A grid site reimagined: building a fully cloud-native ATLAS Tier 2 on Kubernetes

Ryan Paul Taylor[1,*], *Jeffrey Ryan* Albert[1], and *Fernando Harald* Barreiro Megino[2]
on behalf of the ATLAS Computing Activity

[1]University of Victoria, British Columbia, Canada
[2]University of Texas at Arlington, Texas, United States of America

Abstract. The University of Victoria (UVic) operates an Infrastructure-as-a-Service scientific cloud for Canadian researchers, and a Tier 2 site for the ATLAS experiment at CERN as part of the Worldwide LHC Computing Grid (WLCG). At first, these were two distinctly separate systems, but over time we have taken steps to migrate the Tier 2 grid services to the cloud. This process has been significantly facilitated by basing our approach on Kubernetes, a versatile, robust, and very widely adopted automation platform for orchestrating containerized applications. Previous work exploited the batch capabilities of Kubernetes to run grid computing jobs and replace the conventional grid computing elements by interfacing with the Harvester workload management system of the ATLAS experiment. However, the required functionality of a Tier 2 site encompasses more than just batch computing. Likewise, the capabilities of Kubernetes extend far beyond running batch jobs, and include for example scheduling recurring tasks and hosting long-running externally-accessible services in a resilient way. We are now undertaking the more complex and challenging endeavour of adapting and migrating all remaining services of the Tier 2 site — such as APEL accounting and Squid caching proxies, and in particular the grid storage element — to cloud-native deployments on Kubernetes. We aim to enable fully comprehensive deployment of a complete ATLAS Tier 2 site on a Kubernetes cluster via Helm charts, which will benefit the community by providing a streamlined and replicable way to install and configure an ATLAS site. We also describe our experience running a high-performance self-managed Kubernetes ATLAS Tier 2 cluster at the scale of 8 000 CPU cores for the last two years, and compare with the conventional setup of grid services.

1 Introduction

For over a decade, the ATLAS experiment [1] at the LHC and UVic have been at the forefront of evaluating and adopting cloud technologies, particularly in the context of using cloud infrastructure to deliver virtual resources for processing high energy physics workloads, and integrating this approach with the paradigm of grid computing [2][3][4][5]. More recently [6], this activity (and the field of cloud computing in general) has coalesced around container-based methodologies such as Kubernetes, a platform for managing and automating

*e-mail: rptaylor[at]uvic[dot]ca

containerized applications in a distributed and reliable manner using a microservice architecture, marking a new approach known as cloud-native computing, which shifts the focus from provisioning virtual machines on an Infrastructure-as-a-Service cloud to deploying and orchestrating containers in a distributed cluster. In broad terms, cloud-native computing can be understood as employing abstraction and virtualization foremost throughout the application level, rather than solely at the infrastructure level.

Kubernetes and cloud-native computing present a significant opportunity. Kubernetes is the second-largest open-source project in the world, after the Linux kernel, with approximately 75 000 contributors [7], and it is estimated that there are several million Kubernetes clusters deployed [8]. The volume of development, engineering, testing and quality assurance effort that is put into Kubernetes constitutes a highly beneficial network effect and positive feedback loop.[1] Managed Kubernetes offerings are available from all major commercial cloud providers, an increasing number of research and academic sites are deploying on-premise clusters, and the available knowledge pool is vast. It is frequently the case that an internet search for a problem will quickly yield useful information and readily applicable solutions provided by others who have already confronted and addressed it.

Our previous work began capitalizing on this opportunity by demonstrating the use of Kubernetes to run batch jobs for ATLAS at a limited scale, augmenting the computing resources of a traditional Tier 2 site [6]. Based on that success, we have significantly expanded the program of work to encompass and transform the entire UVic ATLAS Tier 2 site: using Kubernetes as the exclusive full-scale computing resource, migrating all ancillary services from traditional servers to Kubernetes, and integrating the grid storage into Kubernetes. Our approach centers as much as possible on Helm, a cloud-native tool for configuration and lifecycle management of applications on Kubernetes. Helm applications are installed and configured via charts, which are packages of YAML templates and customizable values that define the application in Kubernetes. Unlike traditional package or configuration management tools, which typically perform imperative operations on top of pre-existing servers, a Helm chart can fully encapsulate and represent the state of a complex distributed application and enable it to be deployed in a reproducible manner.

