A Multitenant Container Platform with OKD, Harbor Registry and ELK

Jarle Bjørgeengen(✉)

University of Oslo, USIT, Gaustadalléen 23a, 0373 Oslo, Norway
postmottak@usit.uio.no
https://www.usit.uio.no/english/

Abstract. This paper summarizes the open container [2] journey of the University of Oslo's Center of Information technology, (Division for Infrastructure). It describes the background for adopting containers in the first place, the pitfalls of early attempts, the learning that was obtained from stepping into those pitfalls, how they were mended and some thoughts about future direction for container usage. Challenges regarding organizational aspects and increased demand for rapid delivery, combined with the established expectations of security and stability, is also described in relation to container technology. It distills the findings and explains the rationale behind the chosen direction of adapting Openshift community Distribution of Kubernetes (OKD) [1] as our main container platform for long running core services, and how it was adapted to best integrate it with existing automation, monitoring and logging: Elasticsearch, Logstash and Kibana (ELK).

Keywords: Open containers · Kubernetes · Docker · Continuous delivery · Multi tenancy · Self service

1 Introduction

There is an increasing demand for and utilization of open containers [2] by IT-professionals today. The majority of cloud native technologies utilize container technology in some form. Containers make promises of advantages like: faster software delivery, immutable services, portability and dynamic on-demand scaling. Software developers and users of containers and container platforms are compelled by containers' ease of creating and continuously improving powerful functions that provide businesses value in highly competitive markets that change ever more rapidly. But containers also introduce new challenges, especially regarding increased complexity and/or security. As the number of software abstraction layers and interfaces from users down to physical infrastructure increase, the complexity increases too. The exposure for attacks and cascading

Supported by University of Oslo, Center for Information Technology (USIT).

M. Weiland et al. (Eds.): ISC 2019 Workshops, LNCS 11887, pp. 69–79, 2019.
https://doi.org/10.1007/978-3-030-34356-9_7

failures is increasing accordingly. How to balance the benefits and challenges of containers ? With this context the paper describes the use cases, history, considerations, current state and future vision for container usage in delivery of core services at USIT.

2 Past

2.1 Background

We started using Docker [3] containers for some small web based applications in 2015. Applications like Graphite [4], Grafana [5] and Kibana [6] with our home made role based access control (RBAC) system which is used for filtering Elasticsearch [7] documents based on ownership. We found that containers provided value when making multiple instances of the same application, but with slightly different parameters. Orchestration of containers was mainly done with Ansible [8] and templating was done with Jinja2 templates within Ansible. Containers made it possible to run many applications instances on a few hosts that would otherwise require one host per application.

However we also found that the immutable nature of containers introduced some headache when it came to keeping application stacks up to date with security fixes. Long running service containers tended to stay up for long periods of time and their software stack (being immutable) were unchanged. This is of course wanted behaviour when it comes to predictability of application dependencies, and it is much of the value proposition of containers. But keeping those software stacks unchanged over time led to exposure against vulnerabilities that otherwise would have been automatically updated if they just run as ordinary processes in our standardized Linux server environment. This pattern introduced a business risk that needed to be mitigated, so we made a container policy in order to regulate how containers were to be used and managed for production services running on core server networks. The policy mainly described requirements for:

- Image sources: A local registry with a vetted list of replicated images from external sources.
- Detection and mitigation of vulnerabilities: Automated scanning and actions to take.
- Run time requirements: Don't run as root, allowed version of container run times etc.
- Transparency: All artifact inclusion in version control system (VCS). Tagging images with VCS version. Always use image from local registry (build and push before use)

The main purpose of having an on premise registry is to be able to decouple the dependency from external registries both for confidentiality and availability. For instance Dockerhub has been regarded as safe location for software distribution, but recent security breach [9] shows that attacks can happen and that exposure and tampering are real threats.

