Cloud-Native-Bench: an Extensible Benchmarking Framework to Streamline Cloud Performance Tests

Michiel Van Kenhove, Merlijn Sebrechts, Filip De Turck and Bruno Volckaert
IDLab, Department of Information Technology
Ghent University - imec, Ghent, Belgium
michiel.vankenhove@ugent.be

Abstract—**The shift to the cloud among organizations has surged enormously over the last decade. As a result, an overwhelming amount of new cloud-based technologies emerged, making it increasingly more challenging to compare the different technologies and to identify the ideal technology that aligns best with a specific use case. The performance and resource usage of a system or software technology can be assessed by running benchmarks. Performing benchmarks has proven to be a time-consuming and error-prone task, especially when executing multiple consecutive tests on the same system. This sparked curiosity in exploring the feasibility of automating this manual benchmarking process. This paper proposes Cloud-Native-Bench (CNB), a novel open source benchmarking framework implemented as a Kubernetes operator that fully automates the benchmarking cycle. The entire process, including benchmark deployment, the consecutive execution of benchmarks in a queue, results collection, and statistical data analysis, is fully automated. The framework is designed to be extensible without the need to adapt the operator itself, enabling users to develop fine-tuned custom benchmarks according to their specific use cases. A detailed evaluation shows the ease-of-use of Cloud-Native-Bench and how it streamlines the process of running benchmarks in cloud-native environments. Experiments show the importance of running benchmarks on cloud technologies. For example, employing a different web server technology can increase the mean throughput by 252%.**

Index Terms—**Kubernetes, benchmarking, operators, controllers, automated analysis**

I. Introduction

Over the last decade, there has been a significant shift among organizations towards adopting cloud-based solutions [1], resulting in a remarkable growth of cloud infrastructure. This shift has led to the development of many cloud technologies, like application containerization and cloud orchestration platforms such as Kubernetes [2], with the goal to improve the reliability of cloud-based solutions, while trying to keep costs to a minimum. Today, the adoption of containerization technologies to host applications has become mainstream, as acknowledged by the Cloud Native Computing Foundation (CNCF) [3].

In the current cloud-native landscape, a lot of technologies and tools are widely used and new technologies are emerging constantly [4]. Every project or technology tries to solve certain problems, e.g., throughput, resource optimization, resilience, etc., and there is always some overlap between different projects in one way or another. With this large landscape of available technologies, all having their own features and versioning system, it can become a tedious and overwhelming task to determine when one should use a certain technology or version over another.

To optimize their profit, organizations need to carefully determine what their cloud needs are, because an incorrect decision in cloud technologies can lead to immense costs, and even worse, the loss of unhappy customers. Once an organization has determined their needs, they need to map these needs to available technologies and perform a comparative analysis to make a final decision on what technologies and projects to use. Besides choosing a technology stack when first moving to the cloud, newly emerged technologies or software versions need to be compared to the existing cloud solution using an easy and efficient method. This involves running benchmarks according to specific business needs, for example optimizing the throughput of a system.

Running benchmarks is typically a time-consuming task, that involves active monitoring of a currently running benchmark, gathering the results by downloading the data, and finally, starting the next benchmark. Afterwards, the gathered data needs to be parsed and analyzed, often by generating charts that allow technology or software version comparison.

Although there has been a vast amount of research regarding the performance and resource usage of different cloud technologies, only a limited amount of research regarding tools that fully automate the process of benchmarking cloud systems has been conducted. For example, *CloudBench* by Silva et al. [5] and *Smart CloudBench* by Chhetri et al. [6] deploy applications on native cloud resources, instead of in a containerized environment, and still need manual active monitoring of each benchmark run. Other state-of-the-art solutions are either not easily extensible or they do not fully automate the complete benchmarking and analysis process.

This research paper proposes an open source[1] [7] extensible benchmarking framework that aims to answer the following research questions:

RQ 1: How can the complete lifecycle of containerized cloud benchmarks be automated in Kubernetes?

RQ 2: How can an automated benchmarking system be dynamically extensible to enable users to define and run custom benchmarks that are fine-tuned to their specific business needs?

[1]https://github.com/idlab-discover/Cloud-Native-Bench

RQ 3: How to facilitate the comparison of benchmark results with the wide variety of different benchmark output formats?

RQ 4: How much improvement in productivity does the proposed framework bring compared to the traditional way of running benchmarks?

The remainder of this paper is organized as follows. Section II is a study on the state-of-the-art related to cloud benchmarking, while section III goes over the required system functionalities of Cloud-Native-Bench. Section IV presents the design and implementation of the framework. The ease-of-use and effectiveness of the framework is evaluated in section V, while section VI discusses the current limitations and how Cloud-Native-Bench can be improved in the future. Finally, the paper is concluded in section VII.

II. Related Work

In this section, several state-of-the-art benchmark automation tools are discussed. These tools are all designed to work in containerized cloud environments. This shows that a considerable amount of progress has been made over the years, but there still lacks a modern solution to automatically run repeatable benchmarks to evaluate different technologies based on specific business needs.

