Reproducible Benchmarking of Cloud-Native Applications With the Kubernetes Operator Pattern

Sören Henning[1], Benedikt Wetzel[1] and Wilhelm Hasselbring[1]

[1]*Software Engineering Group, Kiel University, 24098 Kiel, Germany*

Abstract

Reproducibility is often mentioned as a core requirement for benchmarking studies of software systems and services. "Cloud-native" is an emerging style for building large-scale software systems, which leads to an increasing amount of benchmarks for cloud-native tools and architectures. However, the complex nature of cloud-native deployments makes the execution and repetition of benchmarks tedious and error-prone. In this paper we report on our experience with developing a benchmarking tool based on established cloud-native patterns and tools. In particular, we present a benchmarking tool architecture based on the Kubernetes Operator Pattern. Accompanied with a role model and a data model for describing benchmarks and their executions, this architecture aims to simplify defining, distributing, and executing benchmarks for better reproducibility.

Keywords

Benchmarking, Reproducibility, Cloud-native, Kubernetes

1. Introduction

Along with its growing adoption in industry, "cloud-native" [1] software systems and their performance are increasingly studied in research. For example, the performance of microservices, an architectural pattern often classified as cloud-native, has recently been benchmarked [2, 3]. As stated by standardization organizations and research communities, reproducibility is a core quality attribute of benchmarking studies [4, 5]. This applies even more to studies on cloud computing, where execution environments are hardly controllable [6]. Also other often mentioned quality attributes such as usability and verifiability support repeatability, which in turn is a prerequisite for reproducibility. A common approach to improve reproducibility is by providing scripts or the like, which set-up the benchmarking environment, and to share the exact configuration used in the original research.

However, we experienced that for cloud-native systems, this traditional way of providing benchmarks poses significant difficulties. Cloud-native applications typically consist of several processes, running in isolated containers, which have to be scheduled to different computing nodes. They require application-level configurations passed to these containers, comprehensive descriptions of internal and external network interconnection, and integration of different storage systems. These and other aspects make the deployment and operation of cloud-native applications a rather complex task and, hence, also complicate benchmarking.

SSP'21: 12th Symposium on Software Performance 2021, November 9–10, 2021, Leipzig, Germany

soeren.henning@email.uni-kiel.de (S. Henning); stu126940@mail.uni-kiel.de (B. Wetzel); hasselbring@email.uni-kiel.de (W. Hasselbring)

On the other hand, several tools emerged in the cloud-native community for enhancing the deployment and operation. For example, Kubernetes is nowadays the de-facto standard for declarative orchestration, Prometheus collects monitoring data of various sources, and Helm simplifies packaging, distributing, and installing entire applications. Recently, the Kubernetes Operator Pattern [7] has gained much attention as an approach to integrate domain knowledge into Kubernetes. We propose to adopt established cloud-native technologies along with the Operator Pattern for designing cloud-native benchmarking tools that particularly support reproducibility.

After a brief introduction of Kubernetes and the Operator Pattern, we present the necessary data model and system architecture for a benchmarking operator. We complement this paper by an overview of how we apply this approach in our *Theodolite* scalability benchmarking tool.

2. The Kubernetes Operator Pattern

Kubernetes is a declarative orchestration tool for cloud-native applications. Users describe the desired state of an application in so-called resources and in a continuous reconciliation loop, Kubernetes adopts the current system to reach the desired state. A typical example of a resource is a *Deployment*, which describes an application component (e.g., a microservice) consisting of container images, number of replicas and others. A common way for users to interact with Kubernetes is by describing resources in YAML files and applying these files to Kubernetes by using the *kubectl* command line tool.

The Kubernetes Operator Pattern [7] is a recent approach to integrate domain knowledge into Kubernetes' orchestration process. Implementing this pattern involves two things: First, custom resource definitions (CRDs) define new types of resources that can be managed by the Kubernetes API. Second, a dedicated software component (the actual operator), runs inside the Kubernetes cluster and manages the entire life cycle of what is described by the CRDs. This so called operator interacts with the Kubernetes API and reacts to creation, modification, or removal of custom resources (CRs, instances of CRDs).