At USIT we have standardized on RedHat Enterprise Linux (RHEL) for Linux servers running core datacenter services. Servers are automatically bootstrapped with configuration management agents (Cfengine3 [10]) that converge and maintain the state of server operating systems into a safe baseline. One of the aspects covered by config management is to enforce that SELinux is turned on in version 7 of RHEL. Many people wrongly assume that containers provide stronger isolation than they in fact do [11]. In RHEL7 the Docker engine is integrated with SELinux so that containers automatically runs in unique SELinux contexts, and with reduced privilige to access host resources. It means that for RHEL7 with SELinux on containers provide a higher level of isolation between them and against the hosting node, than a vanilla Docker Community Edition installation. This is why our policy require production containers to run on RHEL7 (CentOS7 in IaaS-cloud environments) with SELinux on. SELinux is by no means the only runtime security measure to take regarding securing containers and it definitely does not exclude other best practices to secure containers at runtime, for example seccomp [12] for systemcall filtering, EBPF [14] and user namespace remapping [13], however SELinux comes as an additional security layer on RHEL7 and does not require any extra effort to set up.

For on premise registry we required one with open source license, RBAC and support for external authentication/identity-providers in order to re-use existing business logic and organizational structure. Harbor registry [15] were as close as we got to those requirements at the time. Initially Harbor supported LDAP-authentication but not authorization based on group memberships so we made scripts for synchronizing local Harbor groups with ldap-groups. Also Harbor had no built in vulnerability-scanner, so we used a separate instance of CLAIR [16].

2.2 Challenges

The main problem with vulnerability scanning of container images is getting relevant data. One could scan all contents of the registry, but only a subset of the images in the registry is actually in use. Thus, scanning the whole registry would generate alerts for a lot of irrelevant images. Since we have both configuration management (Cfengine3 [10], and centralized logging (Rsyslog, Logstash and Elasticsearch)) we can collect information about which container images actually is in use on server hosts in our networks. This was done by using Cfengine to inject hourly updates of meta-data to Rsyslog about images used by running containers on hosts that had Docker installed and running. The metadata sent to Rsyslog was parsed and sent to Elasticsearch centrally and thus instantly made available for querying via the Elasticsearch API. Hence, the baseline for what container layers will be scanned and alerted upon is the set of image-ids found in the container meta-data store in Elasticsearch.

Meta-data about ownership of images is also collected and it provides email-addresses for alerting container owners about vulnerabilities in their running containers. If images lacks contact information the owner of the host is alerted via the owner's contact email address, which is collected and provided by another tool created at USIT: Nivlheim [17]. If containers run from images that is not

found in the Harbor registry the container owner is notified about this breach of container policy as well.

Even with this pre-seeding of data before vulnerability alerting, the owners got too many alert to handle in an efficient manner. The reasons for this were twofold: there was no option for image owners to flag vulnerabilities for some subset of containers as "acknowledged" and filter them out upon subsequent scanning.

Also the hypothesis were made that many of the alerts may or may not be relevant because of how images are built. Images are frequently built using ordinary package install commands (yum, apt, port, pkg and so on) in the Docker file to add image layers. These tools tend to pull in orders of magnitude more dependencies than the container process needs, and many of the dependencies may or may not be posing a threat to the running process albeit present in the image, and thus can trigger hits by the vulnerability scanner. How to differentiate relevant from irrelevant? Nontrivial. One way of improving this is to put more labour into slimming down the images during builds (this is also mentioned as a desired property of image builds in the container policy). The advantage is that the images produced actually have a smaller attack surface and it will naturally have less likelihood of vulnerability alerts too, and the alerts is assumed to be more relevant. Unfortunately the ability and willingness to make image slimming happen are not very prominent, due to time constraints, labour requirements due to lack of tools for automating the process.

The limitations of Ansible and Jinja2 as a container management and orchestration tool became apparent. There is no rescheduling upon container failure, no service discovery and/or scale out features. The lack of observability is prominent and self service features need to be built using tools like Rundeck [18] and/or Ansible (Tower) and this leads to more infrastructure/platform code to develop, maintain and operate.

Images are replicated from external container registries (dockerhub etc) and into the `/library` project of our on premise instance of Harbor. Replication is done by script on an hourly basis. The script reads a list of `registry/image:tag` instances, pulls them to local container storage, re-tag them with the URL of the Harbor registry and pushes them there.

