Beauty and the Beast: A Case Study on Performance Prototyping of Data-Intensive Containerized Cloud Applications

Floriment Klinaku
University of Stuttgart
Stuttgart, Germany
klinaku@iste.uni-stuttgart.de

Martina Rapp
Jörg Henss
FZI Forschungszentrum Informatik
Karlsruhe, Germany
{rapp,henss}@fzi.de

Stephan Rhode
Robert Bosch GmbH
Renningen, Germany
stephan.rhode@de.bosch.com

ABSTRACT

Data-intensive container-based cloud applications have become popular with the increased use cases in the Internet of Things domain. Challenges arise when engineering such applications to meet quality requirements, both classical ones like performance and emerging ones like resilience. There is a lack of reference use cases, applications, and experiences when prototyping such applications that could benefit the research community. Moreover, it is hard to generate realistic and reliable workloads that exercise the resources according to a specification. Hence, designing reference applications that would exhibit similar performance behavior in such environments is hard. In this paper, we present a work in progress towards a reference use case and application for data-intensive containerized cloud applications having an industrial motivation. Moreover, to generate reliable CPU workloads we make use of ProtoCom, a well-known library for the generation of resource demands, and report the performance under various quality requirements in a Kubernetes cluster of moderate size. Finally, we present the scalability of the current solution assuming a particular autoscaling policy. Results of the calibration show high variability of the ProtoCom library when executed in a cloud environment. We observe a moderate association between the occupancy of node and the relative variability of execution time.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; *Software architectures*; *Publish-subscribe / event-based architectures*; • **Computer systems organization** → *Cloud computing*.

KEYWORDS

cloud, elasticity, performance prototype, modelling

ACM Reference Format:
Floriment Klinaku, Martina Rapp, Jörg Henss, and Stephan Rhode. 2022. Beauty and the Beast: A Case Study on Performance Prototyping of Data-Intensive Containerized Cloud Applications. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22 Companion), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3491204.3527482

ICPE '22 Companion, April 9–13, 2022, Bejing, China

ACM ISBN 978-1-4503-9159-7/22/04…$15.00
https://doi.org/10.1145/3491204.3527482

1 INTRODUCTION

Containers [20] have become the de-facto standard for packaging and deploying microservice-based applications in the cloud. A particular class of such applications continuously processes streams of data generated by a variety of connected data-sources. The performance of such applications is business-critical. In addition to performance, elasticity and resilience have become two required quality attributes to cost-efficiently handle disruptive events like unexpected failures or changes in the demand.

Scaling non-trivial microservice-based applications remains a challenge for service providers due to the uncertain cloud environment. Achieving elasticity and resilience through an upfront engineering process requires suitable prediction models. There is, however, a lack of reference use cases, applications and experiences matching the characteristics of data-intensive cloud applications that would allow researchers to evaluate their approaches. Two prominent reference applications are proposed to foster research of microservice-based cloud applications: TeaStore [24] and SockShop [4]. Both serve more traditional use cases of classical human-centered request-reply applications. They lack, however, a processing pipeline of continuous data and also do not use asynchronous messaging communication which is very popular in such use cases. In addition, when prototyping such systems, it is hard to generate realistic workloads that utilize the resources according to a given specification (e.g., the time, that operations should consume the CPU).

To tackle the aforementioned problems we present a work in progress towards a reference use case and application for enabling research for data-intensive containerized cloud applications. To generate more reliable CPU workloads and to make the application more predictable in terms of performance, we make use of ProtoCom [8], a library for calibrating and generating CPU demands on various hardware. In addition we present two different scaling strategies for the defined application and present scalability experiments to obtain a first assessment of the capabilities of the application. Moreover, we investigate on the high variability of the load generation approach as our initial results showed high deviations from the expected response times.

The focus of this work is twofold: first, in Section 3, we introduce the reference use case and present performance requirements and

the chosen implementation stack; second, in Section 4, we show the variability of the resource demand generation with ProtoCom in a cloud environment, initial scaling variants for the use case and experiments that show the scalability aspect of our microservice-based implementation. In addition, Section 2 presents related work and finally, Section 5 concludes the paper and provides an outlook on future work.

2 RELATED WORK

Our related work can be divided into two different categories. On the one hand, several benchmarks have been developed as a reference for cloud applications and their performance. On the other hand microbenchmarking has been applied to measure the impact of virtualized environments and to quantify the performance isolation available in those. While we cannot solve these inherent problems, developers must be aware of those effects affecting performance and scalability in virtualized and containerized applications.

In [18] Nikounia et al. introduce the noisy neighbour problem, an effect that can be observed in shared infrastructures where the activity on a neighboring core may lead to performance degradation. They report on performance degradation of up to 16x slowdown in virtualized environments. This is caused by noisy neighbour VMs, overcommitment and hypervisor noise.

In [14] Laaber et al. present their findings on using microbenchmarking to assess the performance impact in virtualized environments. The authors performed several experiments on systems deployed in public cloud environments and report on slowdown effects ranging from 0.003% to > 100%. They state that several repetitions on several VM instances are required to get robust results for microbenchmarks and to detect potential slowdowns.

In [16] Lehrig et al. present experiments conducted with the ProtoCom library in a virtualized cloud environment. They show that ProtoCom is well suited to emulate CPU demands realistically when using a calibration based approach.

