. You can easily spin up multiple `core-cluster` s 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 create a given resource on a cloud provider, while another may want to delete that resource. This results in race conditions and unpredictable behavior.

So, the next question is how you test a configuration for the `core-cluster` if you cannot spin up a new one without it competing with the current version running. The only way to allow multiple `core-cluster` s to be run simultaneously is to run a complete copy of all your resources. So, the copy could be named `core-cluster-experimental` , which is then created, e.g., a `prod-cluster-experimental` and `aws-database-experimental` and so on. It is fairly straightforward code-wise, so it is doable, but this would effectively double your infrastructure costs if you wanted to do proper tests and experiments on the `core-cluster` .



But the biggest drawback of running a complete copy of your entire infrastructure is that it adds a lot of complexity. Especially when it comes to managing a separate configuration for a duplicate cluster. One has to figure out a proper way of telling `prod-cluster-experimental` controlled by `core-cluster-experimental` , to use the ip-address/hostname of the `aws-database-experimental` and not the normal `aws-database` . Everything is 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 you organization handles. If you at the same time want to automate this process it gets even more complex. Solving this issue is beyond the scope of this project.

So, to conclude: It is difficult to test/experiment with the core cluster. Depending on how you do it, running two at the same time will either create race conditions or duplicate your entire infrastructure which may be unfeasible or unmaintainable depending on the size and budget of your organization. There is no good solution for this and I consider this one of the biggest drawbacks of running a `core-cluster` for managing the rest of you infrastructure.

11.10. Maturity level

Currently, the biggest limitation of using Crossplane is the lack of providers and the feature-set each provider offers.

When using Kubernetes as a universal control plane for all of your infrastructure, you rely heavily on the stability and flexibility of the controllers made by big cooperations or open-source communities. When you use ArgoCD, you put all your faith in that its control plane correctly deploys your services and does not randomly delete arbitrary pods. When you use Crossplane, you rely on the controller/providers to correctly provision the requested resource and manage their life cycle. As a user of these control planes, it is out of your hands, and you rely solely on the tools. This is the same limitation that Terraform has. Terraform is only as good as the providers that integrate with Terraform.

Both Crossplane and ArgoCD is marked with maturity level Incubating, which is meant for "Early Adopters" [s]. So, it is not expected that the tools give an flawless experience.

Examples of observed issues when working with ArgoCD and Crossplane

- Digital Ocean's Crossplane provider (v0.1.0) cannot delete resources on their platform (Issue is reported here [s]). This means that Crossplane can only be used to spin up, e.g., databases and Kubernetes clusters – but not delete them afterward. This makes the provider nearly useless because you cannot control the full life cycle of resources. This will probably be fixed in the near future. However, this is a good example of cloud providers not being mature and ready for control planes like crossplane.
- ArgoCD cannot natively deploy resources when the generated YAML-files get too large (Issue is reported here [s]). This is completely out of your hands. So, if you want to deploy certain helm-charts that generate long YAML-files, then you instead have to find a custom workaround online or deploy it manually. This can be quite painful since if ArgoCD cannot deploy *every single resource* declared in your infrastructure you must introduce custom logic for edge cases, which doesn't scale well.
- ArgoCD cannot connect to an external cluster based on data stored as a secret in Kubernetes (described here [s]). Argo can only connect to external clusters by running `argocd add cluster <kubeconfig-context-name>` on your machine with a kubeconfig available. This goes against the idea of declaring everything in YAML by forcing the user to call a shell command imperatively (described [here]).

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

11.11. Eliminating state

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

With Terraform, you store a state each time you provision any cloud resource. The resources must be stored in a shared place, and you must make sure it is up to date and that only the right people have access to it. Crossplane doesn't handle state in the same way. Terraform looks at the Terraform state, your local code, and what is currently running in the cloud. Crossplane only looks at what is currently running in the cloud. If the requested resource does not exist, Crossplane will create it. If the requested resource already exists, it will not create anything. This works well with simple resources like DNS-records on GCP, but if you look at resources that need connection details, like managed databases and Kubernetes clusters, it gets more interesting! What the Crossplane provider does with each resource, if it already exists, depends on the individual resource.

If Crossplane requests a GKE cluster *that does not already exist* it will provision a cluster and store the connection details (kubeconfig) as a secret in the cluster with Crossplane installed. If the GKE cluster already exists, it will not create anything but simply pull down the connection details (kubeconfig). That meant that if the `core cluster` gets deleted and recreated, the kubeconfig to the `app clusters` *will not* be lost.

This can also be seen if you manually delete a secret generated by Crossplane. Crossplane will simply detect it and reconcile and recreate the secret. No state lost.

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

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

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

12. Conclusion

13. References