Into the Fire: Delving into Kubernetes Performance and Scale with Kube-burner

Sai Sindhur Malleni
Red Hat, Inc.
Bengaluru, India
smalleni@redhat.com

Raul Sevilla Canavate
Red Hat, Inc.
Madrid, Spain
rsevilla@redhat.com

Vishnu Challa
Red Hat, Inc.
Raleigh, US, NC
vchalla@redhat.com

ABSTRACT

This paper introduces Kube-burner[1], an open-source tool for orchestrating performance and scalability testing of Kubernetes[2], with the ability to operate seamlessly across different distributions. We discuss its importance in the cloud native landscape, features and capabilities and delve into its architecture and usage. Additionally, we also present a case study on performance benchmarking using Kube-burner and subsequent analysis to demonstrate its value.

KEYWORDS

kubernetes, benchmark, workload, pods, containers, metrics, performance testing, scale testing, openshift

ACM Reference Format:
Sai Sindhur Malleni, Raul Sevilla Canavate, and Vishnu Challa. 2024. Into the Fire: Delving into Kubernetes Performance and Scale with Kube-burner. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24 Companion), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3629527.3651405

1 INTRODUCTION

Microservices deployed as containers have become the prominent way of building and delivering software [3]. This rise in adoption of container technologies as well as microservices based architectures has elevated the importance of container orchestration engines like Kubernetes in empowering modern application development and delivery. In that sense, Kubernetes serves as the fundamental building block of cloud native infrastructure.

As cloud native is all about building, deploying and managing applications at scale, the performance and scale of the underlying Kubernetes platform is of essence. Kube-burner, with its versatile capabilities in scaling, creating, deleting, and patching Kubernetes resources as per user defined scenarios, along with its ability to collect and index metrics from the monitoring system and benchmark results plays a crucial role in illuminating the previously obscure aspects of Kubernetes performance. Furthermore, its custom measurements and alerting features notify users when Key Performance Indicators (KPIs) indicate that Service Level Objectives (SLOs) have been breached, adding an extra layer of insight.

[1]https://github.com/kube-burner/kube-burner
[2]https://kubernetes.io

2 ARCHITECTURE

Kube-burner is a CLI based tool written in Golang[3] that provides three major classes of functionality: **benchmark orchestration** - creating, deleting and patching Kubernetes resources at scale; **measurements** - detailed metrics obtained with the help of the Kubernetes API, among others, such as how long it takes for a pod to go from scheduling to ready (also known as podLatency) and **observability** - the ability to collect metrics about the Kubernetes platform under load by scraping a user defined set of metrics from the Prometheus[4] monitoring stack and index them along with the benchmark results into a long term storage like the local File system of the host from which kube-burner is run or an Elasticsearch[5]/OpenSearch[6] endpoint for subsequent retrieval, visualization and comparison. Kube-burner is also capable of extracting certain metadata that defines the configuration of the Kubernetes platform and appending that to the benchmark result data to facilitate regression testing and comparisons between different configurations such as Container Storage Interface (CSI) and Container Network Interface (CNI) plugins or different distributions of Kubernetes that have very nuanced differences as has been done in some past work [1] [2] albeit using different tools. The resource creation, deletion and patching functionality is implemented using client-go, while the Prometheus and Elasticsearch/OpenSearch clients as well as the metadata collection is implemented as part of a common library called go-commons that is imported as a module. The objects created by Kube-burner are rendered using the default Golang's template library.

3 BENCHMARK ORCHESTRATION

Kube-burner is available as a binary that can run on Linux, Windows or Darwin based systems across a wide range of CPU architectures including x86_64 and arm64. Kube-burner in its most common usage is invoked by passing a YAML based configuration file to the executable on the CLI. The configuration file contains certain global configuration options such as the endpoints of the Elasticsearch/OpenSearch indexer followed by a list of jobs, with each job having its own set of supported parameters. Each job could create, delete or patch objects at a rate defined by the QPS and Burst parameters within the job. An example job would be one that creates several deployment objects per namespace across several namespaces (determined by the *jobIterations* parameter) and through each deployment creates multiple pods.

Ready-made benchmarks that mimic production workloads are also available txo users to run directly instead of defining their

[3]https://go.dev
[4]https://prometheus.io
[5]https://www.elastic.co
[6]https://opensearch.org

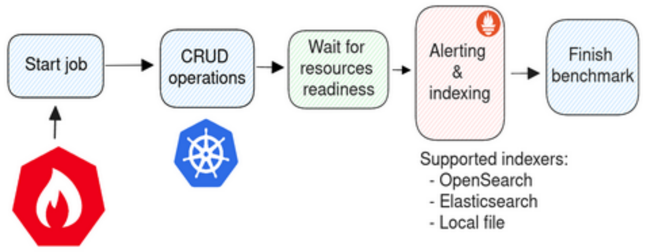

Figure 1: Kube-burner workflow

configuration file, providing a trade-off between flexibility and ease of use.

