Building a Flexible and Resource-Light Monitoring Platform for a WLCG-Tier2

Robert Currie[1,*] and *Wenlong* Yuan[1,**]

[1]School of Physics and Astronomy, The University of Edinburgh, James Clerk Maxwell Building, Peter Guthrie Tait Road, Edinburgh, EH9 3FD

Abstract. Software development projects at Edinburgh identified a desire to build and manage our own monitoring platform. This better allows us to support the developing and varied physics and computing interests of our Experimental Particle Physics group. This production platform enables oversight of international experimental data management, local software development projects and active monitoring of lab facilities within our research group.

Larger sites such as CERN have access to many resources to support general-purpose centralised monitoring solutions such as MONIT. At a WLCG Tier2 we only have access to a fraction of these resources and manpower. Recycling nodes from grid storage and borrowed capacity from our Tier2 Hypervisors has enabled us to build a reliable monitoring infrastructure. This also contributes back to our Tier2 management improving our operational and security monitoring.

Shared experiences from larger sites gave us a head-start in building our own service monitoring (FluentD) and multi-protocol (AMQP/STOMP/UDP datagram) messaging frameworks atop both our Elasticsearch and OpenSearch clusters. This has been built with minimal hardware and software complexity, maximising maintainability, and reducing manpower costs. A secondary design goal has also been the ability to migrate and upgrade individual components with minimal service interruption. To achieve this, we made heavy use of different layers of containerisation (Podman/Docker), virtualization and NGINX web proxies.

This presentation details our experiences in developing this platform from scratch with a focus on minimal resource use. This includes lessons learnt in deploying and comparing both an Elasticsearch and OpenSearch clusters, as well as designing various levels of automation and resiliency for our monitoring framework. This has culminated in us effectively indexing, parsing and storing >200GB of logging and monitoring data per day.

1 Introduction

Development activities at the Edinburgh WLCG Tier2 identified to us the need to deploy a flexible and resource/manpower-light monitoring platform.

*e-mail: rob.currie@ed.ac.uk

**e-mail: wenlong.yuan@ed.ac.uk

Originally we had setup a simplified monitoring infrastructure for work presented at vCHEP 2021 [1]. This involved deployment of a web-based monitoring infrastructure platform, making extensive use of ElasticSearch [2] and multiple custom services. Initially our monitoring system was entirely supported manually as normal deployments of services atop CentOS7 based servers, however it became clear that this scaled poorly with extremely limited manpower. With that in mind, alternative solutions for service/server management were explored.

WLCG dashboards presented at a previous CHEP [3], built atop the CERN Monit infrastructure provided us with a good inspiration of what we would like to achieve. Our goal has been to expand this monitoring service to support multiple experimental goals within Edinburgh in addition to supporting the requirements of international experiments.

After some initial testing it was decided we want to build a containerized infrastructure (using Podman [4] or Docker [5]) backed by either OpenSearch [6] or ElasticSearch [2]. Unfortunately more detailed investigations failed to reveal a which product was superior in either case. With this in mind we decided to test and compare the deployments of ElasticSearch on Docker vs OpenSearch on Podman. Our hope being to determine which solution is best for our use case after multiple months of administration experience.

2 Monitoring Requirements

Our monitoring infrastructure has been designed and optimized to be run/maintained by minimal manual effort and administrator input. The intention is that this can be run as an automatic service without the need for constant manual intervention. In addition to this we wanted to meet the following requirements:

- **Broad Accessibility**
  The main intention of this monitoring infrastructure is to produce a set of publically viewable monitoring dashboards at the same time being a service that digests data which we regard as private. This means that we want to build a system with dynamic public dashboards in addition to supporting private dashboards for internal use.
- **Service Stability/Reproducibility**
  When trying to design a service to be as stable and reproducible as possible we decided to make use of containerized software solution. This provided us with the advantage that when correctly used containers allow us to keep track of fixed versions of software and configurations. One of the aims of our configurations is to be able to treat most of the components as effectively stateless, meaning that should a component fail we can simply replace it with a new instance.
- **Monitoring of Services**
  One of the requirements we had was to be able to track the state and logs of all of the services centrally. In order to do this we made use of tools under the prometheus [7] project as well as log tracking using fluentd [8].
- **Simplified Maintainability**
  One of the aims of this was to reduce the effort needed for either training a new administrator or for an existing administrator to maintain the system. This would be achieved by avoiding too many custom components and non-standard configurations.
- **Reduced Documentation**
  One of the goals of simplifying our mainatainability is to remove the need for lots of documentation. By collecting our configuration files into a few common locations, and making use of our university GitLab we aim to make our configurations largely self-documenting.