In addition to many benchmark and test application like Spring PetClinic [5] or ACME Air [1], several academic case-study systems have been developed in the past for evaluating the performance of cloud and containerised systems: the Tea Store case-study [24] presents a system for studying the performance of microservice-based systems and has been enhanced with resilience and elasticity in [23]. A similar microservices based benchmark using container technologies, the Sock Shop, is presented in [4]. The CloudStore application [15] is a reference application for comparing different cloud providers, cloud service architectures, and assess cloud deployment options. The TrainTicket benchmark [25] in addition focuses on the fault-analysis of microservice-based applications. All four case-study systems have in common that mostly request-response semantics is employed. Asynchronous data-centric communication patterns as typical found in IoT systems are missing.

3 RUNNING EXAMPLE

Before going into the implementation details of the running example in Section 3.3, we will explain the considered use case, its purpose and architecture in Section 3.1, and discuss performance challenges in Section 3.2.

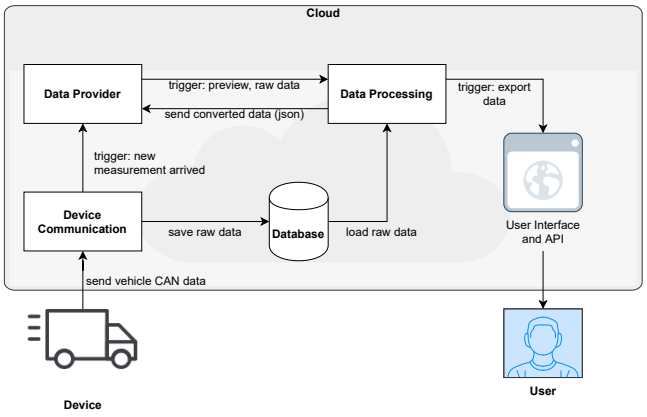


Figure 1: Architectural snippet of remote measuring application in mobility cloud suite.

3.1 Reference Use Case

The herein considered use case – called remote measuring – is a fraction of one service package from the Bosch mobility cloud [10]. The mobility cloud is a cloud-based integration platform for developing and updating vehicle software and services. The services are grouped into three packages: over-the-air services (update, function call, essentials, vehicle data), data services (data integration, navigation, broker), and core services (service integration, application run time, application marketplace). Remote measuring is one service from the over-the-air vehicle data package. We consider parts of remote measuring in the implemented running example and its model representation.

The app remote measuring is designed for vehicle data acquisition campaigns. Such cloud-based campaigns are beneficial in vehicle development, fleet observation (e.g. tracking of failure codes in a delivery car fleet), predictive diagnostics, and optimization of spare part logistics in aftermarket business. Imagine a vehicle homologation task [17], where several vehicles must collect data from test rides. Cloud-based remote measuring allows test engineers to configure and conduct a signal measurement setup through web services. Such a setup contains the number and kind of signals, their recording frequency and recording triggers. The test engineer starts the measurement through a web front end. Data is cached in the car and pushed to the cloud, where the data is stored in a database. Then, the data is converted and presented as dashboard, or exported via an API for external applications. This is more flexible and convenient in contrast to conventional workflows, where each vehicle was equipped with a signal recorder, a laptop, and a test engineer who configured the campaign, stored the data and fed it into a data center afterwards.

Figure 1 explains the architecture of the running example remote measuring. Starting in the lower left corner, one or more devices (vehicles) send data from their vehicle bus (CAN bus [11]) to the device communication service. This service tracks the readiness of the device, caches data, stores the raw data in a data base, and triggers the next service data provider. Once, a new batch of data arrived, the data provider service triggers the data processing service, which conducts data pre-processing and data compression. For

this, the data processing fetches the raw data from data base. This part of the use case is considered and implemented in the running example.

The user interface and API on the right in Figure 1 are not implemented in the running example but shown here to understand the use case from end to end. The data processing triggers an export service, which provides the converted data through API and triggers a data dashboard. The API can be used for customized data analysis on customer side.

3.2 Requirements and Performance Challenges

The remote measuring use case provides several challenges for development and operations in the mobility cloud. These challenges affect elasticity and resilience properties of the application in practice.

In regard to elasticity, the first challenge arises from broad range of configuration options in data acquisition campaigns. These campaigns may differ largely in terms of number of recorded vehicle signals, their record frequency, and the number of devices itself. In vehicle homologation, the number of connected devices is rather small, but the number of considered signals is large. Add to this, the data recording frequency in homologation is large and such a campaign is usually triggered in parallel within a few days. This results into single events where large amount of data is pushed into the cloud from the test vehicles.

In contrast, in fleet observation the number of devices is large, but the number of signals and their recording frequency are small. Due to large number of devices, the accumulated load is large as well, but during fleet observation, we assume that devices connect to the cloud in a rather random and asynchronous profile.

The goal in both usage scenarios is to provide the remote measuring service with acceptable response time for the customer. Hence, remote measuring requires sufficient elasticity to cope with different usage scenarios.

Add to this, the elasticity property of remote measuring determines another service level objective: the cloud costs. While under provisioning causes unacceptable high response times for the users, over provisioning causes high costs, which reduce the revenue of the remote measuring service. Therefore, we search for optimal elasticity property of remote measuring in development and operations.