3. Custom Resource Definitions for Benchmarking

As presented in Section 2, the Operator Pattern combines knowledge of operating Kubernetes with domain knowledge. As the domain at hand is benchmarking, we propose that actors involved in benchmarking describe their operational knowledge in CRDs. In general, we can observe two actors involved in benchmarking:

The Benchmark Designer has knowledge about operating a system under test (SUT) as well as how the quality to be benchmarked can be measured (metric and measurement method). Additionally, he/she defines how metrics and results can be interpreted to compare different SUTs. The Benchmark Designer bundles all of this in a benchmarking tool or sometimes in an artifact that can be interpreted by a benchmarking tool.

The Benchmarker intents to compare and rank different existing SUTs, evaluates new methods or tools against a defined standard, or repeats previous experiments. For this purpose,

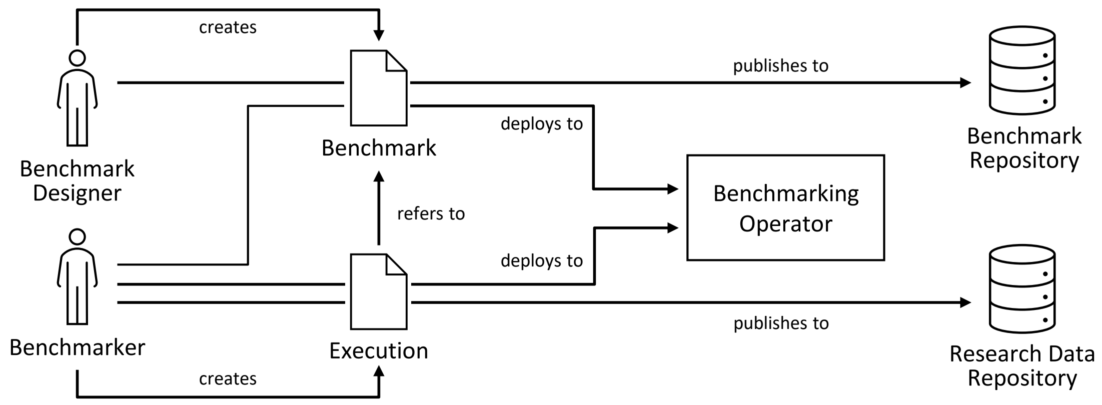

Figure 1: Data and role model for defining benchmarks and their executions.

he/she executes existing benchmarks. A detailed description of this actor can be found by Kounev et al. [8].

Based on this distinctions of roles, we propose a data model in which benchmarks and their executions are separated. Figure 1 illustrates this. We envisage individual CRDs for both benchmarks and executions:

Benchmarks are defined by Benchmark Designers. They describe the SUT as well as the load generation. Additionally, they may describe configurations of the benchmarking method, which are interpreted by the benchmarking tool. Benchmarks can be published as supplemental material of research papers, but ideally they are versioned and maintained in public repositories (e.g., at GitHub). Benchmarks are stateless as they can be executed arbitrarily often.

Executions on the other hand describe a single execution of a Benchmark. They refer to exactly one Benchmark and configure the experimental setup. Definitions of executions can be shared, for example, as part of a research study that benchmarks the performance of different SUTs. When deployed to Kubernetes, Executions have a state, which is typically something like *Pending*, *Running*, *Finished*, or *Failed* if an error occurred (similar to Kubernetes Jobs).

4. Benchmarking Operator Architecture

Standard software components involved in benchmarking are a system under test (SUT), a load generator, an experiment controller, a measurement database, and an (optional) monitoring component for passive observation [9]. We propose a benchmarking tool architecture consisting of these components, which applies the Kubernetes Operator pattern and uses state-of-the-art cloud-native technologies.[1] Figure 2 provides an overview of this architecture and shows how a

[1]In our proposed architecture, we refer to actual technologies instead of abstract component types. The cloud-native ecosystem evolves rapidly and new tools and their underlying concepts often go hand in hand. Nonetheless,

