Clark Bains
COMP 4000
November 23, 2021
101149052

1 The Project

Back in August, Instructor Dave Mckenney reached out to me, knowing I was interested in server maintenance and deployment, and asked if I would be willing to manage the openstack project for the Carleton Web Development Club (CWDC) that he runs, and that I had worked with on a few occasions. The club consists of a few core members and hundreds of inactive members, all of who we wanted to be able to provide a platform for them to deploy their networked applications on, such as dynamic portfolios, games, and other fun projects. I agreed to do it, and we agreed on a few principles for how it should be set up.

- I should easily be able to allocate resources for students, and give them an environment to run their code in.
- It should be simple to get started on. Not everyone will have extensive Linux experience.
- It should be somewhat secure
- It has to work for students outside of the School of Compute Science.
- It has to run on Openstack

When I first started, we were given a fairly odd allocation of resources, with the major constraint being only 8gb of storage on each of the 6 cluster nodes. This presented a lot of challenges, and not knowing much better, I setup a manually administrated environment, and wrote many scripts to help students do what they needed to do. It wasn't till I met with Forest Anderson, the sysadmin for the Carleton Computer Science Society (CCSS), and saw what they were doing that I figured out that there was a far better way to do it. I then migrated all the existing services to the CCSS cluster, obtained far more Openstack resources, and started taking time to overhaul the CWDC cluster. I added the following requirements:

- Should be easy to provision new servers
- Security should be a far higher priority
- Students shouldn't have to do *any* scripting to get up and running
- Some systems should be redundant (Storage, cluster management)

Kubernetes seemed like a mostly good solution to this, however I personally find Kubernetes configuration to be unintuitive and cumbersome, and I did not want to expect CWDC members to write deployment files for it.

2 The Original Design

2.1 Provisioning

Initially, I did not have the tooling, resources or know how to start building the cluster, and configuring services on openstack. Given that I have a powerful home computer, I wanted to replicate final openstack cluster on it, using Virtual Machines. I found out about Hashicorp's Vagrant, and started using that. It would allow me to write Ruby code to programatically define VMs, from their base image, to persistent storage, resources and network. It could then interact with a virtualization solution, such as libvirt, which I already had installed. Opting to use a tool to automate the VM lifecycle meant I could rapidly destroy and recreate my VM when I needed to try a different configuration, or be sure that a new process was the sole reason something would work, or not work.

To complement Vagrant, I wanted a solution to configure the VMs once they were created. While Vagrant has the ability to run scripts on the created VMs, it was an elegant solution for me, as I wanted much better control and coordination between nodes when configuring. I wanted to use Ansible for this, a python tool that primarily reads two files, one describing the groups each node falls into, and another describing commands to run on each member of a specified group. With this I would, for example, be able to read an encryption key out of a file on one node, store it in a variable, then insert the variable into a templated configuration file that gets sent to a /etc/ subdirectory on a set of nodes.

These two pieces of software would make the initial setup process as easy as destroying the existing VMs and then recreating them with Vagrant, then running Ansible to configure all software elements within the cluster, manually testing to find any issues with the existing config, adding new tasks into the Ansible configuration, and retrying.

2.2 Service Discovery + Distributed Consistent Key/Value Store

To achieve my redundancy goal, I would need a framework to help keep track of all the services/projects currently running and their health. I chose Hashicorp's Consul to do this, as it came with other features and integrations that could prove to be useful. Consul creates a network between all nodes in the cluster, and offers service discovery, where a service can be registered with the Consul network by giving it the node(s) and port(s) that the service using, and optionally a health check. It will then keep track of what instances of the service are healthy (passing the health check), and expose them over a conventional API, and also a DNS interface. In addition, it provides a key/value store, and keeps everything consistent between nodes.

Being able to register services and store custom metadata for them is crucial for this club, as it allows for a ingress reverse proxy to have a single authoritative source for where to route ingress traffic.

2.3 Secret Storage

I wanted to provide a distributed, encrypted data store, primarily for the core infrastructure, but also for students who require it. I used Hashicorp's Vault for this. It can run in a High Availability (HA) mode, storing data on Consul (which is also HA). It exposes an API to get and set keys and values, though with much more granularity than directly through consul. Vault also integrates with ACLs in other services, so that you read a specific key, and Vault will get you a token for another service with a predefined access level.