With respect to resilience, the homologation scenario requires credible data handling to avoid data loss during expensive and elaborate vehicle test rides. Compared with fleet observation, data loss during homologation would cause repetitions of test rides, which can destruct project plans and time to market goals in vehicle development projects. In addition, outage of remote measuring during homologation usage would destroy user trust in the application. Due to this, methods to design and test resilience of the applications are of high importance.

3.3 Performance Prototype/Demonstrator

The remote measuring use case from Figure 1 was re-implemented as performance prototype based on the Spring Boot [1] framework.

[1]https://spring.io/projects/spring-boot

Table 1: Example calibration run output

Time (ms)	Iterations	Time/Iterations
1,00	537389	1,86085E-06
2,00	1172345	1,70598E-06
4,00	2539921	1,57485E-06
7,97	5062500	1,57511E-06
15,85	10060004	1,5753E-06
25,73	16319999	1,57641E-06
63,44	40159726	1,57978E-06
126,68	79983883	1,58383E-06
234,10	148379031	1,57771E-06
541,72	316609902	1,71101E-06
1026,21	630079016	1,6287E-06

The components device communication, data provider, data processing, and database were deployed as containers on an eight node Kubernetes (K8s) [2] cluster on bwCloud [3], a state funded academic cloud. The components use the ProtoCom [4] library to emulate CPU demands.

All components and the database were connected through a RabbitMQ [5] message broker, which runs on a dedicated node on K8s cluster. The database was deployed as MongoDB [6] container. The functionality of the devices was resembled by Gatling [7] load generator. Gatling was used to define load profiles for the system. A load profile consists of the number and the ramp up time of the connecting devices, and frequency and size of sent data. Gatling was deployed as container on a dedicated node in K8s.

Several experiments were conducted with different load profiles. Each experiment was triggered as K8s job and the results from Gatling were stored together with monitoring data from Prometheus [8] for following analysis.

4 PERFORMANCE VARIABILITY OF RESOURCE DEMAND GENERATION

As described previously, to emulate processing of messages in the different services (e.g., the data processing service) each microservice uses ProtoCom. ProtoCom requires a low contention calibration phase to determine the input for a particular algorithm (say Fibbonaci number computations) to put load on the CPU for a given time amount (e.g., consume the CPU for 0.2 CPU-seconds). The results of the calibration are stored in a model as shown in Table 1 which contains the approximated input parameter associated with their individual execution times. Every other resource demand is generated by composing these demands. Since the calibration process consumes time (around 20 minutes for `HIGH` accuracy) and the test-bed cluster is homogeneous we initially thought of pre-calibrating ProtoCom and sharing the calibration for all service replicas. This would allow us to execute elasticity experiments and

[2]https://kubernetes.io/
[3]https://www.bw-cloud.org/
[4]https://sdqweb.ipd.kit.edu/wiki/ProtoCom
[5]https://www.rabbitmq.com/
[6]https://www.mongodb.com/
[7]https://gatling.io/
[8]https://prometheus.io

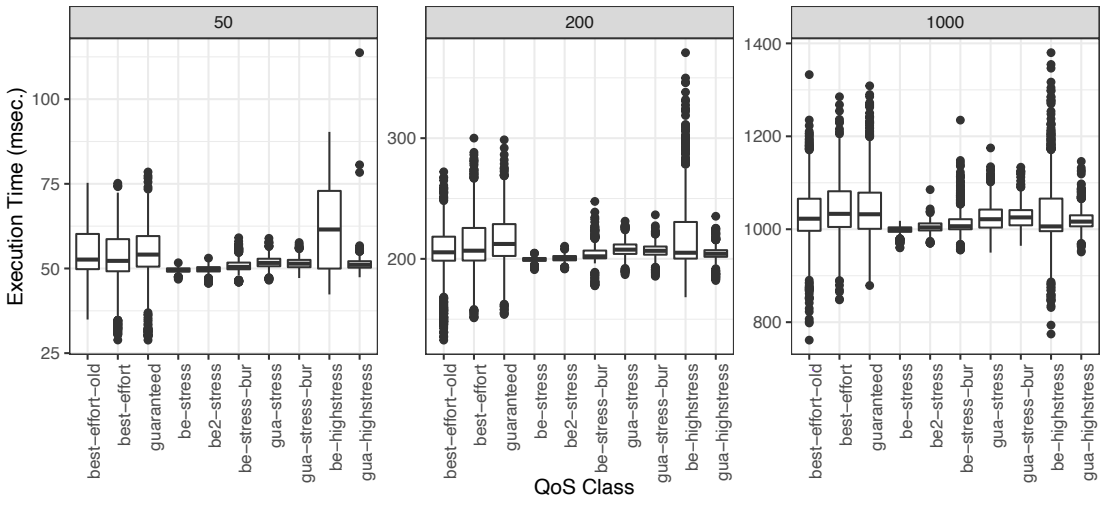


Figure 2: Overall variability of measurements for all instances and executions for the demand parameter set to 50, 200 and 1000 milliseconds for various Kubernetes QoS classes, different load levels and in three cluster compilations: C_I: {*'best-effort-old'*} (1 experiment), C_{II}: {*'best-effort'* and *'guaranteed'*} (2 experiments, after rebooting VMs), C_{III}: {*'be-stress'* – *'gua-highstress'*} (7 experiments, after cloud maintenance).

