This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2023.3236447



# μBench: an open-source factory of benchmark microservice applications

Andrea Detti, Ludovico Funari, Luca Petrucci
Electronic Engineering Dept.
University of Rome Tor Vergata, Italy
Email: {andrea.detti, ludovico.funari, luca.petrucci }@uniroma2.it

**Abstract**—μBench is an open-source tool for benchmarking cloud/edge computing platforms that run microservice applications. The tool creates dummy microservice applications that can be customized and executed on a Kubernetes cluster. μBench allows users to control fundamental properties of the microservice applications it creates, such as service mesh topology, microservices' behaviors using a portfolio of stress functions (e.g., for CPU, memory, I/O, network) or implementing new ones, microservice-to-microservice API (HTTP or gRPC), etc. Application performance can be evaluated by stochastic or trace-driven workloads.
μBench is aimed at researchers and cloud platform developers who lack real microservice applications to benchmark their findings (e.g., new resource control mechanisms, artificial intelligence-driven orchestration, etc.) or wish to thoroughly evaluate their proposals versus a broad set of heterogeneous applications that μBench can create.
In addition to the description of μBench, in this paper, we show one possible use of it. We compared advantages and disadvantages of microservice architectures versus monolithic ones, and analyzed the performance impact of key architectural choices, such as service mesh topology and the use of replication. For this analysis, we generated several microservice applications with different properties, and two of them are derived from a real cloud dataset.

## 1 Introduction

A microservice application is a network service whose server part is decomposed into a set of "micro" services that collaborate to respond to user requests [1] [2] [3]. Microservice instances typically run on a cluster of machines and interact using inter-service communications that can follow a synchronous or asynchronous style.

The synchronous style is widely used and is based on request-response interactions. A microservice serves a request by performing local processing and interacting sequentially or in parallel with other downstream microservices. This cycle continues downstream until all microservices have completed their execution and responded to upstream callers. The network APIs typically used for inter-service communications are based on HTTP REST [4] or gRPC [5].

The asynchronous style is based on publish-subscribe interactions [6]. Service requests are handled internally as messages or events exchanged asynchronously among microservices. Message broker systems such as Kafka and RabbitMQ support this message exchange [7]. The use of publish-subscribe interactions removes the need to wait for responses from downstream microservices before concluding (in whole or in part) the processing of a request. This feature reduces the time for which a request occupies processing resources and thus makes processing more efficient than synchronous solutions. However, the increased programming complexity often limits the use of the asynchronous style to event-driven applications (e.g., IoT) for which responses from downstream microservices may arrive long after the requests; therefore, it would be unacceptable to keep processing blocked.

The advantages of moving from a monolithic to a microservices architecture are many [8]. For instance, the ability to use a cluster of resources by distributing microservices on different machines. The capability to precisely manage resource allocation by replicating only the most loaded microservices or needing more reliability. The flexibility to use the most suitable programming languages to develop different application parts. Disadvantages include increased latency due to internal network interactions, increased difficulty in debugging, etc. However, the benefits are more significant, especially for complex applications that need to support high request loads.

The most widely used technology today to package a microservice is Linux Containers, managed by Docker [9] or other container engines. Then, automating deployment, scaling, and management of containers (microservices) on a cluster of real or virtual servers is usually done using the Kubernetes (k8s) Container Orchestration platform [10]. Along with microservices, tools for observing their service level indicators (SLIs) are of undeniable usefulness [11]. These indicators usually have two forms: metrics and traces. Metrics are aggregate data such as averages or percentiles of measurements, for example, average request rate or average latency. Traces provide deeper observability than metrics because they are data that track a user request as it flows through various microservices. Managing metrics and/or traces can be done with different data models, protocols, and platforms. We briefly mention that Prometheus [12] is a popular system for collecting metrics, just as Jaeger [13] is a popular tracing platform. In theory, both require specific code to be inserted into microservices, however, for Kubernetes it is possible to

use the Istio platform [14], which automatically deploys an Envoy sidecar proxy for each microservice to intercept its input/output traffic and report Prometheus metrics and Jaeger traces from outside.

Microservice applications have become so popular that the research community is proposing solutions to improve the cloud platforms serving them versus different aspects, e.g., resource management [15] [16] [17] [18], networking [19] [20] [21], storage [22], etc. To experimentally assess the effectiveness of these solutions, many researchers use a set of microservice demo applications [23] [24] [25] [26]. These demo applications have the pro of being real. However, the con is that they cannot be modified to carry out fine-grained sensitivity analysis versus key characteristics of the application. For instance, DeathStarBench [24] is the greatest demo application we know, and it is composed of 33 microservices. If a researcher wants to test his findings for larger applications, e.g., made by hundreds of microservices, either he works in a company that actually has these kinds of large-scale applications (see Netflix, Google) or he simply cannot. Overall, many researchers in this area might have limited benchmarking possibilities.

Accordingly, in this paper, we present µBench: an open-source software *factory* that creates and runs benchmark microservice applications [27]. Researchers that use µBench can control key properties of the generated application, such as

- the number of involved microservices;
- the amount and type of resources (CPU, disk, network) demanded by microservices;
- the service mesh of the application, i.e., the dependency graph among microservices;
- HTTP or gRPC as inter-service communication protocols.

After creating the microservice application, µBench runs it on a Kubernetes cluster and exports per-microservice Prometheus metrics, such as latency and throughput. Benchmarks can be carried out involving a set of microservice per request that is random or trace-driven.

µBench facilitates the understanding of microservice applications, improves the assessment of cloud solutions versus different characteristics these applications can have, and, last but not least, we found it useful also for educational purposes to show the advantages, problems, and challenges of microservice applications to students.

In the next Sec. 2 we discuss related work. In Sec. 3, we present the µBench tool. In Sec. 4, we use µBench to compare the advantages and disadvantages of microservice architectures versus monolithic architectures and analyze the performance impact of key architectural choices, such as service network topology and replication. Finally, in Sec. 5, we report the conclusions and lessons learned from the analysis.

## 2 Related Works

Microservice demo applications are used extensively [23] [24] [25] [26] [32] to study the performances and behaviours of applications based on microservices. Many of them are discussed in [33], which lists the main requirements a benchmark for microservices should have.