Henning et al. [8], [9] created Theodolite [10], a framework for benchmarking the scalability of distributed stream processing engines in microservices. A strength of Theodolite is that it is implemented as a Kubernetes operator, a modern solution to automate complex tasks in Kubernetes. Theodolite automates the process of deploying a system under test (SUT), generating load on that SUT, and collecting performance metrics during the benchmark execution. While Theodolite provides a good framework that focuses on benchmarking the scalability of cloud-native applications in Kubernetes, it does not allow for custom benchmarks that can be fine-tuned to a specific use case.

Benchmark-operator [11] by cloud-bulldozer, that was previously known as Ripsaw, is an Ansible [12] based Kubernetes operator to deploy common workloads in a Kubernetes cluster to establish a performance baseline. Benchmark-operator only supports a limited, but common, list of built-in workloads. A custom benchmark can only be added by developing a new Ansible role and rebuilding the operator. Due to this approach, benchmarks are not generic and every benchmark has implementation-specific configuration that cannot be changed without rebuilding the operator. In addition, benchmark-operator does not analyze the results by default. However, benchmark-wrapper [13] is another project maintained by cloud-bulldozer that provides a more convenient way to launch benchmarks, gather the data, and perform a detailed statistical analysis on the results, but it still requires manual monitoring of the running benchmark.

Yoshimura et al. [14] created ImageJockey, a test framework to continuously evaluate the performance of container images. The primary goal of ImageJockey is to gain insight into the impact of different container image types on the performance of an application. Yoshimura et al. concluded, among other things, that the latest available container image does not guarantee the best performance and that the functional portability of container images is not equivalent to performance portability, thus reiterating the importance of container application benchmarking. Unfortunately, ImageJockey is not implemented as a Kubernetes operator, but instead as a standalone service that periodically checks for image updates, triggering an evaluation cycle. This makes it less capable for benchmarking modern microservice-based cloud applications that are often running in a cloud cluster managed by Kubernetes. Lastly, ImageJockey seems to be unavailable either in open source format or as a paid-for license at the time of writing, thereby preventing that the tool can be used by third parties.

Finally, Kubestone [15] by Xridge is a Kubernetes operator that automates the deployment of benchmarks. However, Kubestone does not provide a way to easily deploy custom benchmarks. To run custom benchmarks, an adaptation of the operator itself is required, by means of implementing a custom controller. Kubestone does not aggregate the data nor perform automated analysis: the benchmarker is required to manually retrieve the logs of a completed benchmark and perform manual data cleaning in order to analyze the results. Although Kubestone eases the task of running benchmarks, it does not deliver in terms of extensibility and automated analysis. Lastly, the websites of both Kubestone and Xridge seem offline at the time of writing. As a result, the documentation has become inaccessible. Based on the GitHub history and the fact that the websites are inaccessible, there are indications that Kubestone is no longer being actively maintained.

III. Requirements

Executing complex benchmarks typically requires the setup of multiple components that interact with each other. For example, consider a scenario consisting of a load generator, multiple systems that process a certain amount of generated load, and a load balancer. This setup allows an evaluation of the effectiveness of a load balancer that distributes a certain amount of generated load, and the performance of the various load processing systems. The traditional method to evaluate this scenario involves manual deployment and active monitoring of each component. The first step in deploying this scenario to a Kubernetes cloud environment is the containerization of the workloads and all the related components. The next step consists of writing Kubernetes manifest files that define each component in a declarative way. Whenever the benchmark needs to be executed, the benchmarker manually deploys each component by communicating with the Kubernetes API. This is accomplished by providing the Kubernetes API with the corresponding manifest files, usually by using kubectl [16]. Kubectl is a command-line tool that allows an administrator to invoke commands on the Kubernetes API, to adapt the configuration or to retrieve information about certain components.

To streamline the benchmark process, there is a need for an automated benchmark framework that manages the com-

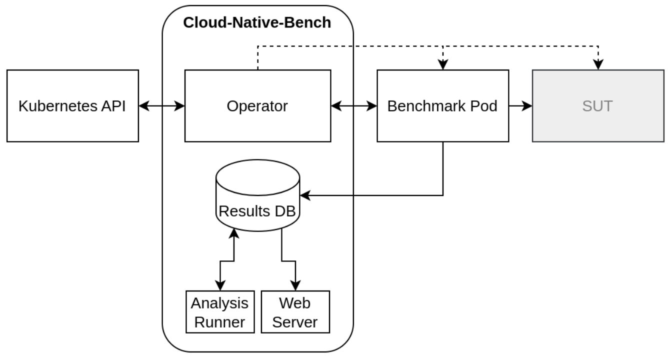


Fig. 1. System architecture overview of Cloud-Native-Bench.