Table 2: Kubernetes nodes and their occupancy in number of Pods and average utilization in millicores

Name	# Pods	Avg. Millicores	Characteristic App Pods
minion-01	17	207.15	
minion-02	12	805.80	rabbit-broker
minion-03	12	152.10	
minion-04	10	176.85	mongodb
minion-05	8	133.30	
minion-06	7	84.50	
minion-07	11	142.20	demonstrator pods

upon the spin-up of new containers, the calibration would not affect the start-up time. A precondition for this is, that there is an acceptable variability in CPU time across nodes. Hence, we decide to benchmark the resource demand generation library, namely ProtoCom, to determine how it performs in our cluster.

4.1 Environment and Experimentation Setup

The Kubernetes cluster consists of seven worker nodes of identical flavor *m1.large* with `4vCPUs`, `8GB RAM` and `12GB` storage. On the worker nodes there are various numbers of container being deployed where some are application-specific and some come from the platform itself. Table 2 summarizes the number of pods per node together with some characteristic application pods. The average millicores determines the average CPU usage of the cluster when no workload is running. During the benchmark execution the three services of the application—namely, the Device Communication service, the Data Processing service, and the Data Provider service—are co-located on node `minion-07`.

We define as a *compilation* of the cluster the current state of the cluster which changes either by actions taken by us (e.g., after VM reboot) or by the cloud provider (e.g., maintenance work). We execute the benchmark with two different container QoS classes enabled: *best-effort* (no limit, no guaranteed share) and *guaranteed* (limits are equal to guaranteed share). For each class we make five *executions* where in each *execution* five *measurement iterations* follow after an initial five warm-up iterations[9]. The execution happens on all the seven nodes.

In addition to changing the QoS class for the benchmark container, we change the background load running on a VM. We create three classes of background load levels: `LOW`, `MEDIUM`, and `HIGH`. The `LOW` case constitutes the state of the cluster where no additional load is present besides the workload deployed as Figure 2 depicts. For the `MEDIUM` and `HIGH` case, we inject load using Chaos Mesh[10] where for the `MEDIUM` case, we use the configuration with four workers (matching the number of vCPUs) inducing 50% load on the CPU, whereas for the `HIGH` case the same number of workers generating 80% load on the CPU.

To automate the benchmarking process of ProtoCom we make use of the Java Microbenchmark Harness (JMH)[11] that facilitates building, running, and analysing (micro-)benchmarks in Java. We containerised JMH and use Kubernetes OpenKruise[12] to define a `BroadcastJob` that will execute the benchmark on all the nodes in the cluster. We execute five times the benchmark on all nodes. In each run the benchmark initially calibrates the ProtoCom library in a `MEDIUM` accuracy setting. There are three different levels of accuracy one can set: `LOW`, `MEDIUM`, and `HIGH`. We chose `MEDIUM` as

[9] In initial experiments we discovered some warm-up effects affecting the proper calibration of ProtoCom.
[10] https://chaos-mesh.org/
[11] https://openjdk.java.net/projects/code-tools/jmh/
[12] https://github.com/openkruise/kruise

a compromise between accuracy and experimentation time that showed sufficient stability. After the calibration, the benchmark varies the resource demand parameter in three levels 50, 200 and 1000 milliseconds. The selection of the resource demands was motivated from the demands which we inject in the demonstrator application.

4.2 Discussion of Performance Results

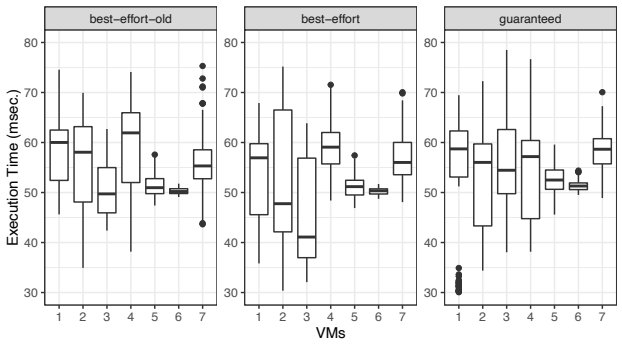
First, we compare the results for the two used Kubernetes QoS-classes and the three parameter levels. Figure 2 summarizes the overall results across three different cluster compilations for various QoS classes and for different load levels: *LOW* (the first three without extra load using Chaos Mesh), *MEDIUM* (the next five, additional stress load of 50%), and *HIGH* (the last two, additional stress load of 80%). The first three configurations (reading from the left) are measurements taken before the major bwCloud maintenance [3], whereas the seven last are experiments after the maintenance. We observe less variability after the maintenance than before, where the performance is highly variable, and observations deviate up to 40% in both directions. Although we cannot hypothesize the root cause behind the improvement, one can speculate on several possibilities. One possibility is that less workload runs on the cloud after the maintenance, leading to performance improvements in our experiments. Another influential factor is the placement of the VMs and the possibility of sharing resources with less noisy neighbors, making the performance less variable. The software overhaul might have also improved the situation with better isolation and scheduling of workloads. All these factors deserve further investigation; however, results show how the performance variability of our load generation mechanism changes in different cluster compilations.