Teastore [23], for example, is a six-microservice application that implements an e-commerce website selling tea-related products. It has been used by [34] [35] [36] to study software management methods, such as autoscalers, evaluate performances, bottlenecks and optimization methods. Eismann et al. [34] exploit the TeaStore application to discuss the pros and cons of Microservices from a performance tester's perspective. Caculo et al. [35] leverage the TeaStore application to demonstrate that selective scaling of services rather than the whole application, can result in uplifted performances.

SockShop [25] is a similar microservices storefront demo application designed for testing and benchmarking, larger than TeaStore as it is composed of 14 distinct microservices. It has been used extensively throughout various studies [33] [23] [34] [35] [24] [32] [36].

DeathStarBench [24] is a suite of benchmarking microservices-based applications used to study the architectural complexity that a microservices applications can reach, their challenges and trade-offs. DeathStarBench includes five end-to-end applications covering social networks, multimedia services, hotels booking, e-commerce sites, and banking systems. At the time of writing, three out of five applications are available [28].

Other microservice benchmarks have been used to bring to light the intrinsic strengths and flaws of microservices, e.g. Ueda et al. [37] exploit AcmeAir benchmark [38] to compare it with the monolithic implementation. Other works compare multiple microservices benchmarks altogether, e.g., Rao et al. [36] use [24] [23] [25] to analyze performance enhancements proposing a placement scheme to map the services of the application to different cores. Several other microservice applications are available like Google's demo application Online Boutique [29], Bookinfo [26], JPetStore [30], PetClinic [31]. Though many of them are not intended to be research benchmarking applications but rather applications for demonstrational purposes. In the following Sec. 2.1, we describe and compare the mentioned applications including our µBench.

In addition to demo applications, it is worth noting that the Alibaba Cluster Trace Program recently released a dataset containing one week of network calls made by 20,000 microservices running in Alibaba cluster [39]. For each user request, the dataset reports the trace of calls made by the involved microservices, allowing in-depth analysis of the characteristics of call graphs [40]. However, in these traces, there is no information about which microservices make up the different applications. It is shown that during a trace $X$, the microservice $A$ calls microservice $B$, but no information is given about which application $A$ and $B$ belong to. Consequently, a technique based on Spectral Clustering is proposed in [41] to group similar microservices so that each group forms a different application.

| | **µBench** [27] | **DeathStarBench** [28] | **TeaStore** [23] | **Online Boutique** [29] | **Bookinfo** [26] | **Sock Shop** [25] | **JPetStore** [30] | **PetClinic** [31] |
|---|---|---|---|---|---|---|---|---|
| **Purposes** | Benchmark | Benchmark, Demo | Benchmark, Demo | Benchmark, Demo | Benchmark, Demo | Benchmark, Demo | Benchmark, Demo | Benchmark, Demo |
| **Number of Apps** | Configurable | 3 | 1 | 1 | 1 | 1 | 1 | 1 |
| **Number of Microservices** | Configurable | 19, 31, 33 | 6 | 11 | 4 | 14 | 4 | 5 |
| **Call Method** | REST, gRPC | REST | REST | gRPC | gRPC | REST | REST | REST |
| **Work model** | Configurable | App. Dependent | App. dependent | App. dependent | App. dependent | App. dependent | App. dependent | App. dependent |
| **Service Mesh** | Configurable | App. dependent | App. dependent | App. dependent | App. dependent | App. dependent | App. dependent | App. dependent |
| **Spanning model** | Stochastic, Trace driven | User driven | User driven | User driven | User driven | User driven | User driven | User driven |
| **Languages** | Python (other with sidecar container) | C++, Java, Node.js, etc. | Java | Node.js, Python, Go, etc. | Python, Java, Ruby, Node.js | Java, Go, Node.js | Java | Java |
| **Platforms** | Kubernetes | Kubernetes, Docker compose, Openshift | Kubernetes, Docker compose | Kubernetes | Kubernetes | Kubernetes, Docker compose | Kubernetes | Kubernetes, Cloud-Foundry |
| **Exported Performance** | Metric, Traces | Metric,Traces | Metric, Traces | Metric, Traces | Metric, Traces | Metric | Metric, Traces | Metric |

Table 1: Comparison with the most popular microservice benchmarking applications.

### 2.1 Comparison of Benchmark Applications

Table 1 provides a quick overview of the most popular microservice benchmarking applications and compares them based on common characteristics. One of the main strengths of µBench is the configurability of the generated applications, which enables extensive sensitivity analysis of cloud solutions designed to optimize microservice architectures. Benchmarking applications composed of a variable number of microservices can be created. In addition, the *work model*, i.e., the way the requests are served, can be configured both in terms of which microservices are involved (*service mesh*) and in terms of which internal functions are performed by the microservices so that it is possible to select which resources (CPU, disk, memory, etc.) to stress. For other applications, these "structural" parameters cannot be configured since they are real applications that, therefore, can also be used for demonstration purposes.

The performance of a µBench-generated application can be measured using two *spanning models* for selecting the microservices involved in a request: stochastic or trace-driven. For the stochastic model, a request is served involving a random sequence of microservices. For the trace-driven model, the user chooses this sequence, which allows µBench to be used even with real traces, such as those provided by Alibaba [39]. As for the other applications, their performance evaluation is usually based on a predefined sequence of HTTP calls that mimic the user's behavior and are generated by external tools, such as Jmeter, for which they provide the input test files.

µBench's microservices are implemented in Python, so it does not emulate multi-language applications in their basic form. However, it is possible to associate a µBench microservice with a sidecar container invoked with each request. This sidecar can run user-defined programs in any language or be an actual application (e.g., MongoDB). Most other benchmark applications, instead, are natively multi-language.

All applications can be deployed via Kubernetes; some also support other orchestrators, such as Docker compose. Benchmark results are exposed as metrics collected by Prometheus or as traces usually managed by a Jaeger server. µBench microservices independently export performance metrics to a Prometheus server and, for tracing, can be integrated with the tracing capabilities offered by Istio and Jaeger.

## 3 µBench

### 3.1 Application model

µBench generates applications like the one shown in Fig. 1. They consist of microservices whose quantity and work model can be configured. Clients access the application through an API gateway that routes incoming HTTP requests to relevant internal microservices.