By registering itself with Consul, other services can use Consul's service lookup (works via DNS queries, e.g https://active.vault.service.consul) to find active Vault servers to interact with.

2.3.1 HA Auto Unsealing

Since Vault encrypts all data, to be secure, on startup it requires a number of keys to unseal it before it can respond to any requests. When using HA, this unseal process must happen on each vault HA node. A production system should see these keys distributed among trusted individuals, so especially during the production-but-still-testing state, I wanted to mitigate the risk of sealing the vault (ie for a minor config change, which requires restarting the vault service) and having to wait for the threshold number of keyholders to be available to unseal all HA vault nodes again. This would also require giving access to the Vault servers to multiple people, which isn't great from a security standpoint. To work around this, I wanted to use a second vault instance, not in HA, to store the encryption keys required to unseal the primary Vault. As long as the secondary Vault transit store stays unsealed, changes can be made to the primary HA vault cluster without requiring the threshold number of keyholders.

2.4 Job Scheduling and Running

Since at its core, this cluster serves to run networked student projects, I wanted a way to actually run everything. Docker was the apparent solution, as it provides a way for students to develop in basically the same environment their code gets deployed in, reducing surprises. Docker however lacked the integrations, security and scalability characteristics I was looking for. I ended up choosing Hashicorp's Nomad, which sits as a layer on top of Docker. It allows the simple docker-compose like configuration, something I've personally come to love, and integrates some more advanced features from Kubernetes. It accepts files describing a task to run, and a way to run it, in my case a docker image, and any required command line options and environment, as well as the network and storage resources it requires. It allows services to be scaled up with multiple copies of the task.

Being a Hashicorp solution, it integrates extremely well with both Vault and Consul. Consul integration allows it to publish information about running tasks into the Consul Catalog, meaning other services can use Consul to find the ip and port of healthy instances of the service to connect to, just by using the service's name. Vault integration allows data to be injected into templated enviroments and files. This way, a user could opt to specify to get the value for an environmental variable from the associated value for a vault key. Since the Nomad job specification files are going to be a new technology for many CWDC members, this allows users to freely share their real job specification files (since there are no secrets), which will allow other members to get started quicker.

2.5 Mass Data Storage

Since Docker was chosen as the Nomad execution environment, and Docker uses ephemeral containers, I needed to have some form of networked, persistent, redundant data store. For this, I chose Ceph. Ceph uses multiple storage controllers, and Object Storage Daemons (OSDs), with the former responsible for coordination, and the latter for persisting data stored in the cluster to their own disk. With only one Openstack networked volume to attach an OSD to, I would have a single point of failure. Since many other nodes in the cluster will have remaining disk space, I could use files mounted as loopback devices. I can make these apear as if they were normal disks attached to the system, and run more OSDs attached to them, which would help the system be resilient to a single host failure.

Additionally, Ceph provides a Container Storage Interface (CSI), which allows it to be used as persistent storage for the jobs running on Docker, through Nomad. Using the CSI means finer access controls, and offers a more elegant solution compared to using Docker bind mounts to go to some redundant filesystem on the host. The CSI for Ceph communicates directly over the network, meaning that in the event running student jobs need to be moved, or scaled off of one node onto another, they will still be able to access all of their data from the new node.

2.6 Reverse Proxy Ingress

Since the CWDC is a Web Development club after all, most projects will be exposing some form of http service. By default, web traffic runs over a single port, 443 for secure https, and 80 for insecure http. Since only a single process is able to bind to this port at a time, I would not directly be able to have all services/projects listen to internet traffic on these two ports. To get around this, internally, I have everything listen on a predefined or random port, and I then use a single process, listening on both port 80 and port 443, and give it a set of rules about how it should analyze the incoming network traffic, and where to proxy the request to. This is the core job of a reverse proxy.