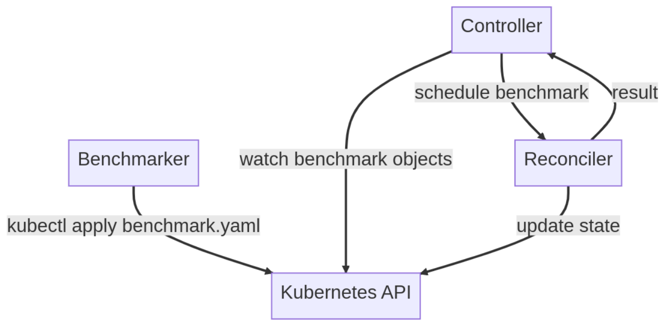


Fig. 2. Kubernetes control loop for Cloud-Native-Bench's operator. Based on [21].

plete benchmarking lifecycle. This system needs to be highly and dynamically extensible, enabling users to define custom benchmarks that are fine-tuned to their specific business needs. Moreover, the system needs to be able to manage a large set of benchmarks and automatically execute them consecutively by means of a queuing system. The resulting data from each benchmark needs to be persisted in a standardized format and trigger an automated analysis system to gather initial statistical insight, for example, by generating easily comparable charts. Finally, the benchmarker needs to have straightforward access to the data and analysis results.

IV. Design and Implementation

This section goes over the design choices and implementation details of Cloud-Native-Bench. First, an overview of the system architecture is given, followed by an introduction to the *Benchmark* custom resource. Implementation specific details of the system are provided. Finally, a detailed introduction to the operation of a Kubernetes operator and how Cloud-Native-Bench employs such an operator is described.

A. Architecture Overview

The architecture of Cloud-Native-Bench is depicted in Fig. 1 and consists of four main components, i.e., the *Operator* that is responsible for managing benchmarks, the *Results Database* for persisting benchmark results, an *Analysis Runner* to automatically perform a statistical analysis on the results of a finished benchmark, and a *Web Server* to provide access to benchmark results and their associated analysis. The architecture in Fig. 1 also illustrates a *Benchmark Pod* that is deployed and managed by the operator. A benchmark pod can be a workload that performs a standalone benchmark test, or perform tests, e.g., as a virtual user, on a segregated *System Under Test* (SUT), that is also fully managed by the operator.

B. Benchmark Custom Resource

A custom resource (CR) [17] is an extension to the Kubernetes API that describes a custom Kubernetes object. A benchmark is defined in a *Benchmark* custom resource, that describes the benchmark in a declarative way. The benchmark CR contains a benchmark type, i.e., *System*, *Network* or *Custom*, and a list of all workloads and components involved in the benchmark. A workload or dependent component can be defined as a *podTemplateSpec*, the Kubernetes built-in resource that describes a pod, or a *helmSpec*, a custom specification that describes a Helm [18] chart. The ability to deploy a workload or a SUT by means of a Helm chart allows for the automated performance testing of complex microservice-based systems. The benchmark CR also consists of a status, storing the current state of the benchmark, i.e., *Pending*, *Running*, *Done* or *Completed*, and the current queue position of the benchmark.

C. Kubernetes Operator

To automate the deployment and management process of benchmark workload components, a Kubernetes operator [19] is implemented. An operator is a software extension to Kubernetes that makes use of custom resources to manage applications and components, and employs a control loop [20] to adhere to Kubernetes principles. Fig. 2 depicts a high-level overview of Cloud-Native-Bench's control loop. The controller of an operator is a long-running program that continuously watches the desired state of a Kubernetes object, and compares it to the effective state of the object in the system. If the desired state deviates from the effective state, the operator schedules a reconciliation to steer the effective state in the direction of the desired state, using idempotent and atomic operations. If a reconciliation is unsuccessful, the reconciliation is retried or requeued, so the desired state should eventually be obtained, following the *eventual consistency* principle.

For example, when a new benchmark should be deployed, the benchmarker creates a benchmark CR object by interacting with the Kubernetes API. The controller notices the newly created benchmark object and will assign the next queue position to the benchmark CR. When a benchmark CR becomes first in queue, the controller starts a reconciliation to create all the necessary components of that benchmark. The reconciler then interacts with the Kubernetes API to update the system state,

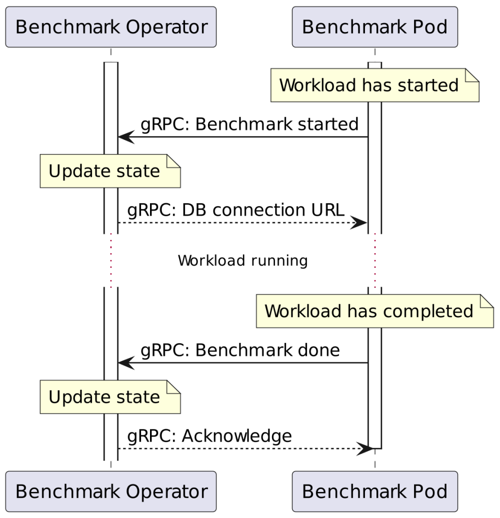


Fig. 3. Communication between operator and benchmark pod.

so the desired state can eventually be achieved. This effectively automates the manual process of deploying and monitoring benchmarks.

The decision was made to develop the operator using the Rust [22] programming language as opposed to the Go [23] programming language that is commonly used for Kubernetes operators. Research shows that operators written in Rust can reduce memory usage by 56.06% compared to the same operator logic written in Go [24]. Rust is a highly performant programming language without runtime or garbage collector. Additionally, it is an inherent memory- and thread-safe language, due to the rich type system and ownership model. Instead of writing the communication with the Kubernetes API from the ground up, the *kube* and *k8s-openapi* Rust crates are used.