The difference between the assigned container QoS classes is not significant both statistically and practically. Only in the case of high load (*'gua-highstress'* in Fig. 2) the Guaranteed QoS class becomes influential and reduces the variability of measurements.

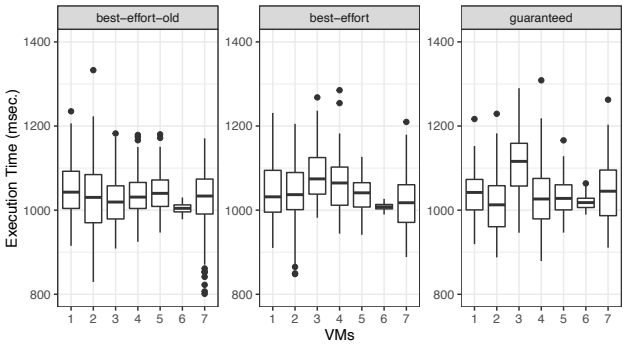
The rest of the paper analyzes closer the results prior to the update of the cloud provider, specifically the *'best-effort'* and *'guaranteed'* cases from Fig. 2. Besides the expected slowdown effects, we also measured several occurrences of speed-ups in our benchmark. The box plots show, that the first quartile is matching the desired execution time. Thus 75% of resource requests are taking more time to complete. Table 3 shows that for parameters 50 and 200 the 95th percentile and standard deviation is slightly lower when comparing best-effort to guaranteed Kubernetes QoS-class. For parameter value 1000 the opposite is true.

Table 3: Execution Time by Parameter and QoS-Class

demand (QoS)	mean	median	95th perc.	SD
50 (best-effort)	52.58131	52.248	66.5653	8.965577
50 (guaranteed)	54.06818	54.102	65.8925	8.115489
200 (best-effort)	211.0581	206.769	254.0771	24.69515
200 (guaranteed)	214.5613	212.349	253.0308	23.41644
1000 (best-effort)	1042.363	1033.147	1155.02	63.28705
1000 (guaranteed)	1042.841	1032.238	1166.254	68.88045



(a) Demand 50



(b) Demand 1000

Figure 3: Variability of measurements across instances for the demand parameters 50, and 1000 for two different Kubernetes QoS classes: *best-effort* and *guaranteed*. The *best-effort-old* is the execution of the benchmark in a different *cluster compilation* prior to the reboot of VMs for the experiment.

Second, we compare the results of the benchmark across the nodes in the cluster. We expect to observe differences in the slowdowns due to different placement of VMs in cloud and different level of occupancy of nodes. As expected, in difference to the QoS class, the variability of the results seems to differ on different nodes. Figure 3 depicts the distribution of data points of all iterations on different nodes. The least loaded nodes—in terms of number of pods deployed—experience less variability. For estimating the reproducibility of the experiment we added an additional benchmark run (best-effort-old) that was conducted in a different cluster compilation before. Results show, though the state of the underlying cloud should have changed, that the variance measured on the nodes is similar.

To analyze the impact of individual node occupancy we calculate the correlation between number of Pods on the node, the CPU utilization in millicores and the sample coefficient of variation (CV) which is the ratio of the standard deviation and the average. In Kubernetes the utilization is measured in millicores that denotes a thousands of one vCPU, i.e., a utilization of 207.15 millicores corresponds to 5.18% for a 4 vCPU node.

The Pearson correlation coefficient (from Figure 4) suggests for a moderate association between both node occupancy, measured in number of pods or millicores CPU, and the relative variability

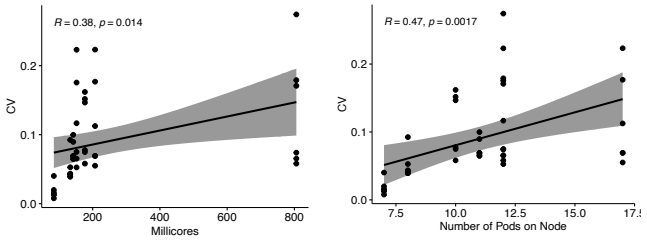


Figure 4: The coefficient of variation plotted against the node occupancy for two different measures: average Millicore CPU (left) and the number of pods deployed on a node (right).

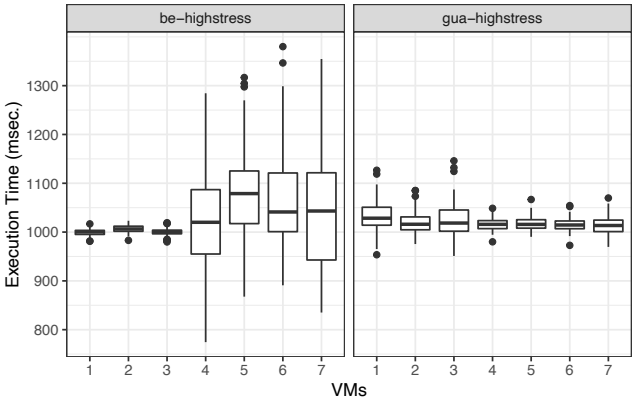


Figure 5: Variability of measurements across instances for the demand parameter 1000 for two different Kubernetes QoS classes: *best-effort* and *guaranteed* in a high stress situation.