I chose the Traefik reverse proxy, as it is highly automated, and requires very little manual configuration. Trefik integrates with the Consul Catalog, which means it has access to the custom metadata for each service, along with the ip and port of all healthy instanced of the service. By storing rules for what traffic gets proxied to the service in the Consul Catalog service custom metadata section, for example, having a Host header of myApp.cwdc.scs.carleton.ca, no additional configuration for each service needs to be manually added into Traefik. In addition, it is able to round robin route traffic between the healthy instances of each service, distributing the traffic evenly

Additionally, Traefik offers letsencrypt integration. In order for a site to be served via https (encrypted, and secure for transmitting passwords and personal information), without showing a large warning in the end user's browser, the site must have valid Transport Layer Security (TLS) certificates. These TLS certificates are given to site administrators by external companies, after they verify that the administrator does own the domain the certificate is for. With letsencypt integration, whenever a new service is added that would like to accept traffic from a specific domain, say app2.example.com, Traefik will connect to letsencrypt, request a certificate, and then letsencypt will verify that traefik does have control over the domain by responding with a long string of random characters, and then verifying that, for example, app2.example.com/.well-known/acme-challenge, returns that long string. Since traefik sits in front of the service, it is able to see this request, and handle it internally. This all means that traeffik can get the certificates to use https without users seeing any certificate errors in their browser, and for CWDC students, it means an effortless option to make their project more secure. In addition, since by nature a reverse proxy has to read the http requests to decide where to route them, it is able to accept the encrypted https traffic from the internet, yet only proxy it as if it were http traffic to the service, which is easier to implement than trying to decrypt https traffic.

Traefik also can read and store data in the Consul Key/Value store, which means it can be run in a HA pair mode, meaning that traffic can still be routed to services even if the primary Nomad server goes down.

2.6.1 User Submitted Jobs

Since one of the goals of this project is to allow students to easily deploy their own project, I chose to use Hashicorp Waypoint. It comprises of a single static binary, that acts as either a server an client. The client can package and publish source code in a variety of different formats, including docker images. The server interacts with Nomad, among other platforms, and allows the clients to automatically deploy their packaged application following a nomad job specification file. Waypoint server also provides the capability to do remote docker builds on the cluster itself, so students would not need docker installed to build their projects. It also offers a poll function, which can poll a specific git repository and auto-build and deploy projects whenever a branch is updated.

2.7 Security

All services with ACLs should have them enabled, and fairly restrictive. Once a user gets access to the cluster, the design of the system should involve very little manual oversight from me, so it has to be pretty secure.

As an example, I could imagine a insecure Consul instance being connected to, and a user could redefine the service for the club site API, and Traefik would pick it up and have all requests (including authentication), go to the attackers own servers, and they could steal credentials. While I think this is extremely unlikely, adding ACLs makes the cluster more secure against attacks like this.

To protect against users who decide to be malicious, or get hacked, I would like to implement a few mitigations.

- ACLs on all services that support them
- System, core, and user "Zones" where ACLs are not adequate, so that more trusted, core services (like the club API and frontend) don't rely on the same instance of services that could otherwise be modified by a service in the user zone.
- Internal services and dashboards should either only be accessible via SSH tcp forwarding, alternatively protected in some other way and at most, only be accessible from within the Carleton network.

Since the entire group runs via a Discord server, I planned to write a discord authorization middleware for traefik. The middleware, like its name implies, sits in the middle, between incoming requests and the service they get routed to. It receives a copy of incoming requests from traefik where the middleware is enabled, and then

3 Current Implementation

I have spent many late nights working on this, so I am mostly done. All that remains is fine tuning. I have the configuration to wipe Openstack, provision the cluster, have my primary vault unseal using my secondary vault, run a waypoint job, on nomad (which relies on consul), and have it access a volume mounted from ceph through CSI. It would then get an automated SSL certificate and be available on the public internet through traefik.

In my current provisioning implementation, I have nearly 1000 lines of ansible configuration, representing 156 different named tasks (and likely a few more I neglected to name) 700 lines of nomad/vault/consul HCL configuration, with additional scripts to generate a /etc/hosts file based off of an openstack project, and to reset all existing instances.

In addition I have many configs that I used to use, but then changed to be more flexible, more organized, or to use built in functionality rather than custom building it.