2 Tier 2 computing

2.1 Harvester operations

Building on our previous experience [6] running PanDA [9] workloads on Kubernetes via the resource-facing Harvester [10] system, further collaboration between the UVic and PanDA teams led to the sharing of operational experience in order to optimize job execution, improve configurability and robustness, and reduce operational overhead. Configuration templates for job submission, including general parameters such as the container image, are managed in Git. Site administrators can edit a template in Git, and it will automatically be pulled and applied on the Harvester instance. Operations teams can now also edit various job configuration parameters directly in the CRIC information system [11], from which they will be dynamically read and applied by Harvester. For example, we can easily modify CPU, memory, and ephemeral storage requests and limits in order to optimize resource usage while maintaining a sufficient margin for excess usage to avoid overly strict termination of jobs. Also, configuration of affinities and anti-affinities between jobs of different sizes can facilitate efficient scheduling and optimal node utilization. Operational experience has also guided us to reimplement the handling of jobs during short unavailabilities of the network or Kubernetes

[1] Conversely, small niche software projects can struggle to reach sustainability and gain adoption.

API. Previously, Harvester would terminate jobs if it could not retrieve their status from the Kubernetes API within a fixed time, but now it tolerates periods when the API is unreachable or slow to respond so that jobs can continue running during brief interruptions.

2.2 Cluster operations and scaling

After establishing proof of concept by running at the scale of about 500 CPU cores for over a year, we prepared for a production-ready deployment by first creating a smaller but otherwise identical Kubernetes cluster, complete with a PanDA queue for functional test jobs, for development and acceptance testing. Prior to implementation on the production cluster, every configuration change and upgrade is first tested on the development cluster — an essential practice for reliable operation. Deploying and maintaining an additional cluster is relatively easy since we are using virtual infrastructure provided by our OpenStack cloud.

As we expanded the production cluster to about 8 000 cores, as shown in Figure 1, we implemented several improvements in order to keep the cluster reliable and performant at this scale, such as using a faster type of storage volume for the IOPS-intensive distributed database etcd, increasing the size of the etcd backend disk quota so that compaction can clean up old records before the database exceeds the quota, applying priority classes to important cluster services to ensure they can run reliably without resource contention when the cluster is fully utilized, reducing the load on the data store of the Calico network fabric by enabling the Typha daemon for caching queries, and trunking a VLAN to all the hypervisor nodes in our cloud so that the VMs of the cluster can be in a flat layer 2 network, thereby avoiding the need for network encapsulation in Calico and eliminating packet overhead for traffic in the cluster. However, this network mode entails the use of BGP to distribute routes in the cluster, by establishing a full mesh of peerings between nodes, implying resource usage that scales quadratically with the number of nodes. With over 130 nodes currently in the production cluster, we had to allocate more RAM to Calico, but scaling the cluster beyond the current level will require setting up dedicated nodes functioning as BGP route reflectors, to form a more efficient partial mesh topology.