Each microservice implements a simple synchronous work model whereby the work done to serve a request is performed in three consequential time phases:

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2023.3236447



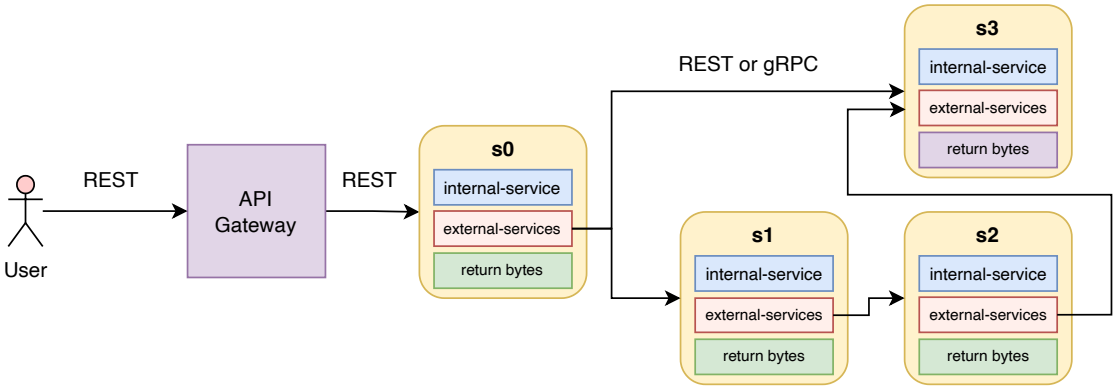



Fig. 1: A µBench microservice application

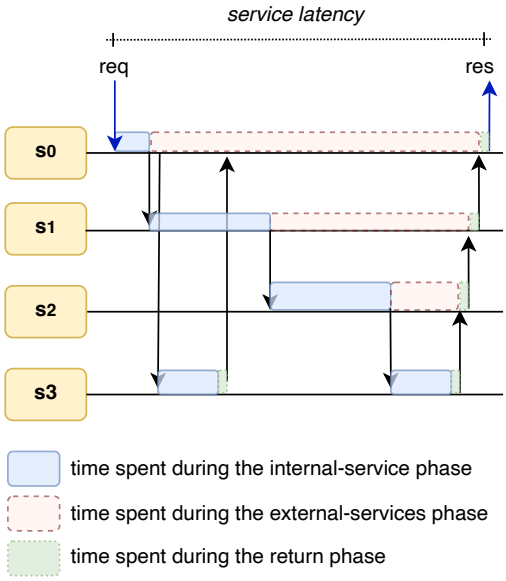



Fig. 2: Trace of a service request

- *Internal-service* phase: execution of an internal-service, i.e., a function that stresses local resources (CPU, disk, memory).
- *External-services* phase: invocation of a set of external-services, i.e., services offered by downstream microservices, and waiting for their results.
- *Return* phase: return of a dummy amount of bytes to the upstream calling party [1].

Fig. 2 shows a trace of the inter-service communications to serve a request received from `s0`. The microservice `s0` spends time executing its internal-service, then makes two parallel calls, to `s3` and `s1`, and waits for their responses. When both these responses arrive, `s0` sends a response to the client consisting of a few return bytes. Similarly, when s1 receives a request from `s0`, it executes its internal-service, calls `s2`, waits for the response, returns the response bytes to `s0`, and so on.

1. We note that in real applications, a microservice may consume local resources not all at the beginning but partly at the beginning and partly during the execution of the service, for example, after receiving the results of external calls to process the received data. However, we believe that our work model that aggregates the overall use of local resources at the beginning simplifies configuration and does not alter the benchmarking capability that this tool aims to.

### 3.2 Software toolchain

µBench builds and deploys microservice applications using the toolchain shown in Fig. 3. It is formed by three Python tools that cope with different aspects of the production pipeline. In the next sections, we will describe better these tools; here, we briefly present them as follows:

- Service Mesh Generator: generates the topology of the service mesh and defines a spanning strategy, for example, whether using parallel or sequential calls to serve a request and probabilities of calling downstream services. It takes as input a parameter file and produces a `servicemesh.json` file.
- Work Model Generator: generates the work model of each microservice, i.e., what its internal-service performs, which are the called external-services (from `servicemesh.json`), and the number of average bytes to send back. It takes as input a parameter file, the `servicemesh.json` file coming from the Service Mesh Generator or the user, for a manually-defined mesh, and produces a `workmodel.json` file that includes the work model of each microservice.
- K8s Deployer: deploys the microservice applications on Kubernetes by creating the necessary resources. It takes as input a parameter file and a `workmodel.json` file coming from the Work Model Generator or manually configured.

### 3.3 Service Mesh

The µBench *service mesh* describes which microservices make up the benchmark application and how these microservices call each other to serve a request. This latter property depends on the spanning model of the mesh that can be stochastic-driven or trace-driven.

#### *Stochastic-driven spanning model*

For the stochastic-driven model, each microservice has its own set of external-services and this set is organized into groups called *external-services-groups*. Microservices belonging to different groups are called in parallel,

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2023.3236447



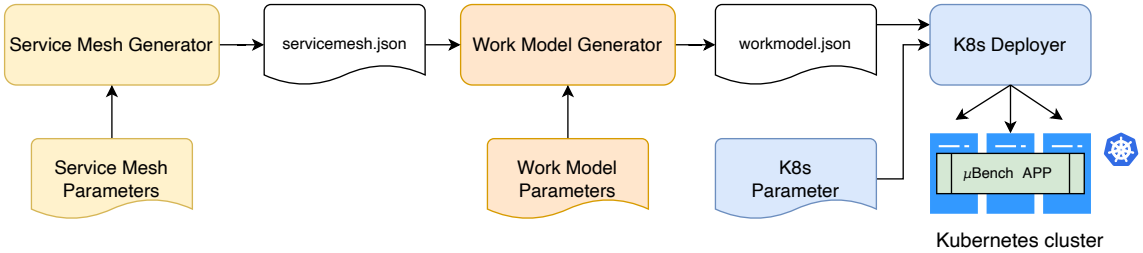



Fig. 3: µBench toolchain

those in the same group are called sequentially, and calls are actually executed according to configurable probabilities associated with each microservice. With this model, µBench can reproduce applications that call downstream microservices in parallel or in sequence. In addition, each application can randomly involve several microservices, thus reproducing stochastic spanning patterns of the service mesh.

```
{"s0": {
  "external_services": [
   external-service-group n.1
   {"services":["s1"],"probabilities":{"s1":1}},
   external-service-group n.2
   {"services":["s3"],"probabilities":{"s3":1}}
  ]},
 "s1": ...}
```

Listing 1: `servicemesh.json` file of application in Fig. 1 with parallels calls. Comments in italic gray fonts

The service mesh is described by JSON objects included in a `servicemesh.json` file. For each microservice, there is a specific JSON array, named `external-services`, whose array elements are the external-services-groups. For example, Listing 1 is the `servicemesh.json` of the application in Fig. 1 in which, for simplicity, only the part of the service mesh related to `s0` has been reported, whose external-services are `s1` and `s3`, both of which are called in parallel during the processing of a service request with a probability of 1, as in Fig. 2. To switch from parallel to sequential spanning, it is sufficient to use a single external-service-group that contains both `s1` and `s3`, and to introduce randomness in mesh exploration, it is sufficient to reduce the call probabilities.

