NOV/2020

Cloud-Agnostic Kubernetes
Demonstration of Implementation

Michael Haddon

INTRODUCTION

This document details a demonstration of a Cloud-Agnostic Kubernetes cluster, its goals is to expand on the architectural decisions as-well as the required and recommended work to make the implementation production-ready.

It should be stressed that this project is an intentionally incomplete solution.

REQUIREMENTS & CONTEXT

The context for this project is that our project's environment should be redeployable on different cloud/hosting providers.

The requirements, therefore, are as follows:

1. We must deploy and configure Kubernetes in a fashion suitable for our context.
2. We must deploy a demo application onto the cluster which is exposed to the internet.

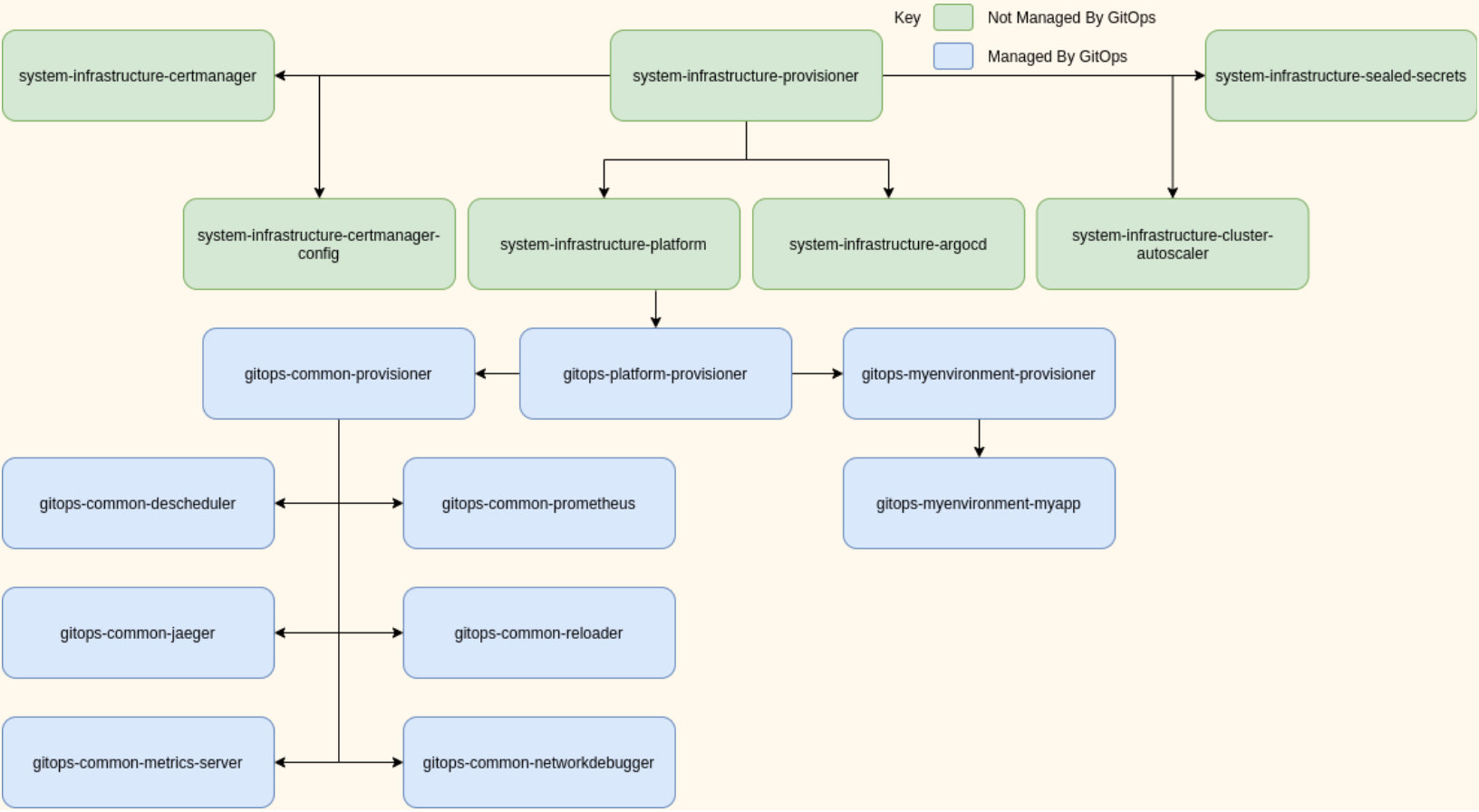
REPOSITORY LAYOUT

The project repository's "git-repository" folder resembles a multi-repository configuration (https://github.com/mhaddon/example-kubernetes-setup).

The repositories can be divided into two distinct groups:

1. The ones installed manually, or from a custom script. The "System" repositories.
2. The ones installed automatically from the GitOps controller. The "GitOps" repositories.

The cluster can be built by running `terraform init && terraform apply` from the "env/dev" folder in the "system-infrastructure-provisioner" repository.



ARCHITECTURAL DECISIONS

To meet these requirements several major architectural decisions were made, this section elaborates upon those.