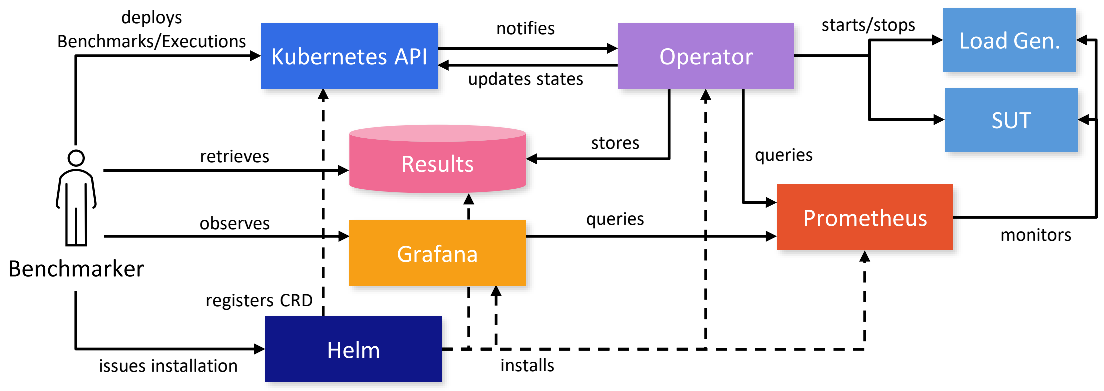

Figure 2: Benchmarking architecture based on the Operator Pattern and cloud-native technologies.

benchmarker will interact with such a benchmarking tool.

To execute benchmarks, a Benchmarker solely interacts with the Kubernetes API, for instance, via tools such as *kubectl* or the Kubernetes dashboard. The Benchmarker registers benchmarks and initiates their executions by applying the corresponding CRs. The operator continuously observes the Kubernetes API for changes in the list of registered executions. It takes on the role of the experiment controller and decides based on a tool-specific policy whether it starts an execution immediately or schedules it to a list of pending executions. A typical constraint is, for example, that only one execution should run at a time to avoid interfering benchmark results. Once a registered execution is started, the operator reads the CRs of that execution and the corresponding benchmark. It creates and starts the SUT and the load generator via the Kubernetes API as well as performs additional cluster configurations.

While executing the benchmark, the operator updates the state of the Execution (e.g., to a state *Running*) and registers events (e.g., once a certain phase of the execution is completed). State and events of an Execution are managed by the Kubernetes API and are used to provide feedback to the Benchmarker. During the entire benchmark execution, Prometheus collects monitoring data from the SUT and the load generator. This can often be done in the same way as the cloud-native applications is monitored in a production deployment. Once a benchmark execution is finished, the operator stops and removes the SUT and the load generator. In addition, it requests collected monitoring data from Prometheus and analyzes them. Also requesting and analyzing monitoring data during an execution to adjust the SUT and the load generator is possible, which is necessary for certain types of benchmarks.

The operator stores benchmarking results persistently (e.g., in a database or as CSV files in a Kubernetes volume), from where the Benchmarker can access them for offline analysis, archiving, or sharing. For the passive observation of benchmark executions, Grafana can be integrated in the architecture to visualize the monitoring data in real time.

Installing a benchmarking tool based on our proposed architecture might quickly become a complex tasks due to the many components and dependencies involved. In particular for reproducibility, it is often also necessary to install a specific version of a benchmarking tool. To

we expect the mentioned tools to be rather matured and that possible successors can be used similarly.

simplify installation and upgrading, we propose to package an operator-based benchmarking tool with Helm. Helm then registers the benchmarking CRDs and installs the operator with all its dependencies.

5. The Theodolite Scalability Benchmarking Tool

Theodolite [10] is our tool for benchmarking scalability of distributed stream processing engines in microservices. We recently rebuild Theodolite according to the architecture presented in the previous sections. Examples for Benchmark and Executions resources are provided with Theodolite's sources.[2]

The Benchmark resource specifies the SUT and the load generator as sets of other resources, which describe their deployment. Since scalability is measured according to a load type and a resource type [11], Theodolite evaluates a SUT for different values of the corresponding load and resource type. The Benchmark resource defines possible load and resource types by a name and a set of patchers. Patchers are functions, integrated into Theodolite, which take a value as input and modify a Kubernetes resource according to that value. For example a load that is defined by the amount of different data sources can be described by a patcher that adjusts an environment variable of the load generator, which defines the number of data sources. A resource type defined by the amount of replicas can be defined by a patcher which adjusts the *replica* field of a deployment resource. As Theodolite integrates Apache Kafka as messaging system between load generator and SUT, the Benchmark resource also configures the Kafka topics that are generated by the benchmarking tool.