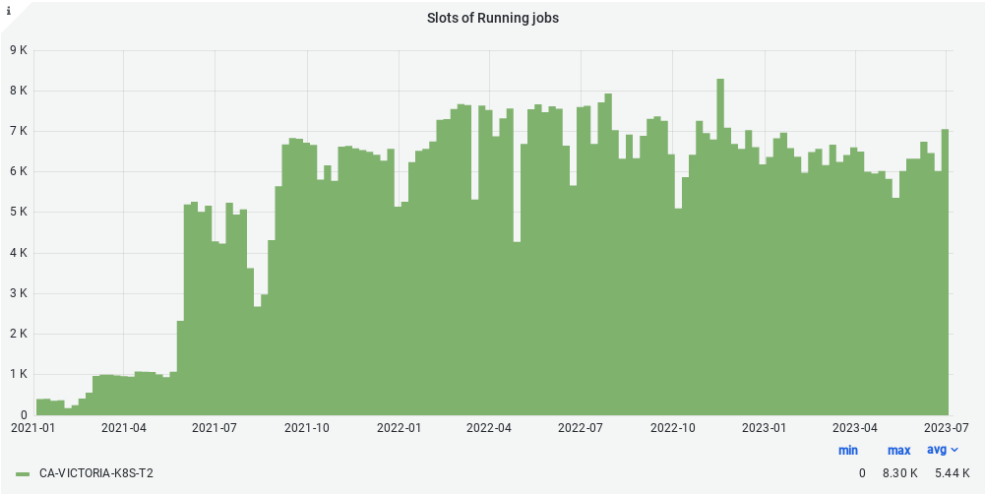


Figure 1: Number of CPU cores used by ATLAS jobs running on the CA-VICTORIA-WESTGRID-T2 Kubernetes cluster.

3 Tier 2 storage

We have operated a dCache [12] storage element on physical hardware since our site was commissioned in 2010. However, motivated by the goals of physical consolidation of all data onto our Ceph [13] cloud storage cluster and logical consolidation of all site services onto Kubernetes in our cloud, we are evaluating EOS [14] as an alternative solution. Thanks to the ScienceBox [15] project, a Helm chart for EOS is available, making it easy to install on Kubernetes. Moreover, using EOS on a CephFS filesystem is a solution that has been previously investigated [16].

However, the use cases supported by the EOS Helm chart so far have only covered access for trusted internal clients using local authentication within a Kubernetes cluster, with access methods such as a FUSE mount and the EOS CLI, whereas a grid storage element must be accessible to the world via the XRootD and HTTPS protocols, using X509 VOMS proxies for authentication and authorization, which further entails installing host certificates and grid certificate authorities, and regularly updating certificate revocation lists.

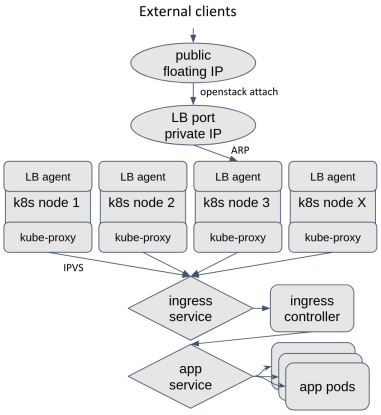
We have contributed several improvements to the EOS Helm chart to extend its functionality using generic configuration hooks to support these needs, and designed a scalable network architecture for external access to EOS in Kubernetes. Whereas a traditional server-based application deployment can be exposed to an external network by simply setting up a publicly routable IP address on the server, Kubernetes includes layers of network functionality providing advanced capabilities such as service discovery, load balancing as a service, and IP failover for high availability, which enable dynamic scaling of an application, spreading traffic across multiple nodes or containers, and fault tolerance against the loss of any individual node.[2]

Our standard network architecture for simple low-bandwidth web applications on Kubernetes involves ingress to the cluster via a single IP address, which has high availability across cluster nodes by means of the PureLB load balancer, and the Traefik ingress controller, which handles TLS termination and service routing (primarily at layer 7, based on header inspection in the case of HTTP, as well as Server Name Indication (SNI)), as shown in Figure 2a. However, an ingress controller can not be used for EOS because it is not possible to use SNI in the XRootD protocol [17]. More importantly, for a high-performance Tier 2 storage element we need a method of load-balancing that can provide aggregate bandwidth of $\gtrsim$100 Gb/s, far more than the capacity of a single typical node. Also, the design of EOS requires each data serving component (called an FST) to be individually addressable, which clashes somewhat with the usual Kubernetes approach of a single service backed by many identical containers. However both of these problems can by solved by reworking the FST service definitions[3] and leveraging PureLB to manage and link an IP address to each one, as shown in Figure 2b. This enables the aggregate bandwidth of an EOS instance on Kubernetes to be increased in the same natural way as for a traditional server-based EOS deployment, by horizontally scaling the number of FSTs.