Self-hosted Kubernetes over Kubernetes as a service

The decision to host Kubernetes was taken because this environment needs to be deployable on different cloud/hosting platforms and behave as similar as possible. This allows us to ensure that we have the same configurations for the Kube-API and the same dependencies such as the CNI.

I would heavily recommend that the “Kubernetes as a service” solutions should be seriously considered.

GitOps & ArgoCD

GitOps is an extremely powerful solution to ensure you exercise the full potential of the intent-based provisioning solution that is native to Kubernetes. It allows you to keep greater control over what is running on your cluster, creating a strong audit log and reduces the mean time to recover from incidents.

ArgoCD was chosen as the tool to implement GitOps as it is very well made and has an easy interface that allows engineers to control and visual their applications deployment.

Terraform

Terraform is a very versatile configuration management solution. The benefit in our situation is that we can create different Terraform implementations for each cloud/hosting environment to ensure that Kubernetes always has the same implementation no matter what provider it is installed upon.

Prometheus

Prometheus is a very powerful tool that allows us to aggregate important metrics, utilising these we can monitor deployments and ensure high availability.

Sealed Secrets

Sealed-Secrets allow us to manage our secrets directly from our source-code. This means we get all the benefits of GitOps in a simple and concise format. The master private key for Sealed-Secrets is stored on AWS in AWS-SSM, and is imported on cluster creation.

Service Meshes & Istio

Service Meshes allow us to abstract security, networking & observability logic out of our application and onto the infrastructure. This greatly enables us a high level of control over our cluster and can severely reduce mean time to recover from incidents.

Istio is the service-mesh of choice due to its community-support and functionality.

IMMEDIATE CHANGES FOR DEMONSTRATION

This demonstration is missing a few features which were intended to be implemented, if not for time constraints. They may be implemented though by the time this document is read.

Configure observability/monitoring tooling for our demo application

The system lacks proper integration with observability tooling and we lack alerting upon incidents. A tool like Grafana should also be installed so this information can be properly visualised.

Integrate Argo Rollouts with Prometheus

Currently Argo Rollouts is configured to require manual intervention with a canary release. This should be handled automatically according to the Istio metrics collected by Prometheus.

Install Jaeger using the Operator Lifecycle Manager

The Jaeger operator already has quite a stable implementation and so it could be wise to utilise it.

IMPROVEMENTS ADVISED FOR PRODUCTION

High-availability master nodes configuration

Currently there is only one single master node. This can cause a huge issue. We should have (at least) 3 master nodes running across 3 AWS availability zones.

If our Kubernetes cluster is not ephemeral then we must also ensure we have etcd backup-solutions implemented.

Mutual-TLS enforced

TLS communication between applications should be enforced as much as possible. This would lay the foundations of building an infrastructure with Zero-Trust principles.

Enable SELinux

SELinux was set into permissive mode to avoid having to configure the policies for the demo. This should be properly configured to ensure a higher-level of security and set to enforcing.

Restrict Node Permissions

Currently the nodes have all the AWS permissions that the containers need. This is bad. We should attach the AWS roles to Kuberetes service accounts, and use that to isolate who has permission to what.

Restrict “my-application-domains” ArgoCD project

For security reasons the developers working on their “my-application-domains” and “my-app” GitOps applications should be unable to modify resources outside of their respective namespaces. In order to achieve this though you may end up implementing multiple ArgoCD implementations.

Configure istio-gateway to not run as root

To keep our port-bindings simple it is currently running as root. This should be changed.

System policies

We should configure relevant network, system and security policies with tools like Calico, OPA & Falco. Compliance of these policies should be easy to visualise, audit and alert upon.

Deletion of cluster should smoothly clean up Kubernetes-created resources

Currently when you delete the cluster aws resources such as the NLB+Route 53 records, are not cleaned up correctly. This conflicts with Terraform and causes it to break. A cluster should be as easily destroyable as it is creatable.

Secure management applications behind central authentication provider

Currently Argocd is publicly exposed, anyone with the url can access it and administer it, this is intentional for the demo. However it, and other management applications should be publicly accessible and secured with some sort of central authentication provider.

LONG-TERM IMPROVEMENTS ADVISED

Migrate Helm charts, where possible, to operators

Operators are a much more powerful and versatile tool, however, most of the various implementations are still in early development.
With the Operator Lifecycle Manager, operators can fully control the implementation of a service, grant you native-level integration and automatically manage their own updates and configuration.

Restrict “ssm-agent’s” permissions

Currently we can interact with our cluster through the AWS Session Manager. This allows for full administration control over the cluster, which is not only un-necessary for an ephemeral cluster, but also poses potential security risks. It could be decided to go down the route of “in-accessible infrastructure” and ensure that no connection can be made at all. This could also mean that the access to the “centos” account is also disabled.

Create CI/CD pipelines for the demo app

Currently the demo-app relies on a manual script to be run for its container to be created, furthermore, it also depends on a manual step to increase the container version in git.

This process should be automated with reliability & security testing built into the process. For security reasons, solutions that can run from within the Kubernetes cluster are the most preferable.