### *Trace-driven spanning model*

The stochastic-driven model that has been presented can be overridden by a trace-driven model whose requests include in their body the sequence of microservices that must be spanned, in parallel or in sequence. The sequence is described by a recursive structure of JSON objects, such as those in Listing 2, representing the trace in Fig. 2.

Currently, the µBench repository [27] contains the service meshes of 30 applications derived from the Alibaba dataset [39] through our software. For example, Fig. 4a and Fig. 4b show the service meshes of two such applications (No. 18 and 22) that have 42 and 78 microservices, respectively. Moreover, for each application, the repository provides a set of traces (JSON files like the one in Listing 2) that are extracted from the dataset and can be used for trace-driven benchmarks.

```
{"s0":[
  {"s1":[
    {"s2":[
      {"s3":[{}]}]}]},
  {"s3":[{}]}]}
```

Listing 2: `trace.json` file of the trace in Listing 2

### *Service Mesh Generator*

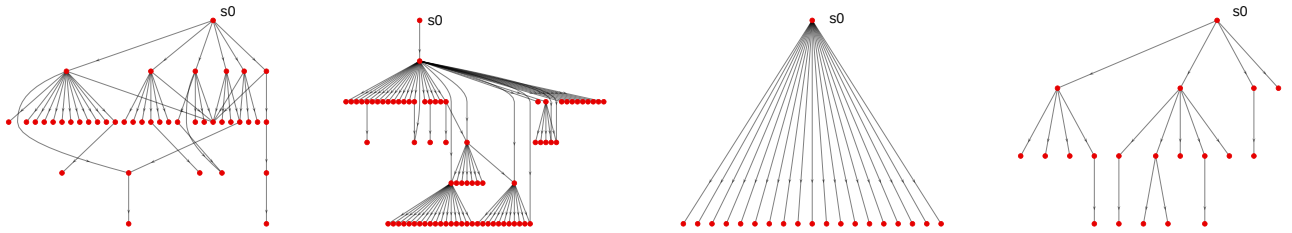
The `servicemesh.json` file that describes the service mesh of a µBench application can be created manually for simple topologies like chain or hub-and-spoke topologies. Even for trace-driven benchmarks, the construction of the `servicemesh.json` file is rather simple, because it is not necessary to specify the external-services since the sequence of involved microservices is driven by the trace.

However, for stochastic-driven benchmarks when the size of the application grows, it can be convenient to use the Service Mesh Generator tool (Fig. 3) that creates random topologies by using the Barabási-Albert algorithm [42]. The topology is built by an iterative process for which, at each step, a new microservice is added as a vertex of a directed tree. This new microservice is connected with a mesh link to a single upstream (parent) microservice already present in the topology, which will include the new microservice in its external-services set. The upstream microservice is chosen according to a preferential-attachment strategy using a power-law distribution. Specifically, the microservice $i$ is chosen as an upstream with a (non-normalized) probability equal to $P_i = d_i^{\alpha} + a$, where $d_i$ is the number of microservices that have already chosen the microservice $i$ as an upstream, $\alpha$ is the power-law exponent, and $a$ is the zero-appeal parameters i.e., the probability of a microservice being chosen as a parent when no other microservice has yet chosen it.

By changing the parameters $\alpha$ and $a$, we can generate microservice applications with different mesh topologies such as those shown in Fig. 4. For example, Fig. 4c simulates the case of an application with a central orchestrator (`s0`) that intercepts requests, invokes downstream microservices, and processes their responses. The other topology in Fig. 4d represents the case of a more decentralized model.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2023.3236447






(a) Alibaba app. 18, 42 microserv. (b) Alibaba app. 22, 78 microserv. (c) $\alpha = 0.9, a = 0.01$, 20 microserv. (d) $\alpha = 0.05, a = 3.25$, 20 microserv.

Fig. 4: Service mesh topologies from Alibaba traces (A,B) and generated by the Service Mesh Generator (C,D)

Regarding the calling probabilities, the Service Mesh Generator allows using different distributions for extracting the value of such probabilities or using a constant value for all. In any case, these probabilities can be fine-tuned a posteriori by editing the produced `servicemesh.json` file.

### 3.4 Work model

The *work model* of a µBench application specifies, for each microservice, (i) the function that it runs as internal-service, (ii) which external-services it calls, for stochastic-driven benchmarks, (iii) how many bytes it sends back, and (iv) other system parameters.

The work model is described in a `workmodel.json` file, made of a JSON object per microservice. For instance, Listing 3 is the work model of the application in Fig. 1. For simplicity, we reported only the JSON object related to the microservice `s0`. With this configuration, `s0` executes an internal-service function called `loader` with specific paramenters, then calls the external-services `s1` and `s3` in parallel as described in Sec. 3.3, and finally sends back a response with average size of 11 kbytes (`res_size`).

Each microservice executes an internal-service chosen by the user from those available in an portfolio of functions. The portfolio consists of a set of Python files that the user can extend to introduce custom functions that stress specific aspects of interest. The current function portfolio of µBench (v1.2) includes the `loader` function and other simpler functions. The `loader` function stresses CPU, disk and memory in a user-configurable way. CPU stress is obtained by computing a number of digits of Pi ($\pi$) using the Unbounded Spigot Algorithms [43]. The number of digits is a uniform random variable in the range `range_complexity` and the computation is repeated `trials` times. Memory is stressed by allocating a `memory_size` bytes and then performing `memory_io` read and write operations of 1 byte each. Disk stress is performed by writing a number (`disk_write_block_count`) of blocks of a fixed size (`disk_write_block_size`) and then reading them randomly. The specific stress action can be enabled or disabled by changing the boolean value of the `run` key.