The procedure for adding new image references to be synchronized involves filling out a form with some standard questions about justification and considerations regarding trust. Submission of the form triggers en informational email to it-security and a new ticket to request tracker. If the request looks reasonable I (or someone like me) add the new image reference to the replication list, and it will be included in the hourly synchronization.

3 Present

Armed with experiences with past usage of containers, and conscious of an increasing demand for application containerization we set out to improve the state of containers at USIT. Some parts were working well; the image replication process and the on premise registry storage, so they were kept albeit upgraded,

but there were ample room for improvement regarding orchestration, observabilty, self service, support for continuous delivery and deployment automation.

3.1 Evaluation of Container Orchestration Frameworks

During 2018 we ramped up the effort for finding improved alternatives for container orchestration. By this time the whole industry seemed to converge against Kubernetes [19] based solutions in one form or another. Kubernetes being backed by large open source founded companies like Google and RedHat, being the core project of which the Cloud Native Computing Foundation (CNCF) [20] was ignited from [21] and major cloud service providers like Google, Amazon and Azure were offering Kubernetes as a service targeted to increase developer efficiency. Hence we focused on use case experiments with frameworks that were utilizing Kubernetes.

Kubernetes is well defined when it comes to API-functionality for the different versions and this is also the value proposition for developers and/or Kubernetes users. It says: "give me a state declaration for scaling, connectivity and resources of your micro services and I will make it so". However, assembling, improving and maintaining a Kubernetes service that will keep that promise reliably and securely over time is a nontrivial task.

From a platform perspective Kubernetes look more like a set of software components that can be assembled into a service in many different ways. Core Kubernetes developers have been comparing it with the GNU/Linux stack and how that can be assembled into more or less opinionated Linux distributions [22]. Taking upstream Kubernetes and assembling it into a stable production service is hard [23].

For this reason we set out looking for "distributions" of Kubernetes that when installed were assembled automatically in a way that aligned as much as possible with our existing operational practices, policies and strategies: secure, standardized, multitenant with RBAC and integration with identity providers, high degree of developer self service, high degree of deployment and build automation and with an open source licens.

We tried a few Kubernetes installation frameworks: Kubespray [24] and NAISible [25] with some local modifications, but found that some core requirements regarding RBAC, multitenancy and developer self service were lacking. Then we tried OKD. OKD claimed to fulfill all the requirements, however it is also a rather large and complex installation process (although automated) and it would inevitably take time to understand all its moving parts.

Experiments with earlier versions (Openshift Origin) a couple of years back exhibited lack of modularity and poor error messages in the installation process combined with inaccurate and incomplete documentation, thus setting the expectations towards a daunting task. However, after installing v3.10 of OKD it became apparent that much of those earlier problems with installation and documentation were improved. Also there generally were a lot fewer bugs in the installation framework than earlier, particularly for the core functionality (Kubernetes).

OKD is multitenant via the consept of projects. Projects are much the same as Kubernetes name spaces, but with isolation features between them. Kubernetes namespaces is just that: separation of names (unless additional steps are taken). OKD projects have by default isolation on network (OVS multitenant) and process (SELinux) level between projects. There are also default enforcing policies to prevent breakout from containers/pods towards the hosting node (SELinux, SecComp, Security Context constraints (SCC)).

In addition to the core Kubernetes API compability, OKD has self service features for automating deployments and builds via image streams and build configs. Build configs can incorporate Jenkinsfiles and run them on demand by spinning up Jenkins instances on the fly or by having Jenkins running in projects (for shortening pipeline traverse times). Another nice feature of image streams is that running applications can be configured to automatically redeploy with updated base images via event triggers in image streams.

Efforts were increased to verify if OKD was a suitable to become our new container runtime platform. The next step was to adapt the installation to our existing environment and hypothesis testing against application development use cases. This effort was mainly organized as weekly "sprints" with members from departments from opposite sides (dev and ops) of the line organization and initiated only by peers communicating and silent approval by relevant managers. Unsurprisingly this method turned out to be efficient for progress and organizational learning, and we have increased the frequency of sprint weeks in order to reach production ready state by the summer of 2019.

