Case Study: Design extensible kubernetes infrastructure in AWS

See an up to date, HTML version of this case study on andrewhowden.com

Skills

This case study required:

- An in depth knowledge of Kubernetes components
- An in depth knowledge of Linux, bash
- The ability to quickly learn and apply Terraform to a production system
- The ability to reconcile multiple sources of truth into a single production infrastructure specification
- The ability to with with colleagues with different specialisations to bring together a large, production cluster

Requirements

As part of some work in conjunction with a major eCommerce hosting parther the underlying infrastructure on which their systems are executed was being rebuilt. Primarily this was because the partner could not tolerate a large amount of risk while upgrading their clusters and their clusters are both fairly large and made up of very large machines with multiple services executing on each machine.

However, a new cluster design presents the opportunity to reduce maintenance cost and try new features available in the Kubernetes landscape such as Cilium, IPVS in the kube-proxy layer and other ways of managing the system.

Definitions largely existed for the cluster as well as a solid plan for the design, workload placement and so on. However, in the past parts of the AWS infrastructure was either manually managed or managed in CloudFormation templates that were in turn managed with custom tooling that was less well understood.

Plan

The plan with the redesign of the cluster was to significantly reduce the cost of maintenance for future clusters so that team members could be allowed to spend their time designing new features, rather than solving and even resolving bugs on the existing cluster.

To that end, the following goals were defined:

1. The technology used to specify the cluster should be widely used and have a low barrier to entry
2. The cluster itself should be designed in a way that this technology was not responsible for the actual cluster specification, but rather for templating images that could be used to expand the cluster as required.
3. The definitions were under version control, and deployed either via CI/CD or in a systematic way (for example, on a given branch))

The technology "Terraform" fit this criteria very well. Though I did not know the technology well at the start of the project it followed principles otherwise common in infrastructure as code tooling such as idempotency, declarative specifications and a dependency graph. Further, it is in wide industry use as a "common" tool for managing cloud infrastructure and has a low barrier to entry and superb documentation.

Implementation

Reviewing existing definitions

The first requirement of this work is to get the cluster definitions as they previously existed in version control, AWS and other sources into a single terraform repository. This included reviewing the existing tooling to generate the CloudFormation templates, diffing that output against the production systems and translating that configuration into terraform definitions.

Practically this meant defining modules for the worker nodes, master nodes and other required supporting services.

Draft cluster

The next phase of the work is to launch a new cluster that is of the same specification as the previous clusters and determine if there were notable differences between what was created in Terraform compared to what existed in production in the other regions.

There were several minor issues that needed to be addressed, as well as some changes to configuration management templates in Salt.

Feature Development

The next phase of the work was to iterate on the existing design of the cluster to attempt to resolve some of the gnarlier issues that were harder to address with the previous cluster. Originally slated were:

- Switch from Calico to Cilium
- Switch from iptables to IPVS
- Network topological changes to remove redundant resources and remove absolute complexity
- Improvements to network security tooling (NetworkPolicy, inter-node communication)

These changes proceeded smoothly in development. The new cluster was load tested at ~20k services and hundreds of pods per node and appeared to be functioning correctly.

Production Rollout

While rolling out the changes to production systems there were some unexpected issues:

- Cilium did not bootstrap correctly as it was unable to connect to etcd, in turn bootstrapped by the clsuter reliably. This has been resolved in 1.6 by using CRDs instead of etcd but was too late in this iteration of the cluster.
- IPVS consumed a very large amount of CPU iterating over virtual interfaces

Accordingly, both of those changes were reverted as the new cluster rolled out. They are still planned for future infrastructure changes but missed this design of the cluster.

Beyond that, the new cluster was rolled out fairly successfully with minor issues where configuration drift from the previous cluster had not been committed to the new cluster, upgraded version of software (NFS/ZFS) did not function exactly as expected or assumptions in the network topology reduction proved incorrect.

Evaluation

As the cluster reached production several issues were discovered that meant some features needed to be rolled back to the previous cluster design. This introduced some of the same problems as the previous cluster has, though those problems were mitigated by upstream changes in various system components that were upgraded with the new cluster.

With that said the new cluster definitions reduced the cost of replicating changes around various world regions as well should reduce the ongoing cost of maintenance for this system. The terraform definitions are now the authoritative source of change management for cluster infrastructure.