The Execution resource refers to a Benchmark resource by its name. It selects one load and one resource type from those provided by the Benchmark and specifies a set of values to be evaluated for each type. Theodolite performs isolated experiments for a certain load intensity with a certain amount of resources and checks for each experiment whether SLOs are met [11]. The set of SLOs to be evaluated can also be configured via the execution resource. In Theodolite, SLO checks are realized in self-contained Python containers that analyze Prometheus monitoring data. Furthermore, an Execution describes the experimental setup, for example, the duration per experiment and the number of repetitions. Finally, an Execution allows to make modifications to the SUT defined in the benchmark, for example, to evaluate different configuration or deployment options of a SUT. This is again done with patchers, which in this case are passed a fixed value.

6. Discussion and Conclusions

The Kubernetes Operator Pattern allows to manage application-level configuration as part of the orchestration process. Applied to benchmarking cloud-native applications, it allows managing benchmarks and executions directly via the Kubernetes API. Main benefits of a benchmarking operator are:

- Benchmarking tool, benchmark, and execution are clearly separated.

[2]https://github.com/cau-se/theodolite/blob/master/theodolite/examples/operator/

- Benchmarks are defined according to a strict format and published as descriptive files without including any tooling.
- All experimental setup is bundled in one file describing the execution of a benchmark.
- Benchmarks and Executions are created in a purely declarative way, saving Benchmark Designer and Benchmarker from implementing any Kubernetes-specific error handling.
- Benchmarks and Executions can be managed via default Kubernetes tools (e.g., *kubectl*), providing a rich user experience without additional implementation effort.
- Since their CRDs can have an OpenAPI schema, Benchmarks and Executions can directly be validated by the Kubernetes API.

All in all, we expect building a benchmark tool with the operator pattern facilitates repeatability and, hence, ultimately reproducibility of cloud-native benchmarking studies.

References

[1] Cloud Native Computing Foundation, CNCF cloud native definition v1.0, 2018. URL: https://github.com/cncf/toc/blob/main/DEFINITION.md.

[2] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović, A. van Hoorn, Microservices: A performance tester's dream or nightmare?, in: International Conference on Performance Engineering, 2020. doi:10.1145/3358960.3379124.

[3] M. Grambow, E. Wittern, D. Bermbach, Benchmarking the performance of microservice applications, SIGAPP Appl. Comput. Rev. 20 (2020). doi:10.1145/3429204.3429206.

[4] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, P. Cao, How to build a benchmark, in: International Conference on Performance Engineering, 2015. doi:10.1145/2668930.2688819.

[5] W. Hasselbring, Benchmarking as empirical standard in software engineering research, in: Evaluation and Assessment in Software Engineering, 2021. doi:https://doi.org/10.1145/3463274.3463361.

[6] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. Von Kistowski, A. Ali-eldin, C. Abad, J. N. Amaral, P. Tůma, A. Iosup, Methodological principles for reproducible performance evaluation in cloud computing, IEEE Transactions on Software Engineering 47 (2021). doi:10.1109/TSE.2019.2927908.

[7] B. Ibryam, R. Huss, Kubernetes Patterns: Reusable Elements for Designing Cloud Native Applications, 1st ed., O'Reilly, 2019.

[8] S. Kounev, K.-D. Lange, J. von Kistowski, Systems Benchmarking: For Scientists and Engineers, 1st ed., Springer, 2020.

[9] D. Bermbach, E. Wittern, S. Tai, Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective, 1st ed., Springer, 2017.

[10] S. Henning, W. Hasselbring, Theodolite: Scalability benchmarking of distributed stream processing engines in microservice architectures, Big Data Research 25 (2021). doi:https://doi.org/10.1016/j.bdr.2021.100209.

[11] S. Henning, W. Hasselbring, How to measure scalability of distributed stream processing engines?, in: International Conference on Performance Engineering, 2021. doi:https://doi.org/10.1145/3447545.3451190.