measured through the CV. However, the plots from Figure 4 make the correlation inconclusive and highlight that more data is needed. To sum up, we show that the intuitive assumption that a higher node occupancy leads to higher variance in execution time during load generation holds. Moreover, even nodes with moderate average CPU utilization of less than 20% are affected by slowdowns and varying speeds. Developers should take that into account when designing performance tests and benchmarks and measure the variability in processing speeds for multiple nodes. Our simple occupancy metrics serve as an initial indicator for potential slowdown variations, however, several other factors, like the burstiness of CPU workloads or the I/O usage, could also affect the execution. Furthermore, these slowdown effects also impact model-based performance prediction methods, as varying execution time must be taken into account. While one could try to model all influencing factors on the nodes in detail, however, a better solution would be to systematically quantify and model the uncertainty in CPU speeds. Based on these information scaling policies and mechanisms can be optimized.

The container QoS class is not the main influential factor in the experiments before the cloud maintenance. However, in experiments after the maintenance, we observe that the QoS class impacts the performance variability. Figure 5 depicts how the variability changes across nodes for the demand set to 1000 milliseconds for two set of experiments. The *'be-highstress'* (on the left) are the results when using the *best-effort* QoS class for the benchmark container, whereas, the *'gua-highstress'* (on the right) is the case with the *guaranteed* class. In both cases, as we mentioned previously, we inject a background load that utilizes the CPU of the VMs (from 4 to 7) to 80% using an additional container to induce the load. On the VMs from 1 to 3, we do not generate extra load, and the state is similar to the `LOW` scenario. Table 4 depicts the new state of the cluster.

By using the *guaranteed* QoS class, we manage to reduce the variability of the results for the nodes with high utilization. However, we can observe that the variability increases for the nodes where no extra workload is running (Figure 5, VMs 1-3). One possible explanation for this is that the container gets throttled after using the specified amount of resources. We collect the key metrics that monitor the throttling behavior of a container, namely *nr_throttled*, *nr_periods*, and *throttled_time*. The number of times the benchmark container gets throttled in the *best-effort* case (Figure 5, *'be-highstress'*) is 0, whereas the same container is throttled for about 28 seconds on average across runs in the *guaranteed* case (Figure 5, *'gua-highstress'*).

Table 4: Kubernetes nodes and their occupancy in number of Pods and average utilization in millicores for the *"high stress"* scenario

Name	# Pods	Avg. Millicores	Characteristic App Pods
minion-01	18	126.20	
minion-02	12	462.40	rabbit-broker
minion-03	11	67.80	
minion-04	11	3228.40	mongodb
minion-05	9	3234.60	
minion-06	8	3218.40	
minion-07	9	3231.70	demonstrator pods

4.3 Scaling Policies and Mechanisms

Bondi [9] distinguishes four general types of scalability, namely load, space, space-time and structural scalability. In this work we focus only on *load scalability* which is the system's ability "to function gracefully, i.e., without undue delay and without unproductive resource consumption or resource contention at light, moderate, or heavy loads while making good use of available resources" ([9]). Mircoservice.io presents commonly used deployment patterns and distinguishes between the cases of multiple service instances per host and service instance per host, VM or container [19]. AWS autoscaling supports a VM-based scaling approach by launching VMs instead of containers [7]. Kubernetes supports scaling on a container level by applying a horizontal pod autoscaling approach, i.e. it assigns more resources by starting additional replicas of a pod (i.e. a container) that is already running for the current workload

Algorithm 1 Node-based Autoscaling of Services

Require: $currentAvgUtilization$, $0 \leq upperThreshold \leq 1$, $0 \leq lowerThreshold \leq 1$
Ensure: $nodes = replicasPerDeployment$

```
loop                                                    ▷ Control loop
    currentAvgUtilization = getUtilFromK8s()
    if currentAvgUtilization > upperThreshold then
        makeOneAdditionalNodeAvailable()
        scaleOutAllDeploymentsByOne()
    end if
    if currentAvgUtilization < lowerThreshold then
        makeOneNodeUnavailable()
        scaleInAllDeploymentsByOne()
    end if
    sleep()
end loop
```

[6] by calculating the number of replicas by:

$$\text{replicas}_{\text{desired}} = \text{ceil}\left(\text{replicas}_{\text{current}} \cdot \frac{\text{metric}_{\text{current}}}{\text{metric}_{\text{desired}}}\right).$$

While this works for any available metric value, commonly an average utilization metric is used. Pods are placed on existing nodes by the k8s node-scheduler using a two-step filtering and scoring approach[13].

Based on the chosen patterns and technology, software architects might end up with evaluating different kinds of policies. For the demonstrator application, since it is containerized and deployable in Kubernetes, it is possible to employ both service-based policies and also node-based policies. Here we describe the two different policies and the mechanisms to implement them briefly.

For the **service-based** autoscaling policy we rely on the Horizontal Pod Autoscaler (HPA) [6] to define a separate scaling policy for the three different services that constitute the demonstrator application. When fully characterizing the services, various metrics could be used in the configuration of the HPA.