3.2 Observability: Logging and OKD

OKD comes with an optional Elasticsearch, Fluentd and Kibana (EFK) [26] stack which is tightly aligned with the multitenancy in OKD. Project owners automatically get access to logs of their own pods, but not to other projects. However, Elasticsearch is by nature a difficult application to containerize and maintain in a stable manner. Also it has high demand on CPU, memory and storage IOPS resources on each deployed cluster.

Since we already have a rather large ELK installation with Elasticsearch running on 21 bare metal servers, it seemed like a more attractive option to scratch the built in EFK stack of OKD and integrate with USITs ELK instance. Another benefit with this is of course the ability to correlate and aggregate logs between OKD cluster and other log-sources. But how to automate this as a part of the installation, and how to create self service for tenants? Our ELK instance is already multitenant but how to make the connection between OKD tenants and ELK tenants?

The standard configuration management baseline for all our servers include a filebeat agent shipping system logs off to Logstash centrally. It turns out that filebeat in recent versions introduced integration with Kubernetes [27], making it possible to discover pods/containers-ids from the Kubernetes/OKD API and automatically ship log events based on matching conditions like namespace name, labels, annotations etc.

We decided to have logs from cluster components tagged and sent to a separate index prefix in ELK (kube-ops). This is done by matching pod events against list of projects/namespaces known to host system pods.

So now we have cluster system logs sent to ELK and accessible by cluster operators (admins with high level of privilege), but what about application logs? Ideally we would like to have tenants decide by themselves which deployments to enable logging for, to mark them as their own and sub-classify using an application field. It turns out that this is possible by using the same filebeat autodiscovery feature for Kubernetes by matching on labels. We introduced a label `log2elk: true` for tenants to signal that they want to ship logs to ELK.

In ELK it is good practice to log events as JSON-data in order to get maximum value of aggregation and grouping features later. So we decided that applications that apply the "log2elk: true" must log output as JSON. Furthermore the JSON must have a "logowner" field matching a known logowner in ELK (logownertenant), and an "application" field that makes grouping of applications within a logowner possible. So filebeat will send log events from pods with the label `log2elk: true` to Logstash centrally, Logstash input processors will check required fields and if they comply restructure events such that they get ingested and become available through Elasticsearch filter aliases for logowners to access from their dedicated Kibana instance. In USIT-ELK each logowner has their own Kibana instance which is tied to a user group in LDAP and is restricted to that logowners data set.

OKD has an advanced audit logging system [28] that can be configured by a separate audit policy file on the master node(s). It can potentially generate a lot of log-data and needs some iterations of policy modifications to balance value with spuriousness. It can be configured to log JSON, so it is straight forward to drop a filebeat config fragment into the directory `/etc/filebeat/filbeat.d/` on master nodes. Filebeat will then tag and ship the audit log to ELK.

Just like in Kubernetes cluster events are temporarily stored in etcd, and we need a way of exporting them as JSON events. Kubernetes has an events endpoint that can be watched/followed and where events will be published as they happen. We will write a small script or program that captures events from the OKD/Kubernetes API and spool them to file, and drop another filebeat configuration fragment into `/etc/filebeat/filbeat.d/` for tailing and shipping events.

3.3 Observability: Monitoring and OKD

We use Zabbix [29] as the central monitoring framework. Configuration of monitoring templates and access to modify configuration (self service) is completely automated through scripts that ask Nivlheim and LDAP to automatically decide what to monitor and whom to give access at which level. A subset of the monitoring automation is to automatically configure health monitoring for java based web-application. This is also done by asking Nivlheim about what to monitor and how. Nivlheim knows this by means of collected files from hosts running java-applications.

These java-applications are the first candidates for moving into OKD, but when containerized the applications can no longer rely on file access to the host and thus Nivlheim has no knowledge about them (being a file based collection tool for hosts) and installing a Nivlheim agent (or any management agent) inside each container is considered an anti-pattern. Finding an alternative method for monitoring applications running in OKD was the topic of the first sprint week, and a minimum viable product (MVP) was created during that week by application developers and operations cooperating. Two main changes were made: Applications made health information available through a http endpoint in its own service and the route/ingress of the application in OKD were marked with annotations (`uio.no/monitor.with.Zabbix:"true"`) which were picked up by a Zabbix auto discovery script that regularly asks OKD clusters about their routes/ingresses and pick out url-endpoints (stored in another annotation) to be configured for monitoring.

