Real-World, Self-Hosted Kubernetes Experience

Mike Packard
Texas Advanced Computing Center
Austin, Texas, USA
mpackard@tacc.utexas.edu

Christian Garcia
Texas Advanced Computing Center
Austin, Texas, USA
cgarcia@tacc.utexas.edu

Justin A. Drake
Texas Advanced Computing Center
Austin, Texas, USA
jdrake@tacc.utexas.edu

Joe Stubbs
Texas Advanced Computing Center
Austin, Texas, USA
jstubbs@tacc.utexas.edu

ABSTRACT

Containerized applications have exploded in popularity in recent years, due to their ease of deployment, reproducible nature, and speed of startup. Accordingly, container orchestration tools such as Kubernetes have emerged as resource providers and users alike try to organize and scale their work across clusters of systems. This paper documents some real-world experiences of building, operating, and using self-hosted Kubernetes Linux clusters. It aims at comparisons between Kubernetes and single-node container solutions and traditional multi-user, batch queue Linux clusters.

The authors of this paper have background experience first running traditional HPC Linux clusters and queuing systems like Slurm, and later virtual machines using technologies such as Openstack. Much of the experience and perspective below is informed by this perspective. We will also provide a use-case from a researcher who deployed on Kubernetes without being as opinionated about other potential choices.

CCS CONCEPTS

• **Information systems** → **Information storage systems**; **Storage architectures**; • **Computer systems organization** → **Distributed architectures**; • **Software and its engineering** → **Reusability**; • **Software and its engineering** → **Software prototyping**; **Software version control**; **System administration**.

KEYWORDS

Abaco, bare-metal, containers, devops, Docker, reproducibility, repeatability, ETL, Functions-as-a-Service, gitops, Kubernetes, Tapis

ACM Reference Format:
Mike Packard, Christian Garcia, Justin A. Drake, and Joe Stubbs. 2021. Real-World, Self-Hosted Kubernetes Experience. In *Practice and Experience in Advanced Research Computing (PEARC '21), July 18–22, 2021, Boston, MA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3437359.3465603

1 INTRODUCTION

Over the course of two years, we have built several Kubernetes [28] clusters at the Texas Advanced Computing Center [25] for the purpose of learning about the technology and to ultimately provide better service to the scientific community, who are increasingly using container technology in pursuit of their computational goals.

As of this writing these clusters have provided service to many projects and domains. Over 1200 containers have been run on the cluster in the last 30 days; currently 500 Running. Several long-running pods have remained up for months. Over 100 namespaces, from small individual user projects to funded projects, including:

- Dashboards on the UT Austin COVID-19 Modeling Consortium [14]. They were able to scale the dashboard during high load times without asking for more virtual machines. The system has handled over 700,000 page views in 1 year.
- Designsafe [32]. Expanded offering of Jupyter notebooks with higher memory and CPU allocation vs. virtual machines.
- Tapis [33]. Took advantage of Kubernetes deployment standards to manage a complex stack of containers, services, and storage.
- Galaxy [30]. Ability to start and stop large numbers of custom containers for training workshops.
- TACC's production Jupyter Notebook service [31]. The added capacity of the cluster allowed more and larger notebooks to run.

Users have taken advantage of Kubernetes' unique features such as:

- *Easy Deployment*: As opposed to managing separate virtual machines with small numbers of containers, Kubernetes provides easy deployment of containers to a cluster. Think "Slurm for containers."
- *Reproducibility*: To use the cluster, users must codify their app into a YAML text file, that can be understood and used by others.
- *Easy Continuous Integration and Deployment*: YAML files can be checked into a repository for CI/CD purposes.
- *Ease of Validation*: Any new software changes can be tested quickly, without impacting production.
- *Upcycling*: Older hardware from other systems can be upcycled easily. Heterogeneous compute nodes are not a problem.

- *Resource Sharing*: Kubernetes allows better sharing of resources (hundreds of containers fit on a single node).
- *Resource Availability*: More hardware available to single containers (some nodes have 256 GB).
- *Cost Effective*: Whole stack is built from free & open source tools.
- *Simplified DevOps for web services*: Kubernetes solves a host of problems including service discovery, scaling, load balancing, monitoring and self-healing

2 CLUSTER ADMINISTRATION

Folks with Linux system administration experience should not be intimidated by running their own Kubernetes cluster. It is relatively simple to set up on a single node or small number of nodes for experimentation and demonstrates excellent stability. Upgrades in place have caused little downtime.