2.1 Limitations

Due to a limited amount of IPv4 space available within the GridPP VLANs at Edinburgh, we are required to keep to a minimum number of unique IP addressed. In order to do this, we decided to opt for a configuration with multiple hostnames pointing to a single IPv4 address. This is the opposite approach normally taken when designing HPC systems at scale where a single DNS record usually corresponds to a single physical host. To be able to configure this we are making use of a delegated DNS service which allowed us to use DNS record management under the domain edi.scotgrid.ac.uk.

Automating certificate management allows us to avoid the need for manual interventions to update services, reducing effort and the possibility of mistakes. As we are looking to make available public dashboards that are viewable by international collaborators, we decided to make use of SSL certificates issued by a commonly trusted authority. As the scotgrid.ac.uk domain is shared with multiple research groups we found that attempting to use the letsencrypt [9] often resulted in rate limiting issues. To avoid this we opted to make use of certificates issued using the acme protocol by ssl.com [10]. To be able to generate and deploy these certificates automatically we made use of containers from the nginx-proxy [11] [12] project. As the ssl.com authority is not supported by default in the nginx-proxy project, we had to make minor modifications to the container to allow us to use this certificate authority.

Additionally, this project is limited by the amount of hardware available to us. Services for this project have to be able to be deployed within overhead capacity on our Tier2 hypervisors and on recycled hardware from decommissioned grid storage nodes.

3 Containerized deployments

In order to efficiently deploy and configure multi-container based solutions we decided to make extensive use of docker-compose [13] and podman-compose [14]. Both of these projects have implemented the Compose Specification [15], meaning that the resulting configurations can be easily ported from one solution to the other should we reach an impasse with one project not meeting our requirements.

Figure 1 shows the various components used in constructing our web proxy system using NGINX [16] containers. Using NGINX in this way allowed for us to have fine-grained control over the routing of incoming connections to the various services we are running.

When designing this system we wanted to support the widest number of potential use-cases with the smallest number of components. To achieve this we opted to use the RabbitMQ messaging system which supports a wide range of incoming and outgoing protocols. In order to reduce the possibility of data loss we decided to use quorum queues within RabbitMQ to increase message durability. Figure 2 shows the various components of the RabbitMQ messaging system that we have deployed. Making use of an NGINX proxy allowed us to build a round-robin load balancer in-front of our RabbitMQ cluster. This allows us to scale the number of RabbitMQ nodes up and down as required without the need to reconfigure clients.

After data has been ingested into our monitoring infrstructure it is then indexed and stored in either our OpenSearch or ElasticSearch cluster. Figure 3 shows the various components of the OpenSearch cluster that we have deployed. An identical configuration was also used to deploy our ElasticSearch cluster. Deploying our indexing and storage systems in this way means that we have been able to perform updates in production without service outages. By making use of container limits we have also been able to deploy these services on low-end hardware without significant needs for intervention.

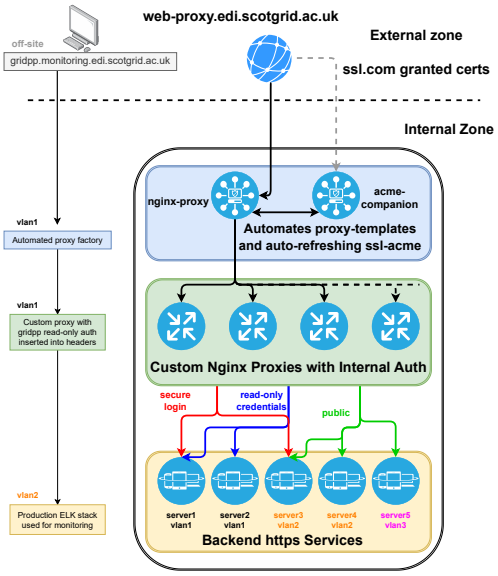

Figure 1. This diagram shows the various components of the Web Proxy system that has been setup to accept and delegate incoming connections to various web services based on the requested hostname. Different inbound requests are routed to different services based on the requested hostname. This allows for custom headers to be injected into HTTP requests to grant appropriate access to different backend services.