4 Future

4.1 Monitoring

Cluster events in ELK can form the basis for alerting rules by querying ELK for events of a certain severity and with other properties that we find we need to know about. Also v3.11 of OKD comes with an integrated Prometheus operator [30] that can monitor cluster health and resource consumption (both cluster wide and applications). The built in Grafana instance provide tenants the ability to monitor metrics of their own liking and resource consumption of their own applications. Also we already have an LDAP-integrated instance of Grafana running centrally with access to all other infrastructure and application metrics. Adding prometheus instances of OKD-clusters as Grafana datasources is trivial and make it possible to cross correlate and combine OKD metrics with other infrastructure metrics. Since v 4.2 of Zabbix there is a built in integration with Prometheus [31] that makes it easier to monitor cluster health directly with Zabbix.

4.2 Container Policy and OKD

Our container policy requires that all container images are being pulled from a local Harbor registry. The installation playbooks of OKD fetches most of the cluster components from images located in dockerhub by default. Fortunately this can be changed by a parameter named “oreg_url” that specify an alternative registry url-prefix to fetch cluster components from.

In order to have a baseline for all running containers that is not too much out of alignment with our general Linux server update policy it was recently decided to require all container images to be rebuilt and redeployed at least every 30 days. Automating container rebuild and push to registry is quite trivial, but automatically redeploying with rebuilt image and verifying that the service works as expected afterwards is not that trivial. Our hypothesis is that OKD can help with this utilizing image streams combined with deployment configs that

trigger automatic rolling redeployment upon new images being pushed into the image stream. Also if the deployment is properly configured with Kubernetes liveness and readiness probes it will decrease or eliminate downtime for updates and make sure that new pods are healthy before service pointers are switched to updated pods.

4.3 Gitops [32] and OKD

We keep all our infrastructure and application code in VCS (git), and we strive for as little human intervention as possible when installing, running, modifying and maintaining infrastructure and platforms. At the moment we have no automatic triggering of configuration changes in OKD clusters after installation. Ideally it would be possible to make a change in the inventory of a running cluster, and it would automatically converge to new desired state when a pull request is merged.

Although Ansible itself have some level of convergence and idempotence, there is too much dependence on order of events in OKD installation playbooks and roles to make such behavior reliable. However, the new major release of OKD (v4) seems promising. It is much more based on self management by means of cluster operators, and will rely less on Ansible. The introduction of operators for managing all cluster life cycle aspects could make OKD more gitops friendly. We plan to try OKD v4 in the future, but meanwhile we will make project configuration (ownerhip, resource quotas, privileges and metadata injection) in OKD completely automated via git, pull requests and application of new state via the `oc apply` command.

4.4 Continuous Delivery in OKD

OKD has features that makes it possible to host complete continuous delivery pipelines inside projects. Today most of our developer groups uses build automation and automatic testing with Jenkins or other tools. There is some operational overhead of running standalone Jenkins instances. Based on our knowledge of OKD features we have a hypothesis that developer efficiency can be boosted and operational overhead reduced by moving the pipelines into OKD.

4.5 OKD in the Cloud

Portability is major benefit of running applications in containers. There pressure for utilizing public cloud services is increasing. The main reasons are bursting resource availability, quick cost scaling and reduction of operational expenses and investments. Just like many other organizations we foresee that public cloud consumption will increase, but it will take time to migrate to a “cloud only” situation, and based on regulation and/or policy some data, and hence the services processing those data, may never be allowed to run in public cloud. For applications with extreme resource consumption over long time it is probably not cost efficient (yet?) to run everything in public cloud. The bottom line is that we will

need both on premise and public cloud in the foreseeable future. The ability to quickly move applications between on premise and public cloud in an automated or even autonomous way is often termed "Hybrid cloud", and we think that container platforms like OKD will play an important role in achieving hybrid cloud functionality.

