Clark Bains
clarkbains@carleton.ca
COMP 4000
December 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 Computer Science (SCS).
- 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 persistent 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 the one existing service to the CCSS cluster, and with Forest's help, 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 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 (VMs). I found out about Hashicorp's Vagrant, and started using that. It would allow me to write Ruby code to programmatically create VMs, from their OS, to persistent storage, resources and network using virtualization solutions I already had, like libvirt. Opting to use a tool to automate the VMs lifecycle meant I could rapidly destroy and recreate my VMs 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, I wanted much better control and coordination between nodes when configuring. For example, with clustered computing, secrets, such as encryption keys often need to be generated, then distributed to all members of the cluster. 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 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 the same environment their code gets deployed in, reducing surprises. Docker however lacked the integrations and scalability characteristics I was looking for. I ended up choosing Hashicorp's Nomad, which sits as a layer on top of Docker, similar to Kubernetes. It allows 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 environmental variables, as well as the network and storage resources it requires. It allows services to be scaled up with multiple copies of the task.

2.3 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 highly scalable Consul to do this, as it comes 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 ip(s) and port(s) that the service using, and optionally health checks. It will then keep track of what instances of the service are healthy (passing the health check), and expose them over a conventional REST API, and also a local DNS interface (so, for example, querying for the domain vault.service.consul will return all vault servers). Nomad integrates well with it, so a Nomad job could specify the Consul service name, and it will be published into Consul's catalog.

Consul also provides a key/value store, and keeps everything consistent between nodes.

2.4 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 the Consul key/value store (which is also HA). It exposes an API to get and set keys and values, though with much more Access Control List (ACL) granularity than directly through Consul K/V.

Vault also integrates with ACLs in other services, such as Consul, so that you can read a specific key, and Vault will get you a token for another service with a predefined access level associated with that key.

Finally, Vault integrates with Nomad jobs directly. Nomad is able to gather secrets from Vault and inject them directly into the environment of the Nomad job. This promotes reusability of Nomad configuration files, and also allows applications to be developed in a way that doesn't exclusively work on the CWDC cluster, as it would be easy to set environmental variables for your own testing.

2.4.1 HA Auto Unsealing

Since Vault encrypts all data, on startup it requires a threshold 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 system 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. Recovery Keys can still be obtained in case the secondary instance is unavailable.

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 would have remaining disk space, I could use files mounted as loopback devices, which makes them appear as if they are 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 can have everything listen on a predefined or random port, and I can 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 Traefik 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 for CWDC developers than trying to unencrypt 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 (waypoint), 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, /or alternatively protected in some other way.

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 responds back to dictate if the request should be sent to the backend, or if an error should be shown to the user.

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 entire reason I wanted to use Vagrant in the first place.

After an openstack issue left one Nomad client corrupted, I used my ansible scripts to completely rebuild the instance, and it took just about 3 hours for me to get everything on it working again. It was my first time trying to rebuild a single node in the cluster, rather than the whole cluster at once, so I needed to make some changes. It would be an extremely similar process to horizontally scale the cluster, and deploy more Nomad clients to handle more student services.

3.2 Hashicorp Stack

The core hashicorp stack (Consul + Vault + Nomad) is working very well. When pushing a job to Nomad, it almost immediately appears within consul. Vault secrets are easy to access through Nomad, it just requires creating a security policy for the Nomad Job, and then specifying it's name, and the path in vault to get to the secrets from within the Nomad job file.

3.3 Reverse Proxy

For HTTP traffic, the reverse proxy has been great. It is able to pick up changes from Consul with only a few seconds of lag, and it is able to provision new letsencrypt certificates within a minute or two of a service being deployed on a previously unused subdomain. Unhealthy services quickly stop getting traffic routed to them.

3.4 Security

I am currently adding ACLs. Everything but Nomad has been locked down, with most tokens for integrations with services that have enabled ACLs having the fewest possible permissions required. Ceph was deployed using an automated tool that set up the configuration and networking, so I don't know how I should be fine tuning the configuration for speed and security. As I am still the only one using the cluster, there is still only one instance of waypoint.

3.4.1 Discord Authentication and Authorization

The Discord system is functional, but requires additional protection against some attacks, like CSRF. On services with it enabled, Traefik will first wipe two headers from incoming requests, then set the same headers (representing which role ids and role names can access the service.) It then uses its inbuilt "ForwardAuth" middleware to forward the request headers, along with some extra metadata to my service. If a user has not yet authenticated using it, it redirects them to a Discord OAUTH2 sign in page, which is then used to get their discord id. The id is then persisted in the user's browser, using a signed json web token (JWT) as a cookie. On subsequent requests, the attached cookie is read, and a discord bot performs a lookup within the discord server to see what roles the user has. These are then compared to the allowed list in the forwarded headers, and the Authorization service responds back to the reverse proxy indicating to either continue with the proxy to the backend service, or to abort and show a message to the user.

This proved to be an even nicer solution than I originally expected, since it means instead of doing some form of central user authentication to manage members, everything can be done with discord roles, which are very easy to administor. Since everything is cached and prefetched where possible, the service is very quick.