In order to get the most vanilla Kubernetes experience and gain experience with its components, we used the "kubeadm" installation method [9]. A lot of automation tools make assumptions and system changes behind the scenes that end up being useful to know about later. We tend to do manual installations first to understand the steps, then use our familiar automation tools for subsequent installations. Going back and automating with comfortable tools (e.g. Ansible [4] or bash scripts) gives us full control over what is deployed.

Due to the relative ease of setup using kubeadm, we have not extensively explored the low-resource/development-oriented Kubernetes derivatives such as MicroK8s [20] or K3s [8]. By following the instructions one can have a working 1+ node functional Kubernetes stack with only about 5 commands [16].

Once the cluster itself is up, we focused on customizing for what we considered would be the most critical components for users to be able to get work done, namely:

- User separation (namespaces)
- Network separation (by namespace, using Flannel [17] or Calico [21])
- Persistent storage
- External network visibility for apps

2.1 Persistent Storage

We provide storage for the cluster using Ceph [15]. We have had great experience re-purposing used, heterogeneous hardware into a storage cluster that has been relatively stable. As with Kubernetes, we installed and managed the cluster somewhat manually using standard installation instructions [3]. The cluster has remained usable for 3 years, through several major software updates and hardware disk failures.

Through the Kubernetes RBD (Rados Block Device) storage class driver and open source RBD provisioner [13] users are able to create their own persistent volumes and attach them to their containers. This works well for apps like databases, in which it is critical that data not be lost between restarts of the application container.

Other workflows, e.g. in a data import mechanism, might require that several applications read and write from a single directory. While RBD does not support this Read-Write-Many access mode [12] directly, there is a simple workaround: It is easy to set up an NFS server container in one's own namespace and export the volume to the desired containers. This is a good example of granting users the ability to provide for their own needs without admin intervention.

If given the credentials to do so by their Ceph admin, users may also mount CephFS [1] directories as volumes directly in their containers.

2.2 External Connectivity

We do not provide automatic public internet connectivity to our Kubernetes cluster. This is done for technological, security, and policy reasons. As of this writing we provide manually-configured proxy access to services in the Kubernetes cluster. This is the only non-self-serve aspect of the cluster.

Ingress [7] is considered a basic component in Kubernetes API, but implementation on a bare metal/VM cluster introduces some challenges, like provisioning IPs, and user permissions among them. We have not yet attempted to implement a self-service functionality for users yet. This is likely the next large Kubernetes challenge to overcome in our self-hosted environment. The three components required are dynamic IP addresses, dynamic proxy setup, and Domain Name Service (DNS).

MetalLB [10] is a third-party plugin for Kubernetes that allows admins or users to provision dynamic floating IP addresses. We have tested this and it works well. It could be one component of a self-hosted Ingress.

There are several dynamic proxy providers, some that are internal to Kubernetes and some that live outside the cluster. We have not been able to evaluate them yet.

DNS is a challenge because it requires the user having control of their domain name service, and making changes after a dynamic IP is assigned. Further complicating matters is that SSL certificates are usually required, which can involve yet a third entity.

3 KUBERNETES ADMINISTRATION

The following section will be of particular interest to project admins or those responsible for devops, gitops, or software stack deployments.

One of the biggest benefits for devops staff is that Kubernetes forces users to engage in reproducibility best practices. Configuration, deployment, and storage must all be defined in easy-to-read YAML files before deploying to the cluster. The only part of deployment that is not *de facto* transparent is the container image. Everything is very discoverable. By doing this, pods are able to crash and be restarted by Kubernetes almost instantly, with little to no impact to the application user.

3.1 User Isolation

Since we are operating a shared Kubernetes cluster, we need a way to separate user workloads. A simple way to achieve this is to create one namespace [11] for each user. In Kubernetes, access to the API is controlled by tokens, and a default token is created for each namespace. We create a fairly low-powered "login" node, similar to an HPC Linux cluster. Users who are granted access to the system login via ssh and find their credentials for the Kubernetes namespace already present in their `~/.kube/config` file. This is sufficient to provide most users all the isolation they need to get started creating application stacks.