D. Communication Between Operator and Workload

The operator needs to be informed about the latest state of the deployed benchmark workload to take actions according to different states. The communication between the operator and the workload is realized using gRPC [25]. gRPC is an open source high performance remote procedure call (RPC) framework used for inter-process or inter-service communication, without environmental restrictions. The communication protocol schema is defined using Protocol Buffers [26] and is easily adaptable. The communication between the operator and a benchmark workload is shown in Fig. 3. Once the benchmark workload pod has been deployed and enters the effective running state, the workload sends a request to the operator, informing that it is ready to perform the benchmark. The operator will update the benchmark CR state from *Pending* to *Running* and provide the workload with the result database connection URL, that it can later use to store the results. Once the workload has completed and the results are stored in the database, the workload sends a request to the operator, that in turn updates the CR state to *Done*, and the operator responds with an acknowledge message. The operator performs a clean-up procedure by removing all the components associated with the benchmark, after which the CR state becomes *Completed*.

E. Storage

The benchmark results and the statistical analysis on these results are stored in a PostgreSQL [27] database. PostgreSQL is chosen due to its open source nature and broad feature set. An SQL scheme is defined and the output of a benchmark is parsed and transformed by the benchmark workload before storing. This design choice is made to ensure that the system remains highly extensible, creating the opportunity that custom benchmarks can be executed without the need to adapt Cloud-Native-Bench in any way. Without this design choice, each additional custom benchmark would require an adaptation of the components that are responsible for parsing and transforming the output to the standardized format.

F. Analysis Runner

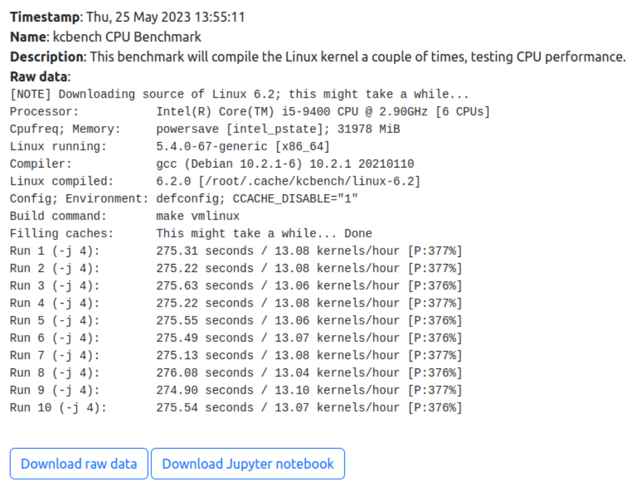
An important component of Cloud-Native-Bench is the analysis runner. A comprehensive data analysis allows the benchmarker to compare multiple benchmark results with each other. Using a more traditional benchmarking method, gaining insight into the results of finalized benchmarks would typically require the benchmarker to manually gather all benchmark results in order to perform a manual data analysis. This is a repetitive and time-consuming task, that also can be error-prone, for example, by accidentally interchanging multiple benchmark outputs. The analysis runner fully automates this process at the end of each benchmarking cycle by periodically checking the results database for newly completed benchmarks. When a newly completed benchmark is found, the benchmark results are fetched and a Jupyter [28] notebook is generated. This notebook contains the raw data of the benchmark, as well as multiple statistical calculations, e.g., standard deviation and mean, and distribution charts, e.g., histograms and probability density function estimations using kernel smoothing, that ease the comparison between benchmark results. As the final step in each benchmarking cycle, the generated Jupyter notebook is stored in the results database by the analysis runner.

G. Web Interface

To facilitate access to the benchmarking results by the benchmarker, a web interface is created. This web interface consists of a list of all completed benchmarks, and a dedicated details page for each benchmark. An example of such a benchmark details page is depicted in Fig. 4 and it allows the benchmarker to download the benchmark's raw data, as well as the generated Jupyter notebook.

V. Evaluation Results

This section shows the ease-of-use of Cloud-Native-Bench and how its automated benchmarking workflow compares to a manual benchmarking workflow. The resource usage of the system is evaluated and a demonstration shows the necessity of performing benchmarks in cloud environments.



Timestamp: Thu, 25 May 2023 13:55:11
Name: kcbench CPU Benchmark
Description: This benchmark will compile the Linux kernel a couple of times, testing CPU performance.
Raw data:

```
[NOTE] Downloading source of Linux 6.2; this might take a while...
Processor:           Intel(R) Core(TM) i5-9400 CPU @ 2.90GHz [6 CPUs]
Cpufreq; Memory:     powersave [intel_pstate]; 31978 MiB
Linux running:       5.4.0-67-generic [x86_64]
Compiler:            gcc (Debian 10.2.1-6) 10.2.1 20210110
Linux compiled:      6.2.0 [/root/.cache/kcbench/linux-6.2]
Config; Environment: defconfig; CCACHE_DISABLE="1"
Build command:       make vmlinux
Filling caches:      This might take a while... Done
Run 1 (-j 4):        275.31 seconds / 13.08 kernels/hour [P:377%]
Run 2 (-j 4):        275.22 seconds / 13.08 kernels/hour [P:377%]
Run 3 (-j 4):        275.63 seconds / 13.06 kernels/hour [P:376%]
Run 4 (-j 4):        275.22 seconds / 13.08 kernels/hour [P:377%]
Run 5 (-j 4):        275.55 seconds / 13.06 kernels/hour [P:376%]
Run 6 (-j 4):        275.49 seconds / 13.07 kernels/hour [P:376%]
Run 7 (-j 4):        275.13 seconds / 13.08 kernels/hour [P:377%]
Run 8 (-j 4):        276.08 seconds / 13.04 kernels/hour [P:376%]
Run 9 (-j 4):        274.90 seconds / 13.10 kernels/hour [P:377%]
Run 10 (-j 4):       275.54 seconds / 13.07 kernels/hour [P:376%]
```