At the moment we are working on automated provision and install of OKD in our community Openstack (RDO) [33] cloud UHIaaS [34] with terraform [35] and OKDs Ansible installer (plus som adaptations). Furthermore the plan is to extend this to work on major public cloud platforms from Google, Microsoft and Amazon, thus enabling maximum applications portability by offering the same run time platform (OKD) both on premise and in different public cloud contexts.

5 Conclusion

So far we have enough evidence that OKD and our approach to integrate with our existing operational systems, and spin up new clusters in different contexts, that we will continue the work along the lines suggested in Sect. 4. The aim has been and continue to be to help application developers and/or container users become more agile and efficient in delivering business value with balanced security, scalability and availability in mind. More automation, probably in the form of GitOps, will help us to improve further.

References

1. Openshift Community Distribution of Kubernetes. https://www.okd.io/
2. Open Container Initiative. https://www.opencontainers.org/
3. Docker Wikipedia Page. https://en.wikipedia.org/wiki/Docker_(software)
4. Graphite Home Page. https://graphite.readthedocs.io/en/latest/
5. Grafana Home Page. https://grafana.com/grafana
6. Kibana Homepage. https://www.elastic.co/products/kibana/
7. Elasticsearch Homepage. https://www.elastic.co/products/elasticsearch
8. Ansible Github Page. https://github.com/ansible
9. Hacker New, Dockerhub Security Breach April 2019. https://thehackernews.com/2019/04/docker-hub-data-breach.html
10. Cfengine Github Page. https://github.com/cfengine/core
11. Walsh, D.: Docker security features in RHEL7. https://opensource.com/business/14/7/docker-security-selinux
12. Seccomp Documentation Page. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt
13. Docker User Namespace Remap Configuration Documentation. https://docs.docker.com/engine/security/userns-remap/
14. Using eBPF to Bring Kubernetes-Aware Security to the Linux Kernel by Dan Wendlandt, Isovalent. https://docker.guru/2019/07/10/using-ebpf-to-bring-kubernetes-aware-security-to-the-linux-kernel-dan-wendlandt-isovalent/
15. Harbor Homepage. https://goharbor.io/

16. CLAIR Github Page. https://github.com/coreos/clair
17. Nivlheim Github Page. https://github.com/usit-gd/nivlheim
18. Rundeck Homepage. https://www.rundeck.com/open-source
19. Kubernetes Homepage. https://kubernetes.io/
20. Cloud Native Computing Foundation Home Page. https://www.cncf.io/
21. Kubernetes Wikipedia Page (Histroy). https://en.wikipedia.org/wiki/Kubernetes#History
22. Kubernetes Distributions and 'Kernels' - Tim Hockin & Michael Rubin, Google. https://www.youtube.com/watch?v=fXBjA2hH-CQ
23. Isenberg, K.: Hard Problems Regarding Kubernetes in Production. https://twitter.com/KarlKFI/status/1020518198817406976
24. Kubespray Github Page. https://github.com/kubespray
25. NAISible Github Page. https://github.com/nais/naisible
26. Documentation of OKDs EFK Stack. https://docs.okd.io/latest/install_config/aggregate_logging.html
27. Documentation of Filebeat Advanced Autodiscovery. https://www.elastic.co/guide/en/beats/filebeat/6.7/configuration-autodiscover-advanced.html
28. Documentation of OKDs Advanced Audit Logging. https://docs.okd.io/latest/install_config/master_node_configuration.html#master-node-config-advanced-audit
29. Zabbix Home Page. https://www.Zabbix.com/
30. OKD Prometheus Operator and Cluster Monitoring. https://docs.okd.io/3.11/install_config/prometheus_cluster_monitoring.html
31. Zabbix' Prometheus Integration. https://Zabbix.com/documentation/current/manual/config/items/itemtypes/prometheus
32. GitOps, Coined by WeaveWorks. https://www.weave.works/blog/gitops-operations-by-pull-request
33. Openstack RDO Project Homepage. https://www.rdoproject.org/
34. Norwegian Cloud Infrastructure for Research and Education (UHIaaS) Homepage. http://www.uh-iaas.no/
35. Terraform Homepage. https://www.terraform.io/