The work model of a microservice also allows control over the resources involved. Specifically, the user can control the number of parallel processes (`workers`) and threads used by the microservice, the values of CPU/Memory Requests and Limits that Kubernetes will assign to an instance of the microservice, and the number of replicas of the microservice. Finally, the work model specifies the method used to call external-services, REST or gRPC, and includes the name of the container image that implements a µBench microservice, which we will discuss in the next session.

```
{"s0": {
  "internal_service": {
    "loader": {
     "cpu_stress": {
      "run": true,
      "range_complexity": [100,1000],
      "trials": 1},
     "memory_stress": {
      "run": false,
      "memory_size": 10000,
      "memory_io": 1000},
     "disk_stress": {
      "run": false,
      "tmp_file_name": "mubtestfile.txt",
      "disk_write_block_count": 1000,
      "disk_write_block_size": 1024},
     "sleep_stress": {
      "run": false,"sleep_time": 0.01},
     "res_size": 11}},
  "external_services": [
    {"services":["s1"],"probabilities":{"s1":1}},
    {"services":["s3"],"probabilities":{"s3":1}}]
  "request_method": "rest",
  "workers": 8, "threads": 128,
  "cpu-requests": "1000m", "cpu-limits": "1000m",
  "replicas": 1,
  "image": "msvcbench/microservice_v3:latest"},
 "s1" : ...}
```

Listing 3: µBench work model description

*Work Model Generator*

The `workmodel.json` file can be created manually or, when the size of the application grows, it may be convenient to use the Work Model Generator tool (Fig. 3) to create random applications. The tool takes as input the information of the service mesh to configure the external-services of microservices. To configure the internal-services, the tool takes as input a list of *function-flavors*, such as `f0` in Listing 4. A function-flavor is a function, such as `loader`, customized with specific parameters and and with a probability of being selected by the Work Model Generator as the internal-service of a microservice.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2023.3236447



```
{ "WorkModelParameters":{
   "f0": {
    "type":"function",
    "value": {
      "name": "loader",
      "recipient": "service",
      "probability":0.5,
      "parameters": {
       "cpu_stress": {"run":true, ...},
       "memory_stress":{"run":false, ...},
       "disk_stress":{"run":false,...},
       "sleep_stress":{"run":false,...},
       "res_size": 11},
      "workers":4, "threads":16,"replicas": 1
      "cpu-requests": "1000m", "cpu-limits": "1000m",
    }},
   "f1": { ...}}...}}
```

Listing 4: µBench work model generator parameters

| Type | Name | Description |
|---|---|---|
| Deployment | sx | Deployment of microservice sx |
| Deployment | gw-nginx | Deployment of NGINX API gateway |
| Services (Node Port) | sx | Services of microservice sx |
| Service (NodePort) | gw-nginx | Service of NGINX API gateway |
| ConfigMap | gw-nginx | ConfigMap for nginx configuration |
| ConfigMap | internal-services | ConfigMap includig custom functions of internal-services |
| ConfigMap | workmodel | ConfigMap includig workmodel.json |

Table 2: Kubernetes resources used to run a µBench application

### 3.5 Application deployment

As shown in Fig. 3, k8s Deployer is the final tool of the chain and is responsible for creating the Kubernetes resources (Tab. 2) needed to run a µBench application on a Kubernetes cluster. A manually configured µBench application can be run using this tool and manually editing a `workmodel.json` file.

The k8s Deployer takes as input the work model description (`workmodel.json`) and some Kubernetes parameters. Accordingly, the tool leverages the Kubernetes API to deploy microservices as k8s Deployments consisting of one or more PODs, depending on the number of replicas of the specific microservice. The service offered by each microservice is exposed through a dedicated k8s (Node Port) Service.

As shown in Fig. 5, a POD of a µBench microservice is made of a container we called *service-cell*. This container executes a Python program that implements the microservice logic [2]. K8s ConfigMaps provide PODs with the `workmodel.json` file and the set of Python files that implements internal-service functions.

When a POD boots, it is informed of its ID (e.g., `s0`, `s1`,...) through an environment variable, and then the service-cell is started. Consequently, the service-cell imports the internal-service functions into its Python code and reads the `workmodel.json` file to discover how to serve incoming request [3]. The service-cell exports some Prometheus metrics, including the time taken to run the internal-service and the time taken for calling downstream external-services, the number of bytes sent back, and the number of requests processed [11].

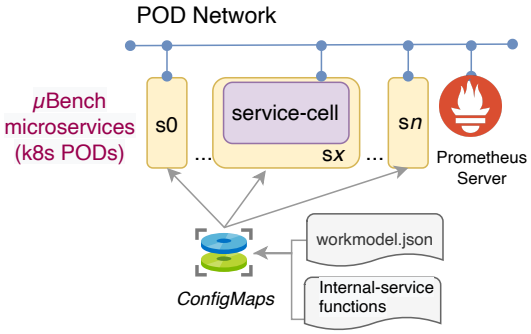



Fig. 5: Implementation of a µBench application in a Kubernetes cluster

## 4 Analysis of microservice applications

This section reports some simple analyses we made with µBench, whose objective is to provide an overview of the possible exploitation of the tool for educational and research purposes [4]. The testbed platform is a Kubernetes cluster with $K = 3, 5, 10$ worker nodes where µBench microservices run; the nodes are VMs provided by Azure cloud with 4 CPUs each[5]. To generate the request stream, we used Apache JMeter [44]. $M$ users generate requests in parallel towards the microservice `s0`, considered as the application entry-point. When a request is satisfied, the user immediately resends another one. In this way, the number of requests concurrently served is kept constant to $M$ and the resulting number of requests served per second, i.e., the throughput $T$, is equal to $M/D$, where $D$ is the average request latency.

### 4.1 Monolithic vs. Microservice

When claiming the advantages of microservice applications over monolithic implementations, a typical argument

2. The implementation is based on Gunicorn WSGI HTTP server and gRPC libraries

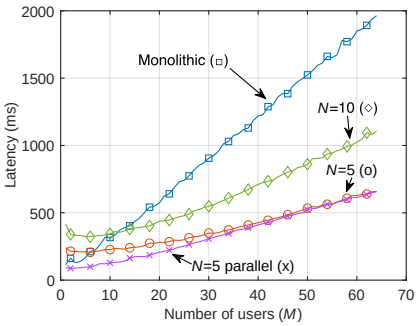
3. To make µBench applications more realistic, it is possible to configure the work model to include a *sidecar* container in the microservice PODs. The sidecar container can be invoked by the internal-service of the service-cell. The sidecar container is meant for executing a real service used in microservices applications, such as a MongoDB database, and can be invoked by a internal-service of the service-cell that the user must develop.