3.4.2 Networking

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 remove access on almost all unknown ports, and then 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.

3.5 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.

4 Shortcomings

4.1 Nomad CSI + Ceph RBD

Currently, all persistent storage is handled by Ceph Remote Block Devices (RBD) formatted and mounted to Nomad jobs through a regular filesystem, which Nomad handles via a partially implemented Container Storage Interface (CSI) API. Being block devices, these only support one writer, which limits scalability. Unfortunately, Nomad still has some issues with CSI, that can lead it to believe a stopped job still has exclusive write permissions on an RBD, even after purging the stopped job and running a Nomad system garbage collection. This prevents new versions of the job from starting, as they cannot acquire an exclusive write lock on the volume. While the issue does seem to go away with time, it is not acceptable to have this and expect students to work around it, as resolving it requires service interruptions, administrator privileges, and can sometimes lead to the RBD getting lost, when resetting the Ceph CSI interfaces.

4.2 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 currently is a clean reinstall and distribute new tokens to existing users. The Hashicorp team does have ACLs in the works, though no set timeline until it is available.

4.3 Redundancy

Since the entire service runs virtualized on openstack, a single openstack host failure could take down a critical amount of cluster nodes. This has happened once already, where power issues within the openstack server room took down 2 instances, and corrupted the disk on another. A new generation of SCS's openstack is in the works now, as discussed in a recent SCS OpenStack meetup. This is likely to resolve the power issues, however this will not make the system entirely redundant. 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. While it is possible to use Consul to coordinate a HA pair of Traefik instances, 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 more external IP addresses, as well as punching additional holes in the Carleton Firewall for them, and adding additional DNS records to our domain.

4.4 Domains

Currently the SCS has graciously assigned us the cwdc.scs.carleton.ca subdomain. This will not be optimal for the final setup, as it would require students to purchase their own domain for their project, or deal with path based reverse proxying, 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. Currently the CWDC is in the process of working with both Carleton ITS and the SCS to secure wildcard Carleton DNS records, so that we can put projects on domains such as myApp.cwdc.carleton.ca.

4.5 Traefik TLS TCP Forwarding

Some of the hashicorp products, namely waypoint, use gRPC encrypted with a self signed TLS certificate to communicate, with no option to disable it. Like HTTP, gRPC works over TCP, however it has a different protocol, which means that Traefik is unable to parse it as HTTP and forward it to waypoint directly. To be able to handle the gRPC traffic coming through one of the only 2 (80, 443) ports open in the Carleton firewall, I had to configure Traefik to forward as TCP, which foregoes all parsing abilities, as tcp data itself has no set format. To uniquely identify the waypoint gRPC TCP packets, I had to use Server Name Indication (SNI) information that is usually present when communicating over encrypted TLS.

SNI is similar to the ”Host” header in HTTP. It is attached as extra metadata and indicates the hostname of the server that the the packet was destined for. The waypoint user binary supports this.

Unfortunately, raw TLS TCP forwarding in Traefik cannot be rencrypted after being decrypted by Traefik, so it is unable to encrypt with a valid letsencypt certificate when going to the user, and a self signed certificate when going to the waypoint gRPC backend. The only alternative is to use TLS passthrough, where the user is simply presented with the self signed certificate from the waypoint gRPC backend. This forces users to disable TLS certificate verification within waypoint, and makes them susceptible to Man-in-the-middle attacks.

5 Going Forward

5.1 CephFS

To get around the problem of Ceph RBD write allocations preventing new deployments, I am looking into switching from Ceph RBD to CephFS. When mounted directly as a filesystem, multiple writers are permitted, so even if dead jobs fail to release their write claims, new jobs would still be able to get a new write claim. This is not an end solution, but could be good enough until the CSI interface is completed in Nomad.

5.2 User Driven Deployment

Ultimately, even just giving approved users access to deploy arbitrary web services is not the most secure idea. While the original goal of this stack was to provide a fully automated way for deploying, I no longer believe that this should be fully controlled by students with little previous knowledge of the tools. Waypoint is not ready to provide the security required for this. Ultimately, it may make sense to offload waypoint and Nomad job file creation and maintenance to the cluster administrators. From there, we could setup projects on a private version of waypoint, and retain control over the deployment specifications, while letting waypoint take care of deploying the code itself according to our specification.

5.3 Assisting Students + Automation

Despite being made for CWDC members, there are still no student projects running on it. Once I am satisfied with its operation, I plan to prepare documentation and tools to help students build their tools to best leverage features of the stack with their project, for example, Vault integration via environmental variables. Afterwards, I should look into promoting the technology within our group, possibly by doing a presentation, and by working closely with ongoing projects.

5.4 Sharing My Work

I've done a lot of work for this cluster, including my many lines of ansible configuration, my discord authentication project, and this paper. I also have had to fork Traefik to get the Consul Key/Value configuration source to work with my non-enterprise version of Consul, and will need to update its documentation to reflect the changes I made. Given that this whole project was made possible because of many different open source communities, I feel as though publishing what I have, and sharing the knowledge I've gained is the best way to give back.

My repositories are not currently available, as they contain some secrets used within the cluster. Source files can be obtained by emailing me.