4 Discussion

During the development and deployment of our monitoring system we gained several insights which were not readily apparent at the start of this project.

- **Only mount folders into containers**:
  After working with containers for multiple months we ended up adopting a best practice of only mounting folders from the system into containers. Although individual file mounting is supported, we found this often led to more confusion and inflexibility due to production configuration changes requiring to be made via in-place file modifications. As previously stated working with SELinux was also found to be easier when using folder mounts as contexts are applied recursively when new files are added/created.
- **One container per DNS record**:
  In order to maximize both stability, configurability as well as clarity, we opted for a system where we manage one DNS record per container. One of the main advantages of limiting 1 container per DNS record is the fact that our configuration systems end up more self-documenting as it's clearer what component of what service is running on each host.
- **Security with Containers requires care**:
  Both Docker and Podman work extremely well in hardened environments with SELinux enabled and with strict firewall rules. Originally we managed SELinux contexts via the Compose Specification for deployment and administration. This gave us a simple way to allow for the correct SELinux contexts to be applied to mounted folders. However, we found that this scales poorly with many files on disk due to the implementation of the specification requiring all files be re-labelled when launching a service. As a result we

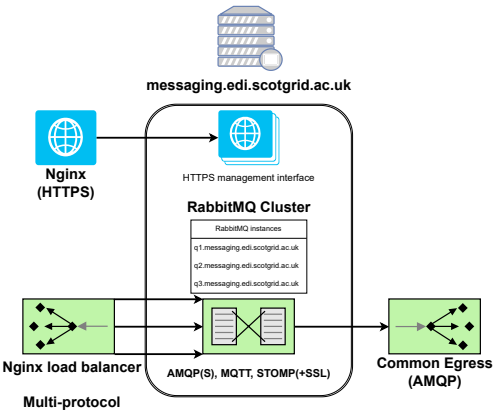

Figure 2. The different components of the RabbitMQ messaging system that we have deployed. The NGINX proxy allows us to build a round-robin load balancer in-front of our RabbitMQ cluster. This nginx instance redirects incoming connections directly to the various RabbitMQ nodes functioning as simple load balancer.

opted to use this for initial service deployment(s) and then to rely on SELinux contexts being recursively applied to mounted folders.

In addition to SELinux we found that container management was simplified by configuring iptables directly. As container support for nftables by most containerization solutions we decided to opt to a more traditional iptables based approach.

- **Prefer NGINX frontends**:
  Making use of NGINX for load-balancing and proxying incoming connections has been extremely useful. This allows us to maintain service availablity during essential upgrades and migrations. We also found using NGINX in this way has provided a way of managing both complex authentication as well as managing SSL encryption at our network perimeter.

After administering various services with comparable underlying technologies, we are also in a position to be able to evaluate the performance of different technologies in terms of service stability, maintainability and reproducibility.

- **Docker vs Podman**:
  After many months of working with both Docker and Podman on CentOS7 and Alma Linux 9 hosts, we have found no significant benefit to either technology in terms of service reliability or reproducibility. Early experiment with Podman identified areas such as network configurability, which differ between both Podman and Docker which requires special attention. When running either solution on Alma Linux 9 hosts we found these differences were further reduced when running container orchestration systems on a more modern OS.

  With this in mind we envision future deployments being constructed using the Podman service as it is the default containerization solution on Alma Linux 9. It also has the advantage of not requiring root level daemons when running containers.