4. The `workmodel.json` files used to generate the application used in this section are available in [27]

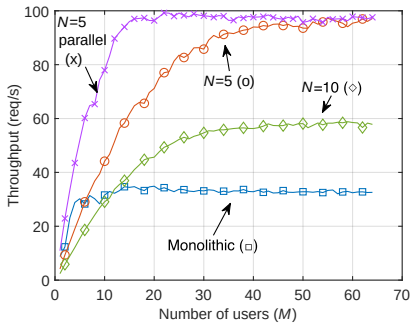
5. Azure virtual machines run in the Western Europe region, they have 4 CPUs at 2.3GHz (without Hyper-Threading), 16 GiB of RAM and a Gigabit Ethernet interface. All the VMs run Ubuntu 18.04.4 LTS with the 64-bit version of the x86 instruction set architecture (ISA). The throughput of internal communications is 1 Gbit/s, and the VM-to-VM RTT is less than 2ms. The request stream is generated by a different Azure VM with 8 CPUs

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2023.3236447

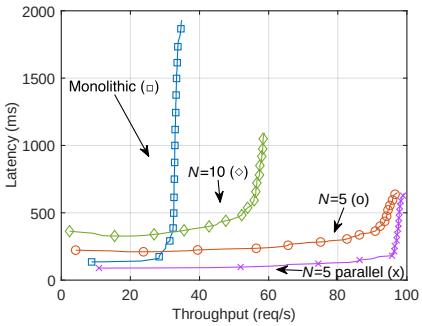




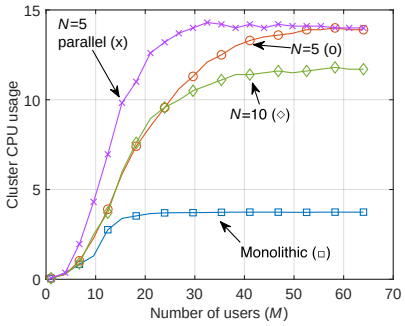

(a) Request latency vs the number of users ($K = 5$, $N = 1, 5, 10$)



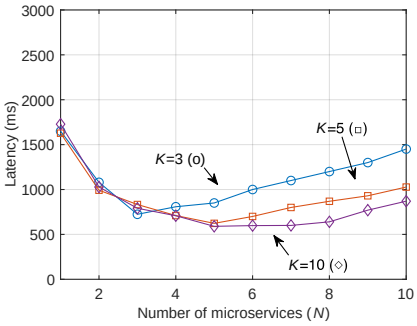

(b) Request per seconds vs the number of users ($K = 5$, $N = 1, 5, 10$)



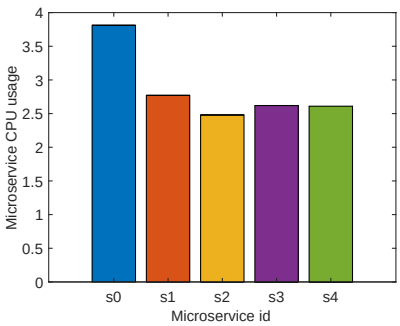

(c) Request latency vs request per seconds ($K = 5$, $N = 1, 5, 10$)



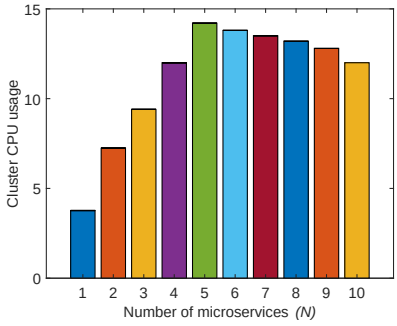

(d) CPU used in the cluster vs the number of users ($K = 5$, $N = 1, 5, 10$)




(e) Request latency vs. the number of microservices ($K = 3, 5, 10$, $M = 60$)




(f) CPUs used by microservices ($N = 5$, $K = 5$, $M = 60$)




(g) CPUs used in the cluster vs. the number of microservices ($K = 5$, $M = 60$)

Fig. 6: Performance of microservice applications in Fig. 4c

is their ability to scale better on a cluster of resources. To provide evidence of this claim, we used µBench to create a set of applications made of a different number $N$ of microservices. The case $N = 1$ simulates a monolithic implementation. We configured the work model so that:

- the service-mesh has the star topology in Fig. 4c and the calling probabilities are all equal to 1; consequently, users send requests to the microservice s0, which in turn calls all the other $N - 1$ downstream microservices, either in sequence or in parallel;
- the inter-service communications use the HTTP/REST model;
- the internal-service run by the $N$ microservices is the same. It is the `loader` function presented in Sec. 3.4, configured to transmit back 100 kB, and stress only the CPU by computing 100 digits of Pi ($\pi$) and repeating this computation $100/N$ times. The overall amount of $\pi$ digits computed by the application is $100,000$ independently of $N$. Therefore, the work done by the application for the user is the same, and we can make a fair comparison versus $N$ to understand the impact of splitting the application into microservices.

The following paragraphs analyze the pros and cons of splitting an application into microservices. First, we consider applications for which `s0` calls $N-1$ downstream microservices sequentially, and then applications for which `s0` makes these calls in parallel all at once.

### *Sequential downstream calls*

Fig. 6a shows the request latency for a cluster of $K = 5$ worker nodes, varying the number of users ($M$) and for three different implementations: monolithic ($N = 1$) and microservice-based with $N = 5, 10$ microservices. Fig. 6b shows related throughput in requests per second and Fig. 6c shows the request latency versus the throughput. The figures also show the results obtained for a scenario for which external services are called in parallel ($N = 5$ parallel), which we will comment on in the next subsection.

We note that the monolithic implementation provides lower latency and higher throughput when the number of users is low ($M \leq 5$) [45]. The microservice implementations perform better when the users grow, especially the implementations composed of 5 microservices.