Fig. 4. Example of the web interface that displays a detailed overview of a benchmark result.

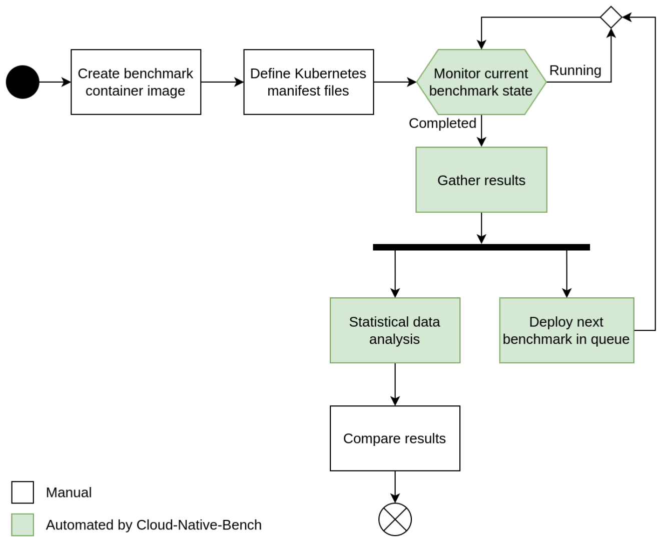

Create benchmark container image
Define Kubernetes manifest files
Monitor current benchmark state
Running
Completed
Gather results
Statistical data analysis
Deploy next benchmark in queue
Compare results
Manual
Automated by Cloud-Native-Bench

Fig. 5. Schematic overview of a typical benchmarking workflow. Cloud-Native-Bench automates four out of the seven activities involved in the process of executing benchmarks.

A. *Benchmarking Workflow Evaluation*

A typical benchmarking workflow is depicted in Fig. 5 and shows the activities that are fully automated by Cloud-Native-Bench in green. After a benchmarker develops a benchmark, it needs to be containerized. To be able to deploy the benchmark using the traditional manual method, all the different Kubernetes manifest files, for each component involved in the benchmark, need to be defined. Using Cloud-Native-Bench, only one *Benchmark* custom resource manifest file is necessary. An example of such a benchmark custom resource manifest file is given in Fig. 6. Prior to deploying the benchmark in the manual setting, the benchmarker should verify that no other benchmarks are currently running, to prevent that the next benchmark is started before the current benchmark is

```yaml
apiVersion: michiel.van.kenhove.ugent.be/v1
kind: Benchmark
metadata:
  name: cnb-example
  namespace: cnb-run
spec:
  title: Example
  benchmarkType: Custom
  workloads:
    - podTemplate:
        metadata:
          namespace: cnb-run
          labels:
            benchmark: cnb-example
        spec:
          containers:
            - name: cnb-example-virutal-user
              image: cnb-example-virtual-user
              env:
                - name: OPERATOR_GRPC_ADDRESS
                  value: "http://operator.cnb-system.svc.cluster.local:50051"
                - name: ITERATIONS
                  value: "100"
                - name: ...
                  value: "..."
    - helmChart:
        repositoryUrl: https://charts.bitnami.com/bitnami
        chartReference: mongodb
```

Fig. 6. An example of a benchmark custom resource manifest file. This example will deploy a “virtual user” pod, simulating user behavior by performing certain actions on a SUT.

completed. Running two or more benchmarks simultaneously can cause incorrect or misleading results, due to the influence that benchmarks may have on each other, and should therefore be avoided. Cloud-Native-Bench monitors the current running benchmark’s state and automates the consecutive execution of benchmarks through the use of a queuing system. New benchmarks are placed at the end of the queue. The time-consuming task of collecting all the results and a statistical analysis is also automated. Cloud-Native-Bench proves to streamline the traditional benchmarking workflow by automating all time-consuming and error-prone tasks.

B. *Resource Overhead*

To ensure the wide applicability of Cloud-Native-Bench, particularly in low-resource environments such as edge computing, the idle resource usage of each component is shown in Table I. The total idle memory footprint of the system is 88 MiB with a negligible idle CPU usage. It shows that the components written in Rust have the least amount of overhead, with an idle memory usage of 6MiB and 2MiB for the operator and web server respectively. The complete Cloud-Native-Bench system proves to have no significant resource usage overhead.