- **ElasticSearch vs OpenSearch**:
  Using both ElasticSearch and OpenSearch products for several months as the main data storage backend for our monitoring infrastructure, we have found that both products perform well at ingesting, indexing and searching data at scale. Intially OpenSearch in earlier

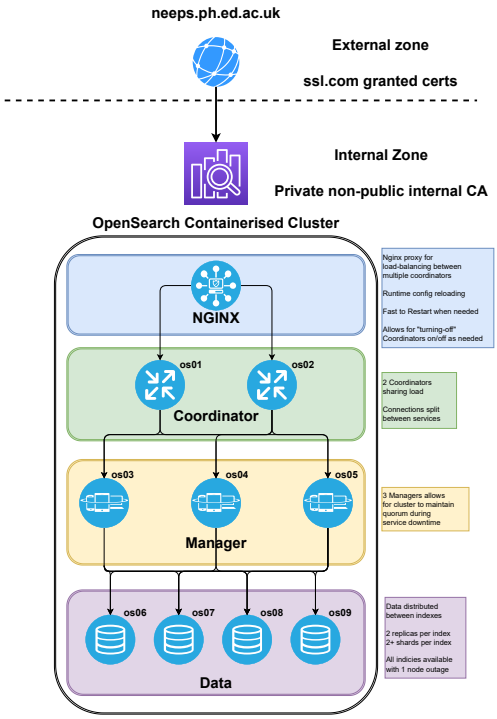

Figure 3. The configuration of the components within our OpenSearch cluster. The NGINX proxy allows us to build a round-robin load balancer in-front of our ingestion nodes. Designing the OpenSearch cluster this way we have found we're able to perform in-place service upgrades without service interruption.

versions proved slightly less stable, however the project has quickly improved in terms of stability and usability. Although we are yet to make use of much more advanced features in production such as machine learning, we have found that both products have been well suited to the needs of ingesting service logs and providing a searchable interface for our monitoring dashboards.

5 Conclusion

We have been able to build and deploy a containerized monitoring infrastructure which has been able to adapt to meet the requirements from within our research group as well as to work with international collaborations. Administration of this has been achieved with minimal manpower and hardware resources. Service level interventions have been avoided by making heavy use of automation. This has allowed us to focus on more important issues such as the physical relocation of some services between datacenters within the university.

We have also found that by making use of NGINX proxies and redundant services has been an advantageous approach to service management. When performing service updates/upgrades we have found that we're able to catch production problems directly when updating individual components. Containerization and backing up configurations in Git has also allowed us to asses the impact of changes before they are deployed to production, whilst

giving us the opportunity to roll-back and delay an update until the impact is better understood.

After approximately 12 months of hosting the various services associated with this there has only been one unforseen major impact on service availability. This was running our OpenSearch cluster on a filesystem backed by aging/failing storage drives. We found that whilst OpenSearch has built-in sharding and replication to avoid data loss, performance and stability is significantly impacted by drives entering into a pre-failure state with high I/O latency. After identifying this as the cause of issues within the OpenSearch cluster we were able to remove these drives from production allowing the system to recover. Additional monitoring of hardware performance with node_exporter [17] was found to be beneficial in both identifying this issue and in monitoring the health of our ageing hardware.

References

[1] Currie, Robert, Yuan, Wenlong, EPJ Web Conf. **251**, 02052 (2021)

[2] *Elasticsearch*, https://www.elastic.co/elasticsearch

[3] A. Aimar, A.A. Corman, B.G. Bear, D.L. Nicolau, G.M. Borge, L. Magnoni, N. Tsvetkov, P. Andrade, S. Brundu, *Wlcg dashboards with unified monitoring* (2019), https://doi.org/10.5281/zenodo.3598977

[4] *Podman*, https://podman.io/

[5] *Docker*, https://www.docker.com/

[6] *Opensearch*, https://opensearch.org/

[7] *Prometheus*, https://github.com/prometheus/prometheus

[8] *Fluentd*, https://www.fluentd.org/

[9] *Let's encrypt - free ssl/tls certificates*, https://letsencrypt.org/

[10] ssl.com, *Ssl certificate & digital certificate authority - ssl.com*, https://www.ssl.com/

[11] N. Duchon, *acme-companion*, https://github.com/nginx-proxy/acme-companion

[12] N. Duchon, *nginx-proxy*, https://github.com/nginx-proxy/nginx-proxy

[13] *docker-compose*, https://github.com/docker/compose

[14] *podman-compose*, https://github.com/containers/podman-compose

[15] *Compose specification*, https://github.com/compose-spec/compose-spec

[16] *Nginx*, https://www.nginx.com/

[17] *node_exporter*, https://github.com/prometheus/node_exporter