Kubernetes does support much more elaborate permissions models via role-based access control (RBAC) [22], but isolation by namespace achieves much of the desired result. It is similar to non-root user accounts in a Linux environment. We usually create a namespace with the same name as the Linux user account accessing it.

3.2 Quotas and Resource Management

The namespace-centric model allows for some decent built-in resource limits via the ResourceQuota API object [23]. Namespaces may be limited to aggregate use of CPU, memory, GPU, and other consumables. Admins can configure their quotas to achieve different goals for users.

In the case of a fairshare model, one could configure the quotas so that no one namespace may consume too large a majority of the cluster resources, blocking others.

In another model–as in our Jupyter Notebook cluster–we can ensure that every notebook gets guaranteed access to a certain amount of memory & CPU. Conveniently, fractions of CPU are possible to enforce.

Anecdotally, we have found that a single compute node can support hundreds of containers as long as they are not all going full-bore all the time (i.e. are not all compute-heavy codes). Often, overconsumption of resources is simply not a problem for the kinds of applications being used.

3.3 Deprecation of Docker as Kubernetes runtime

In 2020 it was announced that Kubernetes would discontinue support for Docker and transition to containerd as preferred container runtime backend “containerd" [6].

Much has been written about this transition but this mostly affects cluster admins, not users. Docker Hub is still used for the primary container repository. Since containerd runs Docker images natively, most users will never know if their Kubernetes runtime is Docker- or containerd-based.

After some experimentation, it is possible–with a small amount of downtime–to convert a Docker cluster to using containerd backend with only a restart to user pods. Ideally though, a whole new cluster should be built and user workloads moved.

Initial testing has shown containerd to be at least as stable as Docker in our environment, but as of this writing we have not had enough time with containerd to make a long-term judgement.

Containerd offers fewer mature management tools than Docker. In particular, the `docker system prune` has been useful for reclaiming resources such as old images and volumes. Presumably these tools will become more mature in containerd as it gains adoption.

4 USING KUBERNETES

Users who have already containerized their applications can benefit from Kubernetes almost immediately. This section focuses on developers and researchers who wish to deploy apps and get work done with as little interference from the platform as possible.

By the time an app is successfully deployed to Kubernetes, one has already done much of the legwork to deploy it repeatably and reliably:

- By building containers and uploading them to a registry, they have codified the ability to build the app and have it run consistently on most Linux systems.
- By using Persistent Volume Claims, we have insured that our valuable data will be preserved even if containers die.
- By using ConfigMaps, we have separated our dynamic configuration files.

Regarding the primary command for interacting with Kubernetes, we have found it advantageous to alias `kubectl` as `k`. This saves many typed keystrokes, and helps to avoid the controversial topic of how to pronounce the command [26].

4.1 Development Process

The Kubernetes application development process can be cumbersome. Users who are not used to distributed and networked systems can find the learning curve steep. Successfully deploying an application means creating a viable container image, mounting configuration files, setting environment variables, allocating network Services [24], viewing logs, etc.

The most common place for users to post their publicly-available images is on Docker Hub [29]. Kubernetes necessarily requires access to an image *registry*, or repository of container images. Our normal mode of image creation is to develop and debug the individual containers on local resources (e.g. Laptop or Linux virtual machine with Docker.) The user then uploads the image to an image registry. It is possible to build container images directly on the Kubernetes cluster using a tool called Kaniko [2].

Traditional interactive debugging/tracing tools do not work on a remote cluster, so users must rely on logs and performance analysis tools. There are quite a few tools to assist with this, but we do not yet have experience with them. One example of such is Jaeger [19]. We plan to evaluate these types of tools in the future.

4.2 Automation Helpers

There are many tools devoted to further automating the deployment of apps on Kubernetes. Helm [18] is the most popular. Helm employs “charts" which contains all of the resource definitions necessary to run an application, tool, or service inside of a Kubernetes cluster. We found that some Helm charts work out-of-the-box and some require deeper knowledge of the specific Kubernetes deployment and modification of the chart.