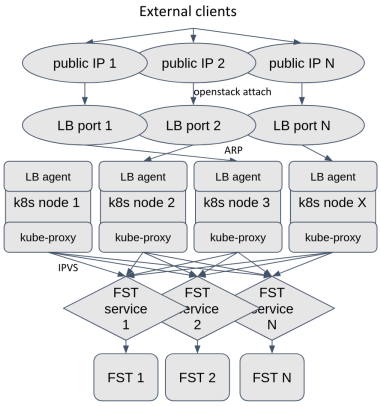
A major advantage of using Helm to deploy EOS is that we can leverage our pair of development and production Kubernetes clusters to trivially deploy a pair of identically-configured EOS instances for testing and production, whereas building and maintaining an additional grid storage element on traditional servers would incur overhead effort, and maintaining configuration parity between them would be difficult. After finalizing some details of the Helm chart configuration, we plan to benchmark the performance of EOS and CephFS and evaluate the deployment for production-readiness.

[2] For further information see https://kubernetes.io/docs/concepts/services-networking/.

[3] This is accomplished by using the Helm range operator to create a LoadBalancer-type Service for each member of the FST StatefulSet, and specifying the statefulset.kubernetes.io/pod-name selector for each one.



(a) A simple architecture for basic web applications, leveraging an ingress controller. The traffic flows through a single node, and is routed by the ingress controller to the appropriate application service, which load balances at the service level to a set of identical backing containers.



(b) Network architecture for EOS on Kubernetes. Each FST has its own individual service, linked to a private IP and bound to different nodes because the PureLB agent prefers to spread IPs across nodes to maximize available bandwidth. The aggregate bandwidth for N FSTs is that of N nodes.

Figure 2: Different Kubernetes network architectures for receiving and routing traffic. The traffic arrives at the cluster via one or more OpenStack public floating IP addresses, each attached to a private IP. The PureLB load balancer agents use gratuitous ARP to bind each private IP address to a node. Kube-proxy implements the Kubernetes service discovery and routing mechanism and provides transport-layer load balancing within the cluster using IPVS.

4 Tier 2 services

Aside from providing computing and storage resources, the other services an ATLAS Tier 2 site must operate are Frontier-squid and APEL accounting.

4.1 Frontier-squid caching service

Frontier-squid is a caching HTTP proxy service, deployed as a forward proxy to provide scalable low-latency access to both CVMFS [18] and conditions data [19]. To deploy Frontier-squid on Kubernetes, we selected the Helm chart from the ScienceBox [15] repository for its simple, lightweight and container-native approach. However before deploying it in production, we contributed a number of improvements to security, robustness and configurability, such as running as an unprivileged user, setting resource requests and limits, supporting modification of ACLs and network service configuration, and adding a backup URL for redundancy of readiness probes. Using a Kubernetes service and deployment, as shown in Figure 3, makes it trivial to scale up the number of Frontier-squid instances as needed, with client traffic automatically load-balanced across them.