TABLE I
IDLE RESOURCE USAGE OF CLOUD-NATIVE-BENCH’S COMPONENTS

	Technology	Memory (MiB)	CPU (milliCPU)
Operator	Rust (kube)	6	1
Result DB	PostgreSQL	27	3
Web server	Rust (axum)	2	0
Analysis runner	Python	53	1

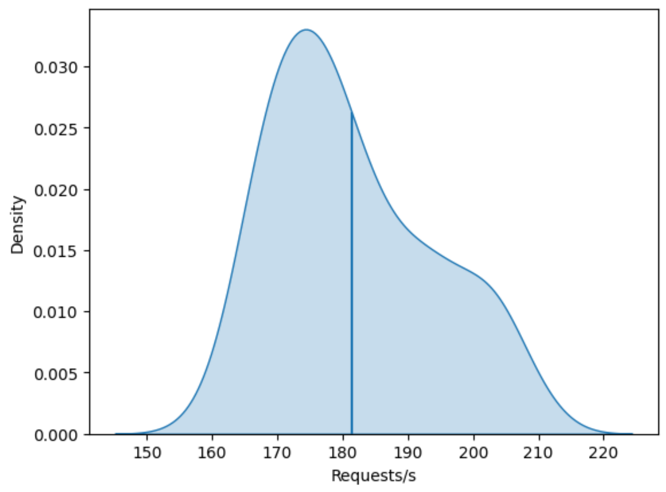


Fig. 7. KDE chart that estimates the PDF of the throughput for httpd:2.4.57, with a resource limit of 512MiB of memory and 75 milliCPU. The vertical line indicates the mean throughput. This chart is one of the automatically generated Jupyter notebook charts by the analysis runner.

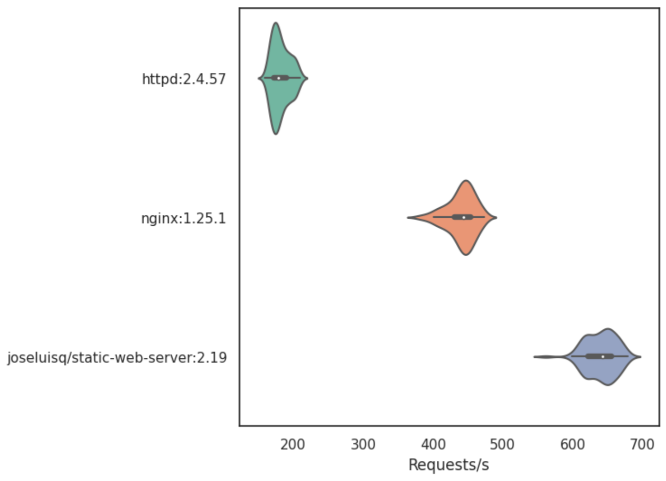


Fig. 8. Comparison of the throughput of three HTTP web servers. All three web servers have a limit of 512MiB of memory and 75 milliCPU.

Although the resource usage of Cloud-Native-Bench is negligible, it is still advisable to run the operator on a separate dedicated node to prevent any influence on the benchmark results. In scenarios where benchmarks need to be performed on highly constrained low-resource environments, such as edge computing clusters, Cloud-Native-Bench has the capability to be deployed on separate hardware to interact with the Kubernetes API, rather than being deployed within the cluster itself. Furthermore, it is possible to deploy certain components in the cluster while deploying others to dedicated machines.

C. Demonstration

To demonstrate the usability of Cloud-Native-Bench, the throughput of three different HTTP web server technologies are benchmarked. Gaining insight into the throughput of different web server technologies enables the ability to choose the most suitable web server technology for a specific use case, dependent on the resource availability and environment. The first tested web server is *httpd*, the Apache HTTP server [29], that is being used by 31.5% [30] of the publicly available websites as of writing. *NGINX* [31], the second tested web server, is currently the most wildly adopted web server technology, used by 34.2% [32] of all websites. The last tested web server, *Static Web Server* [33], developed by Jose Quintana in Rust [22], is a project that has significantly less popularity compared to httpd or NGINX, due to its more limited functionality. Static Web Server can be considered as a use-case-specific web server as it is designed to be extremely lightweight and easy-to-use.

Multiple tools are available to test the throughput of a web server, for example *wrk* [34] and *ab* [35], where the latter is also known as ApacheBench. ApacheBench only supports HTTP/1.0 and consequently requires a separate connection for each request, adding a large amount of overhead. This limitation is not present in wrk, and is therefore the chosen load generator used in this demonstration.

To test the throughput of each web server, three container images, one for each web server, are built. The base images used to build the three web servers are *httpd:2.4.57*, *nginx:1.25.1*, and *joseluisq/static-web-server:2.19*. Each web server uses their respective default configuration and hosts an identical 1KiB static HTML file. To generate HTTP load on the web servers, wrk uses ten concurrent connections spread over two threads. To discover and compare the behavior of the web servers under different environmental constraints, the set of benchmarks is repeated under different memory and CPU limits, by using Kubernetes resource limits. Each load test runs for one minute and is repeated 100 times to be statistical representative. The total runtime of all benchmarks combined was ten hours, where a new benchmark needed to be started every 100 minutes. This process was fully automated by Cloud-Native-Bench without any human intervention. This proves that the complete benchmarking process is streamlined by using Cloud-Native-Bench.