4 OBSERVABILITY

Performing a benchmark using Kube-burner is relatively simple. However, it is sometimes necessary to analyze and be able to react to some KPIs in order to validate a benchmark. That is why Kube-burner ships metric-collection and alerting systems based on Prometheus expressions. As stated earlier, Kube-burner is capable of connecting to the Prometheus stack running on the Kubernetes platform and extracting metrics pertaining to the platform's response to the benchmark load. The extracted metrics are then indexed to the configured indexer, either the local File system on the host running kube-burner or an Elasticsearch/OpenSearch endpoint for long term storage, retrieval and further integration with tools capable of graphing data such as Grafana[7]. The metrics collection feature is configured through a file referenced by the *metrics-profile* flag, which can point to a local path or URL of a YAML-formatted file containing a list of the Prometheus expressions that Kube-burner will perform one by one after all the jobs are finished. Kube-burner also includes an alerting feature that is capable of evaluating Prometheus expressions in order to fire and index alerts based on user-specified Prometheus expressions. These alerts could be used to indicate anomalies found in certain Key Performance Indicators.

5 MEASUREMENTS

Not all of the data that is capable of providing insights into the performance of the platform during the course of a benchmark run and facilitating debugging once a potential bottleneck has been found can be obtained from the Prometheus stack running on the Kubernetes cluster. For example, there are no readily available metrics in Prometheus to quantify the performance of the platform in terms of the time taken to schedule and run pods during periods of heavy churn on the cluster. Furthermore, there is a need to be able to gather Golang profiling data correlating to the benchmark run from the Kubernetes infrastructure pods such as the API server or etcd. Pprof is another supported custom measurement in Kube-burner which helps the user gather profiling information to further assist advanced debugging. In total, Kube-burner supports three custom measurements during a benchmark run: **podLatency** - for measuring the time it takes for pods to go from scheduling to ready and further reporting quantiles across the entire set of pods created as part of the benchmark, **vmiLatency** - similar to podLatency but for virtual machine instances running on Kubernetes through KubeVirt and **pprof** - for collecting Golang profiling information from Kubernetes infrastructure pods as well as user applications.

[7]https://grafana.com

6 CASE STUDY

As part of continuous testing we undertake at Red Hat to establish the performance and scale leadership of OpenShift (Red Hat's enterprise grade Kubernetes distribution), we use Kube-burner extensively to quantify the performance of every release. A relevant recent use case of Kube-burner has been its role in validating the performance and scalability requirements of a CNI plugin in OpenShift (OVNKubernetes) before transitioning to General Availability (GA) and replacing the previous one (OpenShiftSDN) as the default. Workloads and metrics provided by Kube-burner were crucial to detect the different bottlenecks this plugin had. For example, one of these metrics, podLatency, , can provide deep insights into several aspects of a CNI plugin's performance, scalability and efficiency. We used the 99th percentile from the quantiles reported by the podLatency metric to summarize the time taken for the different pod lifecycle stages, starting from their creation and ending up in a ready status, which includes the network setup of the pods. All of this was done during a benchmark that spins up thousands of pods across the cluster. As can be seen from the figure below, the 4.14 release of OpenShift showed improved scalability by being performant at larger cluster sizes.

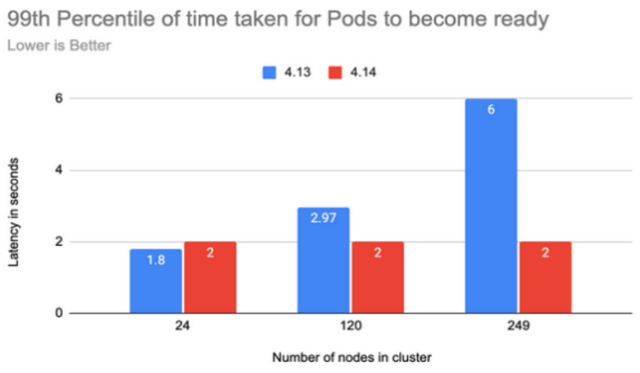

Figure 2: Measurements from Kube-burner graphed to quantify performance improvements compared to previous release

REFERENCES

[1] Heiko Koziolek and Nafise Eskandani. 2023. Lightweight kubernetes distributions: a performance comparison of microk8s, k3s, k0s, and microshift. In ACM/SPEC International Conference on Performance Engineering (ICPE '23), Coimbra, Portugal. ACM, New York, NY, USA. https://doi.org/10.1145/3578244.3583737.
[2] Foutse Khomh Mohab Aly and Soumaya Yacout. 2018. Kubernetes or openshift? which technology best suits eclipse hono iot deployments. In *11th Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 113–120. https://doi.org/10.1109/SOCA.2018.00024.
[3] Olaf Zimmermann. 2017. Microservices tenets. *Computer Science-Research and Development*, 32, 3-4, (July 2017), 301–310. https://doi.org/10.1007/s00450-016-0337-0.