To explain this behavior, it is worth mentioning that the latency of a request is the sum of computational and I/O (e.g., storage, network) delays introduced by the involved microservices (Fig. 2). In terms of computation,

a microservice executes the internal-service and handles interactions with external-services (open/close sockets, process received data, etc.). For the considered applications, the overall footprint of internal-services on the cluster CPUs is independent of $N$ because 200,000 $\pi$ digits are calculated per request in each configuration. Instead, the overall computational load associated with handling external-services, which we name *CPU overhead*, increases with $N$ because the number of microservices to be called increases.

In terms of I/O, microservice implementations introduce a network delay proportional to the number of microservices involved per request. In fact, `s0` must interact sequentially with $N-1$ downstream microservices before sending the result back to the client. We name this penalty as *network overhead*.

When the number of users is low ($M \leq 5$), the CPU/network overheads penalize microservice implementations. In contrast, microservice architectures show their advantages when the number of users increases ($M > 5$). In fact, the overheads are abundantly outweighed by the ability to leverage cluster resources in parallel to deal with the computational load [6].

In this regard, Fig. 6d shows the average number of CPUs used in the cluster versus the number of users. When the number of users is low ($M \leq 5$), the exploited CPU resources are similar. Therefore, microservice implementations offer no computational advantage but only penalties given by CPU/network overheads. When the number of users increases, the monolithic implementation serves the requests stream exploiting at most the 4 CPUs of the host where it runs. On the contrary, microservices implementations run their microservices on many hosts and this allows them to exploit up to 14 CPUs, and the ability to use more CPUs results in a relevant reduction of computational delay and improvement in throughput.

Obviously, by increasing the number of CPUs of the monolithic application, it would perform even better than the microservices one by not having to pay for the CPU/network overhead. However, such vertical resource scaling has technological bounds (e.g., the largest VM in Azure today provides 416 vCPUs) that microservice applications can overcome by leveraging resources in parallel.

The results show that splitting into five microservices is the best choice when users grow. This fact indicates the existence of an optimal number $N$ of microservices between 1 and 10 in which to split the application. Accordingly, Fig. 6e shows the latency of requests as a function of the number of microservices the application is divided into, for three different cluster sizes ($K$), namely 3, 5 and 10 nodes. For a cluster of 5 nodes as that used in Fig. 6a, the number of microservices providing the minimum latency is indeed 5. This optimal number $N$ of microservices depends on the cluster's configuration and the application's work model. In a very general way, we can say that by increasing the number of microservices, the application tends to exploit the computing resources of more hosts, and this reduces the impact of computational delays on request latency. At the same time, however, the number of involved microservices to solve a request is higher, increasing the impact of CPU and network overhead on request latency. For clusters with $K = 3, 5$ nodes, when we divided the application into $N = K$ microservices, Kubernetes distributed each of them on a different node. In this way, the application can use all the cluster resources. This better utilization of cluster resources overcomes the CPU and network overhead inherent in microservice architectures, so the $N = K$ configuration provides the lowest delay. For the cluster with $K = 10$ nodes, the CPU and network overheads for in the $N = K$ case are so high that they cancel out the advantage of having the ability to use the entire pool of cluster CPUs. Consequently, the lowest delay is obtained for $N < K$. In any case, we can say that dividing the application into a greater number of microservices than the number of nodes decreases performance because it adds unnecessary CPU/network overhead; therefore, the optimal value of $N$ lies in the range $1 \leq N \leq K$ and is the value that provides the best tradeoff between cluster resource exploitation and CPU/network overhead.

We note that with a cluster of 5 nodes and for 5 or 10 microservices, the application would theoretically be able to utilize all the CPUs of the cluster, i.e., 20 CPUs, but Fig. 6d shows that, on average, only 14 CPUs are used for $N = 5$, and 12 CPUs for $N = 10$. This inefficiency is related to the fact that `s0` requires more computing resources than the other microservices because it has to handle $N-1$ external-services, while the other microservices have no external-services. In practice, all the CPU overhead of this application is borne by `s0`. As a result, `s0` saturates its resources before the others and acts as *throughput bottleneck*, preventing the application from fully exploiting cluster resources. In this regard, Fig. 6f shows the CPU used by microservices in the case of five microservices. The microservice `s0` uses about the entire set of 4 CPUs of the node where it runs. In contrast, the other microservices use fewer CPUs because `s0` does not send them requests fast enough to saturate their computing resources. This bottleneck behavior of `s0` worsens by increasing the number of microservices because the CPU overhead increases. Consequently, the exploited CPUs are more for $N = 5$ than $N = 10$.

Fig. 6g shows the CPU usage of the cluster with 5 nodes versus the number of microservices. Up to five microservices, the amount of CPU exploited increases because the number of nodes over which the application is distributed increases as well. Thereafter, there are no other nodes to exploit, so only the bottleneck effects of `s0` increase and CPU utilization tends to decrease slightly.

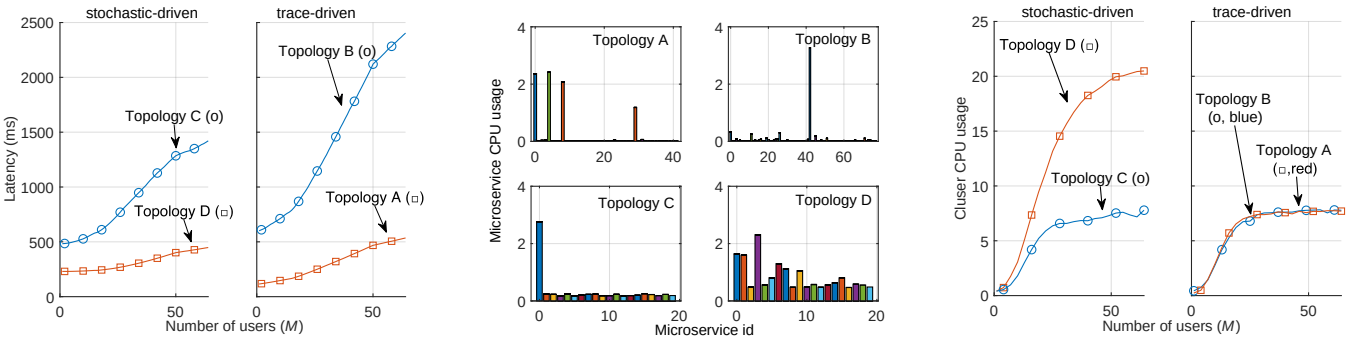
*Parallel downstream calls*

Fig. 6a, 6b, 6c and 6d, also show the performance achieved when s0 calls the downstream microservices in parallel in the case of $N = 5$ microservices. In this case, the latency penalty due to network overhead is limited to

6. We verified that this conclusion and the subsequent ones on latency are also valid when considering the 99th percentile of delays.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2023.3236447






(a) Request latency vs the number of users (b) CPUs used by microservices for $M = 40$ users (c) CPUs used in the cluster vs the number of users

Fig. 7: Perfromance of the microservice applications using topologies in Fig. 4 for sequential downstream calls in a cluster of $K = 10$ nodes

a single round-trip-time, so it is very low and immediately outweighed by the benefits of parallel processing provided by microservice implementations. This results in reduced latency and improved throughput for any number of users.