Fig. 7 shows one of the charts that was automatically generated by the analysis runner, and depicts a kernel density estimation (KDE) chart that estimates the probability density function (PDF) of the throughput of httpd:2.4.57 with resource restrictions of 512MiB of memory and 75 milliCPU. The mean throughput under these conditions is about 181 requests per second.

A throughput comparison of the three web servers, with their respective throughput distribution, is shown in Fig. 8. The mean throughput for httpd, NGINX and Static Web Server is about 181.5, 439.8 and 639.6 requests per second respectively. These results show that using NGINX over httpd in this particular use case improves the mean throughput by about 142%, and using Static Web Server even improves the mean throughput by about 252%. Using Static Web Server would thus require significantly less cloud resources for the same amount of traffic and can therefore drastically decrease the cost of hosting an application. This is proven by Fig. 9: it

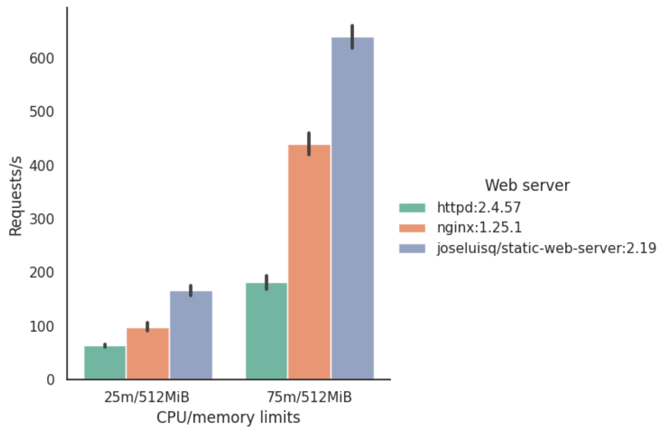


Fig. 9. Comparison of the throughput of three HTTP web servers when the CPU usage limit is increased from 25 milliCPU to 75 milliCPU.

shows that Static Web Server can provide about the same throughput as httpd, with 1/3 of allocated CPU. These results are especially insightful to choose a suitable web server technology on highly constrained devices, such as cloud edge nodes.

VI. Future Work

Despite Cloud-Native-Bench being an effective system that addresses a lot of the traditional benchmarking problems, the system can always be improved in the future. Below is a list of features that are to be investigated to improve the overall usability of Cloud-Native-Bench.

- Real-time resource usage: the system can be extended with Prometheus [36] and Grafana [37] as a supplementary data storage and visualization tool, to extend the data analysis possibilities and to get real-time resource usage insights.
- Tracing tools: additional tracing tools or frameworks can support the benchmarker in gaining insight into benchmark results and discovering bottlenecks in large scale applications. It should be investigated which tracing tools are useful in the context of Cloud-Native-Bench and how such tracing tools can be integrated with the system.
- Failed benchmark diagnostics: the system currently assumes that benchmarks will not fail. This assumption is fine in most scenarios, but there is always the possibility that a certain benchmark fails. The system currently ignores these failed benchmarks and starts the next benchmark in the queue, without informing the benchmarker that something went wrong. In the future it can be useful to retry failed benchmarks and provide the benchmarker with adequate diagnostics to debug the issue when repeated failures occur beyond a specified number of attempts.
- Priority based queue: the benchmark execution queue is currently implemented as a First-In-First-Out (FIFO) queue. As a result, it is only possible to influence the execution sequence by applying the benchmark CRs to the Kubernetes API in the order the benchmarks should be executed. This can be solved by introducing a priority based queuing system that allows the benchmarker to prioritize certain benchmarks over others.

This paper proposes Cloud-Native-Bench with the main focus being on the requirements, design and prototype implementation of the system itself. In the future, the framework will be used for large-scale comparative studies on different cloud-native technologies and systems.

VII. Conclusion

Running a large set of benchmarks is typically a time-consuming task, that involves manual interventions and active monitoring of running benchmarks. In this paper, an open source extensible benchmarking framework, Cloud-Native-Bench, is proposed. This framework is implemented as a Kubernetes operator and completely automates the process of running benchmarks, gathering its data, and performing statistical data analysis to enable the comparison of different benchmarks. Deploying and configuring benchmarks is done in a declarative way, and a queuing system manages consecutive execution of benchmarks without the need for human interaction during the complete benchmark execution cycle.

Cloud-Native-Bench is a highly extensible benchmarking system, that enables users to run standardized and custom benchmarks that are fine-tuned to their specific business needs and that fully integrate with the system without the need to adapt the system in any way. The automated data analysis greatly reduces the time spent to gather and parse different benchmark outputs, and enables the benchmarker to gain useful insights more swiftly.

The proposed system proves to have a negligible amount of resource overhead in most scenarios, with a total idle memory consumption of 88 MiB. This result is achieved by using the Rust programming language instead of the Go programming language that is more typically used to build Kubernetes operators.

A set of tests show the ease-of-use of Cloud-Native-Bench and the necessity of running benchmarks in cloud environments. Using the Rust based *Static Web Server*, developed by Jose Quintana, proves to improve the mean throughput by about 252% compared to *httpd*, the Apache HTTP server.