We also found that sometimes these automated tools make the assumption that the user is the owner of the cluster and/or has elevated administrative privileges. In our case users do not have such permissions on the cluster, and end up troubleshooting the automation tool instead of their application.

Thus, in keeping with the theme of developing a deep understanding of Kubernetes initially, we have yet to explore these high-level tools in much detail. They can be very powerful and we look forward to evaluating them in the future.

Next we will describe the experience of a real-world user who has successfully taken advantage of the Kubernetes clusters at TACC, with no prior Kubernetes experience.

5 USER CASE STUDY: INFLUXDB AND APPLICATION

This example involved prototyping and optimizing solutions for processing and storing heterogeneous time-course data. It describes experiences with InfluxDB [27] and Kubernetes by a single researcher, using a user-level namespace within the shared Kubernetes cluster.

The Health Analytics and Life Sciences Computing Groups at TACC provide computing and data storage support for the DARPA Warfighter Analytics using Smartphone for Health (DARPA-WASH) program. The DARPA-WASH program aims to uncover physiological representative signals embedded within data collected from sensors (e.g. accelerometer, gps, light sensors, etc.) built in to modern smartphones [5]. Cohort studies, in which a smartphone application facilitated the longitudinal collection of various data streams from participants' phones, generated significant amounts of heterogeneous time course data. One of the challenges presented was to take this raw data (150M small files amounting to 10TBs), transform it, and host it in an accessible, scalable way to support future analyses.

The Kubernetes deployment at TACC offered the perfect environment to iteratively test and prototype the various components of an ETL pipeline. It is important to note that the user had limited experience in standing up and managing databases as well as system administration in general at the start of this project. From their perspective, one of the strengths of the Kubernetes environment was the ease with which resources are accessed, managed and provisioned. The abstraction of such tasks to Kubernetes objects allowed them to more quickly focus on prototyping the ETL pipeline. For example, in a matter of minutes they were able to deploy and explore various databases using Kubernetes deployments and pre-built Docker images hosted on Docker Hub. They ultimately chose to use InfluxDB because it is optimized for reading, writing, and querying high-density time series data, provides UI and dashboarding tools, and supports a number of different client libraries (e.g. Python, Java, Go).

Two separate persistent volume claims were created and associated with two NFS servers: one pair was used to host the raw, smartphone data and the other was used to mount the InfluxDB storage engine. Separating the two ensured that reading and writing IO, which was expected to be considerable given the size and density of the data, were ultimately uncoupled from a performance optimization standpoint. Kubernetes made it trivial to trade out Pods responsible for different steps in the pipeline. For example, shortly after deploying InfluxDB 1.8, version 2.0 was released, which included significant changes. They were able to maintain the deployment of InfluxDB 1.8 while simultaneously exploring version 2.0 using a separate pod. The researcher particularly appreciated the fact that these deployments were containerized and that they did not need to keep track of two different installs, which use similar if not the same PATH variables, on their local machine.

The other major component of the pipeline was responsible for transforming the data into the necessary format (i.e. InfluxDB line protocol) for ingress into the database. Again, the inherent modularization of the Kubernetes environment was ideal for testing different approaches of converting and loading data. In one such effort they deployed a "data science" pod (mounted with NFS server hosting the raw data) and leveraged Kubernetes NodePort service to access interactive Jupyter notebooks and the InfluxDB Python API. In their resulting solution they created a custom Docker image containing open-source Go packages to convert raw CSV data to InfluxDB line protocol and asynchronously post data to the Influx server. Ultimately, they were able to achieve a write performance of 2.6 million values/second using 15 processors and leveraged the substantial RAM provided by the Kubernetes deployment at TACC. They are now in the process of deploying this solution in a production environment.

The researcher commented that the hardware available to each pod was incredible: much larger than they previously had access to.

The experience with Kubernetes was largely one of success. It allowed them to more rapidly, iteratively test different components of the ETL pipeline independently when compared to purely VM options available. The size of the data and hardware requirements prevented local development. While the researcher reported a learning curve, it was not overly prohibitive. The user looks forward to future development on the system.