For the **node-based** autoscaling policy, we design and implement a controller that replicates Kubernetes nodes and proportionally scales the pods of the three services deployed on the cluster similar to the cluster proportional autoscaler of Kubernetes [2]. A scale out (in) decision occurs whenever the average utilization of available nodes surpasses an upper threshold (falls behind a lower threshold). The number of replicas for each service follows the number of available nodes. For example, if there are two nodes available in the cluster there will be 2 replicas for each of the services. The initial state constitutes one node whereas there may be up to four nodes where the servises can be allocated. Other nodes are reserved for different services and experimentation tools.

The two different policies lead to different *cluster compilations* over time. In the next section we explore the scalability boundaries of the demonstrator when assuming a node-based autoscaling policy. This allows us to design meaningful scenarios for the elasticity experiments and later evaluate various autoscaling policies including the described ones.

[13] https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

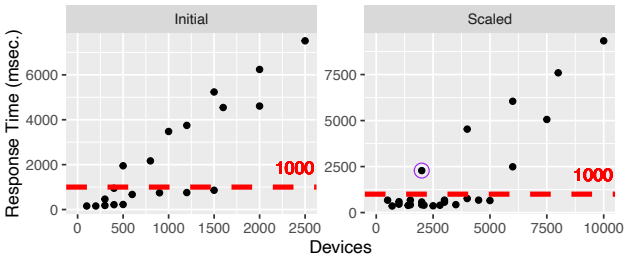


Figure 6: The 95th percentile of response time of the *initial* and *scaled* cluster compilation for synthetic generated load assuming that the application scales using a node-based autoscaling policy.

4.4 Scalability Assessment as a Prerequisite of Designing Elasticity Scenarios

Several proposals exist in literature to asses the scalability of applications, e.g., [12]. The main purpose of assessing the scalability in our case is to estimate the processing limits of the application and use this information for designing elasticity experiments. To determine whether a configuration can handle a certain load (in terms of devices) we observe the performance and the utilization of the system. We follow a binary search procedure [22] to reach faster the upper bound on the number of devices that a particular configuration can handle without violating a given performance service level objective (SLO). For the remote measuring use case, we define the target SLO to be one second response time for the 95th percentile during a constant load of active devices. Since we lack data for workload characterization for the reference application, we generate synthetic load using Gatling that stresses the application in two configurations: the initial configuration where the demonstrator is deployed on one node and the final configuration where the demonstrator is deployed on four nodes.

As Figure 6 depicts, the demonstrator application is able to scale with additional resources i.e., when such resources are provided by means of VM replication and scaling proportionally the corresponding pods. One data point in the plot shows the 95th percentile of response time for requests that have been generated by the active devices given on the x-axis. The red dashed line shows the SLO boundary of 1000ms that should not be violated. The synthetic workload influences response times through the number of concurrent users (devices), how fast they ramp-up, and the sleep value. The combination of the time to ramp up (5 seconds), the frequency of sending data (60 seconds) and the number of devices together with the intrinsic randomness of the workload generation tool and setup leads to different distributions of inter-arrival times for requests. As the highlighted data point in the scaled part depicts that spawning 2000 devices in 5 seconds in one case leads to the violation of the SLOs while the configuration is able to sustain higher number of devices—and up to a maximum of 5000 in one observation—when the devices are increased gradually.

Initial experiments allow us to estimate the performance boundaries of the application when assuming a **node-based** scaling policy for the *initial* and the most *scaled* configuration that may occur. However, when employing a **service-based** policy, although the

initial configuration might be assumed similar, the most scaled configuration is not easily derived upfront due to the dynamics behind adaptations of single services and involved constraints. However, the approximated processing capabilities are used for designing scenarios for elasticity experiments in which both alternatives for autoscaling policies could be evaluated.

5 CONCLUSIONS AND FUTURE WORK

Existing reference applications for experimenting and research are not representative for data-intensive containerized cloud applications. They are serving more traditional use cases of human-centered request-reply communication without a continuous data processing pipeline and without using asynchronous messaging for the communication.

In attempt to fill this gap, in this work we present a reference use case coupled with the initial architecture design and with the engineering challenges for elasticity and resilience. To make the reference application suitable for research of elasticity and resilience mechanisms through increasing the predictability in performance, we have experimented with the CPU load generation tool ProtoCom. Variability of execution times differs for different cluster compilations due to unknown but speculated factors, however, ProtoCom reliably genereates CPU load in cloud environments. We observe a noticeable improvement after a major cloud maintenance and upgrade which illustrates the importance of repeating performance experiments in different times. We observe a moderate association between node occupancy and the relative variability. In addition, we sketch how the application could be scaled through two different autoscaling policies and investigate the scalability of the solution assuming a node-based approach.

In the future we plan to investigate further the factors that impact the high variability of the results. Moreover, we will perform additional experiments using container quotas and CPU throttling. Consolidating and publishing the reference application is another item for future work. In parallel we have started to construct a performance model of the application that will allow the evaluation of architecture alternatives. Moreover, we created initial simulation models to emulate the QoS-based shared CPU scheduling regime used in Kubernetes and alike. In the long run, we want to optimize and make the autoscaling and resilience of dynamic cloud applications more robust. On one side, by offering suitable modeling languages [13] to support architects for design-time analysis, and, on the other side by incorporating models of uncertain execution environments [21] into simulation-based prediction techniques.