Acknowledgment

This work was partially supported by the Flemish Government through FWO SBO (Strategic Basic Research) project MOZAIK S003321N.

References

[1] “Gartner Forecasts Worldwide Public Cloud End-User Spending 2023.” [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2022-10-31-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-600-billion-in-2023
[2] “Kubernetes.” [Online]. Available: https://kubernetes.io/
[3] “CNCF Annual Survey 2022 — Cloud Native Computing Foundation.” [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2022/

[4] “Cloud Native Landscape.” [Online]. Available: https://landscape.cncf.io/

[5] M. Silva, M. R. Hines, D. Gallo, Q. Liu, K. D. Ryu, and D. Da Silva, “CloudBench: Experiment automation for cloud environments,” in *IEEE International Conference on Cloud Engineering, IC2E*, 2013, pp. 302–311.

[6] M. B. Chhetri, S. Chichin, Q. B. Vo, and R. Kowalczyk, “Smart cloudbench-Automated performance benchmarking of the cloud,” in *IEEE International Conference on Cloud Computing, CLOUD*, 2013, pp. 414–421.

[7] M. Van Kenhove, “idlab-discover/Cloud-Native-Bench: v0.1,” 6 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8101912

[8] S. Henning, B. Wetzel, and W. Hasselbring, “Reproducible Benchmarking of Cloud-Native Applications With the Kubernetes Operator Pattern,” in *Symposium on Software Performance, SSP*, 2021.

[9] S. Henning and W. Hasselbring, “Demo Paper: Benchmarking Scalability of Cloud-Native Applications with Theodolite,” in *IEEE International Conference on Cloud Engineering, IC2E*. Institute of Electrical and Electronics Engineers Inc., 2022, pp. 275–276.

[10] “Theodolite.” [Online]. Available: https://www.theodolite.rocks/

[11] “cloud-bulldozer/benchmark-operator: The Chuck Norris of cloud benchmarks.” [Online]. Available: https://github.com/cloud-bulldozer/benchmark-operator

[12] “Ansible is Simple IT Automation.” [Online]. Available: https://www.ansible.com/

[13] “cloud-bulldozer/benchmark-wrapper: Python Library to run benchmarks.” [Online]. Available: https://github.com/cloud-bulldozer/benchmark-wrapper

[14] T. Yoshimura, R. Nakazawa, and T. Chiba, “ImageJockey: A framework for container performance engineering,” in *IEEE International Conference on Cloud Computing, CLOUD*, vol. 2020-October. IEEE Computer Society, 10 2020, pp. 238–247.

[15] “kubestone/kubestone: Performance benchmarks for Kubernetes.” [Online]. Available: https://github.com/kubestone/kubestone

[16] “Command line tool (kubectl) Kubernetes.” [Online]. Available: https://kubernetes.io/docs/reference/kubectl/

[17] “Custom Resources Kubernetes.” [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[18] “Helm.” [Online]. Available: https://helm.sh/

[19] “Operator pattern Kubernetes.” [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

[20] “Controllers Kubernetes.” [Online]. Available: https://kubernetes.io/docs/concepts/architecture/controller/

[21] “kube.” [Online]. Available: https://kube.rs/

[22] “Rust Programming Language.” [Online]. Available: https://www.rust-lang.org/

[23] “The Go Programming Language.” [Online]. Available: https://go.dev/

[24] M. Sebrechts, T. Ramlot, S. Borny, T. Goethals, B. Volckaert, and F. De Turck, “Adapting Kubernetes controllers to the edge: on-demand control planes using Wasm and WASI,” in *Proceedings of the 2022 IEEE Conference on Cloud Networking, CloudNet*. Institute of Electrical and Electronics Engineers Inc., 2022, pp. 195–202.

[25] “gRPC.” [Online]. Available: https://grpc.io/

[26] “Protocol Buffers Documentation.” [Online]. Available: https://protobuf.dev/

[27] “PostgreSQL: The world’s most advanced open source database.” [Online]. Available: https://www.postgresql.org/

[28] “Project Jupyter.” [Online]. Available: https://jupyter.org/

[29] R. T. Fielding and G. Kaiser, “The Apache HTTP Server Project,” *IEEE Internet Computing*, vol. 1, no. 4, pp. 88–90, 1997.

[30] “Usage Statistics and Market Share of Apache.” [Online]. Available: https://w3techs.com/technologies/details/ws-apache

[31] “NGINX - Advanced Load Balancer, Web Server, & Reverse Proxy.” [Online]. Available: https://www.nginx.com/

[32] “Usage Statistics and Market Share of Nginx.” [Online]. Available: https://w3techs.com/technologies/details/ws-nginx

[33] “static-web-server: A cross-platform, high-performance and asynchronous web server for static files-serving.” [Online]. Available: https://github.com/static-web-server/static-web-server

[34] “wg/wrk: Modern HTTP benchmarking tool.” [Online]. Available: https://github.com/wg/wrk

[35] “ab - Apache HTTP server benchmarking tool.” [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html

[36] “Prometheus - Monitoring system and time series database.” [Online]. Available: https://prometheus.io/

[37] “Grafana: The open observability platform — Grafana Labs.” [Online]. Available: https://grafana.com/