4.2 APEL accounting service

Clients for APEL accounting [20] exist for several traditional batch systems, but to publish accounting records representing the CPU usage delivered by our Kubernetes cluster, we

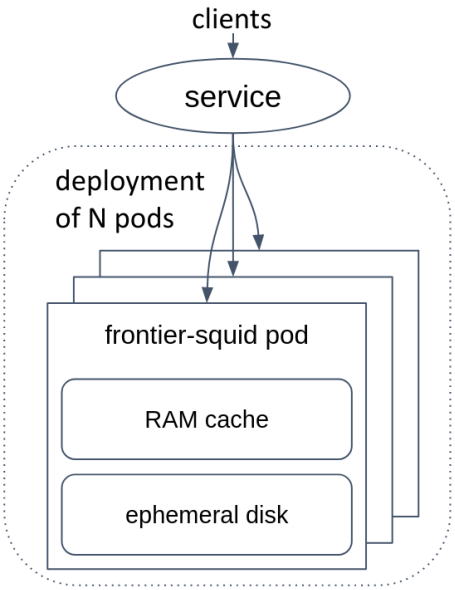


Figure 3: Architecture of the Frontier-squid deployment on Kubernetes.

had to develop one for Kubernetes. Publishing records to the APEL server was challenging because the required information and record format are not documented [21]. However, collecting and processing the job metrics in Kubernetes was substantially facilitated by leveraging built-in functionality of Kubernetes, as shown in Figure 4. It was only necessary to write a modest Python program, and some YAML for the Helm chart to deploy it, in order to create a complete production-ready solution for APEL accounting on Kubernetes, which we call KAPEL.[4]

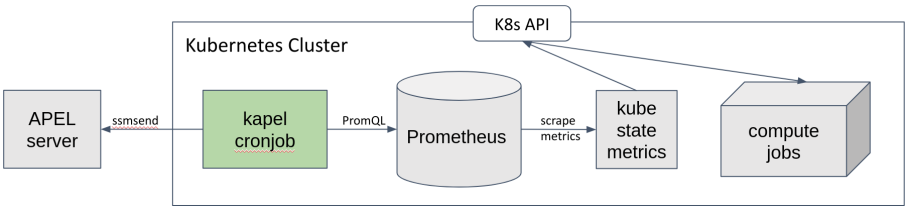


Figure 4: Architecture of KAPEL. The KAPEL Python program runs as a regularly scheduled Kubernetes CronJob. It retrieves records from the Prometheus time-series database, using PromQL to formulate a query that aggregates, filters, and analyzes metrics. Prometheus scrapes and stores metrics from kube-state-metrics, which presents snapshots of the current state of compute jobs in the cluster. After querying and processing the job metrics, KAPEL prepares accounting reports and sends them to the APEL server.

[4]The Python code and Helm chart for KAPEL are available at https://github.com/rptaylor/kapel.

5 Summary

We are forging a path to running a fully cloud-native ATLAS Tier 2 site on Kubernetes. The resilience and automation provided by Kubernetes have significantly simplified site operations and reduced the amount of time spent on mundane technical issues. Re-using existing monitoring and metrics components in the Kubernetes ecosystem to implement an APEL accounting solution illustrates that the burden of developing and maintaining middleware can be much lighter when cloud-native practices are adopted. Similarly, built-in capabilities of Kubernetes provide the functionality needed for remote submission of jobs to an API and scheduling jobs for execution on cluster nodes, eliminating the need to deploy a separate grid computing element and batch system.

This work also has potential implications and benefits for the deployment of grid sites more broadly. While ATLAS and the Worldwide LHC Computing Grid have a very strong history of collaborative information sharing and open-source software practices, the manner in which the software is deployed and operated across the grid tends to be highly variable to the point of being unique to each site. Building a new WLCG Tier 2 site and batch cluster from scratch, commissioning it for production-readiness, and smoothing out the initial bugs is a daunting process that typically requires pre-existing familiarity with specialized grid software, and can take approximately 6–12 months of time. Integrating the grid middleware components together into a functional site, with many different operating system and hardware options, further compounded by the number of available batch systems and storage systems to choose from, is neither straightforward nor standardizable.

Because of these environment-specific differences between sites, sites often develop their own specialized deployment tools and recipes from scratch, or forego automation altogether and rely on manual configuration and operation instead. This situation reveals the need for abstraction layers to hide underlying details and disparities, and stems from the nature of traditional deployment methodologies whereby software is provided as plain packages of files to be installed on individual servers, but deploying a distributed application requires configuring and integrating several components together across multiple servers.[5] This integration work, and the specialized recipes for facilitating it, are sometimes referred to as "glue", in the sense of sticking components together to form a larger whole. Unfortunately, attempting to share or reuse glue is often messy and impractical.