ACKNOWLEDGMENTS

This paper was partly funded by the Federal Ministry of Education and Research under grant number 01IS18069A. For computational resources we acknowledge the bwCloud (https://www.bw-cloud.org), funded by the Ministry of Science, Research and Arts Baden-Württemberg (Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg).

REFERENCES

[1] ACME Air: Acme Air Sample and Benchmark . Online: https://github.com/acmeair/acmeair. [Online; 2022-01-25].

[2] Cluster Proportional Autoscaler. Online: https://github.com/kubernetes-sigs/cluster-proportional-autoscaler. [Online; 2022-01-25].

[3] Maintenance of all bwCloud regions starting 2022-03-01, subsequent reregistration necessary - FINISHED! |bw-cloud.org.

[4] Sockshop microservice demo application. Online: https://microservices-demo.github.io.

[5] Spring PetClinic. Online: https://github.com/spring-project:spring-petclinic. [Online; 2022-01-25].

[6] Kubernetes Horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2021. [Online; 2021-12-09].

[7] Amazon EC2 Auto scaling. https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html, 2022. [Online; 2022-01-25].

[8] Steffen Becker, Tobias Dencker, and Jens Happe. Model-driven generation of performance prototypes. In Samuel Kounev, Ian Gorton, and Kai Sachs, editors, *Performance Evaluation: Metrics, Models and Benchmarks*, pages 79–98, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[9] André B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the second international workshop on Software and performance - WOSP '00*. ACM Press, 2000.

[10] Bosch. Mobility Cloud. https://www.bosch-mobility-solutions.com/media/global/products-and-services/mobility-services/plcs/mobility-cloud/21-08-02_bosc_21028-06_mobilitycloud_onepager-rgb_en.pdf, 2021. [Online; 2021-12-16].

[11] Marco Di Natale, Haibo Zeng, Paolo Giusto, and Arkadeb Ghosal. *Understanding and Using the Controller Area Network Communication Protocol: Theory and Practice.* Springer Publishing Company, Incorporated, 2014.

[12] Sören Henning and Wilhelm Hasselbring. How to measure scalability of distributed stream processing engines? In *ICPE '21: ACM/SPEC International Conference on Performance Engineering, Virtual Event, France, April 19-21, 2021, Companion Volume*, pages 85–88. ACM, 2021.

[13] Floriment Klinaku, Mir Alireza Hakamian, and Steffen Becker. Architecture-based evaluation of scaling policies for cloud applications. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2021, Washington, DC, USA, September 27 - Oct. 1, 2021*, pages 151–157. IEEE, 2021.

[14] Christoph Laaber, Joel Scheuner, and Philipp Leitner. Software microbenchmarking in the cloud. how bad is it really? *Empirical Softw. Engg.*, 24(4):2469–2508, aug 2019.

[15] Sebastian Lehrig, Richard Sanders, Gunnar Brataas, Mariano Cecowski, Simon Ivanšek, and Jure Polutnik. Cloudstore—towards scalability, elasticity, and efficiency benchmarking and analysis in cloud computing. *Future Generation Computer Systems*, 78:115–126, 2018.

[16] Sebastian Lehrig and Thomas Zolynski. Performance prototyping with protocom in a virtualised environment: A case study. *Proceedings to Palladio Days*, pages 17–18, 2011.

[17] Albert Lutz, Bernhard Schick, Henning Holzmann, Michael Kochem, Harald Meyer-Tuve, Olav Lange, Yiqin Mao, and Guido Tosolin. Simulation methods supporting homologation of electronic stability control in vehicle variants. *Vehicle System Dynamics*, 55(10):1432–1497, 2017.

[18] Seyed Hossein Nikounia and Siamak Mohammadi. Hypervisor and neighbors' noise: Performance degradation in virtualized environments. *IEEE Transactions on Services Computing*, 11(5):757–767, 2015.

[19] Chris Richardson. microservices.io deployment patterns. https://microservices.io/microservices/news/2015/03/15/deployment-patterns.html, 2021. [Online; 2022-01-25].

[20] Rami Rosen. Linux containers and the future cloud. *Linux J.*, 240(4):86–95, 2014.

[21] Max Scheerer, Martina Rapp, and Ralf Reussner. Design-time validation of runtime reconfiguration strategies: An environmental-driven approach. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 75–81, 2020.

[22] Piyush Shivam, Varun Marupadi, Jeffrey S. Chase, Thileepan Subramaniam, and Shivnath Babu. Cutting corners: Workbench automation for server benchmarking. In Rebecca Isaacs and Yuanyuan Zhou, editors, *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings*, pages 241–254. USENIX Association, 2008.

[23] Sandro Speth, Sarah Stieß, and Steffen Becker. A Saga Pattern Microservice Reference Architecture for an Elastic SLO Violation Analysis. In *Companion Proceedings of 19th IEEE International Conference on Software Architecture (ICSA-C 2022)*. IEEE, March 2022.

[24] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 223–236. IEEE, 2018.

[25] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Trans. Software Eng.*, 47(2):243–260, 2021.