6 CONCLUSION

Kubernetes has become a popular tool for deploying both long-running applications and short-term scientific computation jobs. Here we have documented our experiences in building, operating, and using Kubernetes in real-world scenarios. Hopefully others may benefit from this experience and apply it in their own environments.

ACKNOWLEDGMENTS

This work was made possible by the National Science Foundation, the University of Texas at Austin, and Texas Advanced Computing Center. We thank them and in particular TACC staff for their ongoing support.

REFERENCES

[1] 2016. Ceph File System. https://docs.ceph.com/en/latest/cephfs/index.html.
[2] 2018. Kaniko. https://github.com/GoogleContainerTools/kaniko.
[3] 2019. Install Ceph. https://ceph.io/install/.
[4] 2020. Ansible. https://www.ansible.com.
[5] 2020. DARPA-WASH. https://www.darpa.mil/program/warfighter-analytics-using-smartphones-for-health.
[6] 2020. Don't Panic: Kubernetes and Docker. https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/.
[7] 2020. Ingress. https://kubernetes.io/docs/concepts/services-networking/ingress/.
[8] 2020. K3s. https://k3s.io.
[9] 2020. kubeadm. https://kubernetes.io/docs/reference/setup-tools/kubeadm/.
[10] 2020. MetalLB. https://metallb.universe.tf.
[11] 2020. Namespaces. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.
[12] 2020. Persistent Volumes. https://kubernetes.io/docs/concepts/storage/persistent-volumes.
[13] 2020. RBD Volume Provisioner for Kubernetes. https://github.com/kubernetes-retired/external-storage/tree/master/ceph/rbd.
[14] 2020. UT Austin COVID-19 Modeling Consotrium. https://covid-19.tacc.utexas.edu.
[15] 2021. Ceph. https://ceph.io.
[16] 2021. Creating a cluster with kubeadm. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/.
[17] 2021. Flannel. https://github.com/flannel-io/flannel.
[18] 2021. Helm. https://helm.sh.
[19] 2021. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io.
[20] 2021. MicroK8s. https://microk8s.io.
[21] 2021. Project Calico. https://www.projectcalico.org.

[22] 2021. RBAC. https://kubernetes.io/docs/reference/access-authn-authz/rbac/.
[23] 2021. ResourceQuota. https://kubernetes.io/docs/concepts/policy/resource-quotas/.
[24] 2021. Services. https://kubernetes.io/docs/concepts/services-networking/service/.
[25] 2021. Texas Advanced Computing Center (TACC). https://www.tacc.utexas.edu.
[26] December 8, 2017. Canonical pronunciation of "kubectl". https://twitter.com/arungupta/status/939168964411838464?lang=en.
[27] Sept 29, 2020. Influx DB: Time series DB. https://www.influxdata.com/products/influxdb-cloud/.
[28] Sept 30, 2020. Kubernetes: Container Orchestration. https://kubernetes.io.
[29] September 9, 2019. Docker Hub. https://hub.docker.com.
[30] Enis Afgan, Dannon Baker, Bérénice Batut, Marius van den Beek, Dave Bouvier, Martin Čech, John Chilton, Dave Clements, Nate Coraor, Björn A. Grüning, Aysam Guerler, Jennifer Hillman-Jackson, Saskia Hiltemann, Vahid Jalili, Helena Rasche, Nicola Soranzo, Jeremy Goecks, James Taylor, Anton Nekrutenko, and Daniel Blankenberg. 2018. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. *Nucleic Acids Res.* 46, W1 (2018), W537–W544. https://doi.org/10.1093/nar/gky379
[31] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87 – 90.
[32] Ellen M Rathje, Clint Dawson, Jamie E Padgett, Jean-Paul Pinelli, Dan Stanzione, Ashley Adair, Pedro Arduino, Scott J Brandenberg, Tim Cockerill, Charlie Dey, et al. 2017. DesignSafe: new cyberinfrastructure for natural hazards engineering. *Natural Hazards Review* 18, 3 (2017), 06017001.
[33] J. Stubbs et al. 2021. Tapis: An API Platform for Reproducible, Distributed Computational Research. *Future Generation Computer Systems* (2021).