3.1 Provisioning

Currently, I have an ansible playbook to setup most of a Consul+Vault+Nomad+Waypoint system, and start a Ceph cluster, all on the provided CWDC OpenStack resources. The resource needs (9 instances) rapidly grew to be too much for my system and Vagrant to handle, so I abandoned Vagrant in favour of testing on the production infrastructure, since it is vacant of projects anyways. I have written scripts to interact with Openstack to rebuild (reset) all of the CWDCs instances, which lets me rapidly destroy and deploy, which was the objective with Vagrant.

3.2 Security

I am currently adding ACLs, with Consul done, and Vault and Nomad in progress. Ceph I understand far less, I have deployed it using an automated tool, so I don't know how I should be fine tuning the configuration for speed and security. The auth proxy is nowhere near done. Waypoint, the one service that lacks ACLs, still needs to be split into zones.

The networking side of things are still wide open. Every node in the cluster has unrestricted access to all ports on other nodes. This greatly helped with setup, but is not a viable long term solution. I would like to experiment with Consul sidecars. These would allow me to lock down access between compute nodes to basically 0, and services running on two different hosts could communicate only if they explicitly both declared sidecars, special proxies that proxy traffic over one port from one host to another, for a specific service to access.

3.3 Redundancy/Recovery

I have occasionally restarted single hosts, however I would like to simulate a larger failure, and see how the cluster responds. This could include wiping Ceph storage devices, and shutting down multiple servers/clients, or even just killing Nomad job allocations.

4 Shortcomings

4.1 Waypoint

Waypoint is a very alpha piece of software. It does not support proper ACLs, just a binary access/no access based on unique user tokens. It is unable to revoke the sessions of users, the recommended way to do so is a clean reinstall and distribute new tokens to existing users. The Hashicorp team does have ACLs in the works, though I am unsure of how long until it is GA. Waypoint is the one piece of software I will have to split up into zones, and keep the higher priority projects away from the lower priority ones.

4.2 Redundancy

Since the entire service runs virtualized on openstack, a single openstack host failure could take down a critical amount of cluster nodes. Traefik is also not redundant, with a single instance being hosted on one of the nomad clients. This makes it easier as there is no coordination required when doing http challenges that letsencrypt requires before giving a certificate. It is possible to use consul to coordinate a HA pair of traefik instances, though I am not currently planning to do this. This would also require additional coordination with Carleton ITS, and the Carleton School of Computer Science, as it would require punching additional holes in the Carleton Firewall, and adding additional DNS records to our SCS domain.

4.3 Domains

Currently the School of Computer Science has graciously assigned us the cwdc.scs.carleton.ca subdomain. This will likely not be optimal for the final setup, as it would require students to purchase their own domain (which, per Carleton ITS policy, technically needs to be approved by the school), or deal with path based routing, and the intricacies it creates. Many applications will be hardcoded, to, for example, request /style.css. This will break the site when the site is hosted at cwdc.scs.carleton.ca/myApp, and the stylesheet is located at cwdc.scs.carleton.ca/myApp/style.css. Being able to use myApp.cwdc.scs.carleton.ca would be a much neater solution, but will require additional coordination between either the School of Computer Science, or Carleton ITS, if we were to move off of the SCS subdomain.

4.4 Traefik TLS TCP Forwarding

Some of the hashicorp products, namely waypoint, use gRPC encrypted with a self signed TLS cert to communicate, with no option to disable it. Since Carleton ITS would likely need convincing to open up ports other than 80 and 443 in the firewall, we have to reverse proxy this TLS TCP gRPC traffic somehow. Since the waypoint user binary supports TLS, we can effectively use SNI to identify the host the TCP packet is heading for, and forward it to the waypoint server. Traefik, the main reverse proxy, unfortunately doesn't support re-encrypting raw tcp data going to the backend though, only ssl passthrough or termination. To have a by default (no need to install certs on the user's computer), trusted certificate presented for the gRPC traffic, the only real option is to use traefik to terminate the TLS connection using a certificate automatically obtained from letsencrypt, proxy to a service like nginx in cleartext, and have that proxy to waypoint itself, using waypoint's self signed certificate. Hopefully this is addressed soon, as termination + re-encryption already exists for https traffic, just not TCP TLS.