Containerization changes this by delivering pieces of software as encapsulated microservices, which can be orchestrated on a platform like Kubernetes, allowing the relationships between services to be clearly defined and managed, so that a distributed application can be neatly assembled from its component pieces. Moreover, with Helm charts, a deployment becomes code, which can be easily shared, re-used, and improved, bringing the same benefits of open source that are eminent in software development into the operational realm[6] of system administration: the more that people work together in a framework where improvements can be contributed back to a shared code base, the more everyone benefits together. When operators' efforts are mostly spent on developing deployment recipes and addressing issues that are primarily local and site-specific in nature, our collective efforts are less productive. On the other hand, by channeling operational effort into improvements that benefit all users of a Helm chart, we can realize a higher return on investment of effort. Adopting Kubernetes and Helm has already resulted in significant operational benefits for our site, and we believe that it can also reduce the time and effort required to deploy a new ATLAS Tier 2 site by approximately an order of magnitude.

[5] For example, configuring one system to connect to a database on another system, opening a firewall to allow communication between systems, etc.

[6] This is a hallmark of the DevOps paradigm.

References

[1] The ATLAS Collaboration, J. Inst. **3**, S08003 (2008)

[2] F. H. Barreiro Megino *et al* on behalf of the ATLAS Collaboration, J. Phys. Conf. Ser. **396**, 032011 (2012)

[3] S. Panitkin *et al* on behalf of the ATLAS Collaboration, J. Phys. Conf. Ser. **513**, 062037 (2014)

[4] R. P. Taylor *et al* on behalf of the ATLAS Collaboration, J. Phys. Conf. Ser. **664**, 022038 (2015)

[5] R. P. Taylor *et al* on behalf of the ATLAS Collaboration, J. Phys. Conf. Ser. **898**, 052008 (2017)

[6] F. H. Barreiro Megino *et al* on behalf of the ATLAS Collaboration, EPJ Web Conf. **245**, 07025 (2020)

[7] https://www.cncf.io/reports/kubernetes-project-journey-report/ [accessed 2023-08-02]

[8] https://cloudnativenow.com/topics/how-many-kubernetes-clusters-exist-today/ [accessed 2023-08-02]

[9] T. Maeno *et al* on behalf of the ATLAS Collaboration, J. Phys. Conf. Ser. **898**, 052002 (2017)

[10] T. Maeno *et al* on behalf of the ATLAS Collaboration, EPJ Web Conf. **214**, 03030 (2019)

[11] A. Anisenkov *et al*, EPJ Web Conf. **245**, 03032 (2020)

[12] A. P. Millar *et al*, J. Phys. Conf. Ser. **396**, 032077 (2012)

[13] S. A. Weil *et al*, OSDI '06, 307–320 (2006)

[14] A. J. Peters *et al*, J. Phys. Conf. Ser. **664**, 042042 (2015)

[15] E. Bocchi *et al*, EPJ Web Conf. **245**, 07047 (2020)

[16] A. J. Peters & D. C. van der Ster, Comp. Softw. Big Sci. **5**, 25 (2021)

[17] https://github.com/xrootd/xrootd/issues/1951 [accessed 2023-08-02]

[18] A. De Salvo *et al* on behalf of the ATLAS Collaboration, J. Phys. Conf. Ser. **396**, 032030 (2012)

[19] D. Barberis *et al* on behalf of the ATLAS collaboration, J. Phys. Conf. Ser. **396**, 052025 (2012)

[20] M. Jiang *et al*, *Data Driven e-Science* (Springer, New York, 2011) 175-186

[21] https://github.com/apel/apel/issues/212 [accessed 2023-08-02]