Comparing the performance of parallel and sequential calls for $N = 5$, we notice that with the former the cluster CPU utilization grows more quickly due to the parallel use of resources that occurs even within the same request. On the contrary, in the case of sequential calls, the parallel use of resources occurs only among different requests, so the full utilization of the cluster CPUs occurs only when there are many requests/users served simultaneously. The limits of throughput and CPU saturation remain the same for in sequence and parallel approaches because the cluster resources are however fully exploited with many users.

### 4.2 Impact of service mesh

In this section, we evaluate the impact of the service mesh on the performance of microservice applications specifically considering the topologies in Fig. 4 in the case of sequential downstream calls.

For the topologies (C and D) obtained from the Service Mesh Generator, we performed stochastic benchmarks, with a call probability equal to 1 for each microservice. Therefore, each request is served by a constant number $L = 20$ of microservices. For the Alibaba meshes (A and B), we performed trace-driven benchmarks in which, for each request, we randomly select a trace related to the specific application. As a result, the number of microservices involved for each request depends on the specific trace and Fig. 8 shows the related CDF. The application traces related to topology A involve fewer microservices ($L = 5$ on average) and the variance is rather small. In contrast, the application traces related to topology B involve a larger number of microservices ($L = 25$ on average) and the variance is higher.

To better highlight the effect of service mesh in performance comparison, we ensured that the average CPU load required to serve a request by the different applications was equal to each other. Specifically, each microservice computes an amount of $\pi$ digits equal to $2000/L$.

Fig. 7 shows latency, cluster CPU utilization, and CPU utilization per microservice for the different topologies.

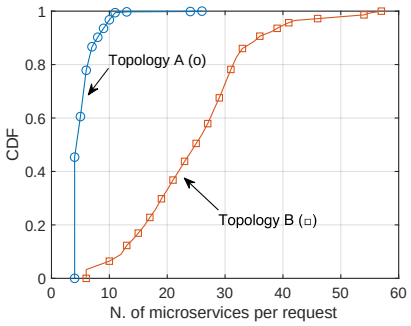



Fig. 8: CDF of the number of involved microservices per request in the case of applications derived from the Alibaba dataset.

Let us first comment on topologies C and D, for which the number of microservices involved per request is the same ($L = 20$). The application with topology C has higher latency than D because it is centralized and its microservice `s0` acts as a throughput bottleneck, preventing other microservices from taking full advantage of cluster resources. This bottleneck effect is because `s0` has more external-services to call than the other microservices, which makes its computational load higher, as shown in Fig. 7b. In contrast, the topology D is highly distributed and Fig. 7b shows that no particular microservice acts as a bottleneck. Consequently, the application can make greater use of the cluster's computational resources, as shown in Fig. 7c, thus increasing throughput and reducing latency.

As for the Alibaba applications with topology A and B, both are rather distributed, but have microservices that act as bottlenecks (see Fig. 7c), which limit the use of cluster computing resources. Moreover, the application with topology B uses, on average, 25 microservices per request compared to the 5 used by the application with topology A. This implies a rather higher network overhead that significantly worsens the performance of B compared to A, even though both applications are exploiting the same amount of CPUs in the cluster.

### 4.3 Horizontal scalability

The previous sections have highlighted that the presence of bottleneck microservices exhausting their resource be-

fore others is critical for request throughput and latency. The presence of bottlenecks can be reduced by designing the application so that the service mesh is used in a more distributed way. Another solution is to provide more resources to bottleneck microservices by replicating them and implementing load-balancing strategies that fairly distribute service requests among replicas. This approach is called *horizontal scaling* and with k8s can be handled manually, or automatically using the Horizontal POD Autoscalers (HPAs) [46]. In this regard, we show how µBench can be used to analyze the pros and cons of autoscaling mechanisms by considering k8s HPA as a use-case. For the analysis, we used trace-driven benchmarks of the Alibaba application with topology A (Fig. 4a).

We recall that a k8s HPA linearly adjusts the number of "desired" replicas of a POD (i.e., a microservice) so that the workload of each POD in the replicas set is less than a desired value ($DesMV$). A typical workload metric is the CPU utilization percentage, measured as the ratio between the absolute CPU utilization ($U_{cpu}$) of the POD and the amount of CPU requested ($R_{cpu}$) for the POD by the user, which is a minimum value that k8s will guarantee. After the HPA has evaluated the desired number of replicas, the k8s scheduler tries to deploy them. This action is performed completely only if there are enough free resources, otherwise, the maximum number of replicas possible with the available resources is deployed. In fact, each node has an available CPU/Memory budget equal to its CPU/Memory capacity (4 CPUs, 16GB in our case) minus the sum of the CPU/Memory requested by PODs running on it.

The interplay between HPA and the scheduler requires proper configuration of requested resources for PODs to utilize cluster resources efficiently. Requiring more resources than a POD needs implies over-reserving resources on the node where the POD is running, possibly preventing other PODs to be deployed in the node and thus limiting the full utilization of the node's capacity. On the other hand, a request of fewer resources than a POD needs causes HPA to replicate it excessively, and such a large number of replicas unnecessarily consumes cluster resources, both because it overloads k8s load-balancing mechanism (e.g., many Linux IPtables rules) and because, in any case, each replica has an *idle* CPU and memory consumption.

To provide a proof of concept of this observation, we performed two types of experiments using HPAs for which the metric considered is the percentage of CPU utilization and its desired value is $DesMV = 70\%$.

In the first set of experiments, named *flat*, we chose a value $R_{cpu}(i) = R_{cpu}$ of CPU request equal for all microservice PODs. In the second set of experiments, named *tuned*, we used a simple tuning strategy to evaluate $R_{cpu}(i)$ for each $i$th microservice. Specifically, we deployed the application with one POD per microservice. We then generated a volume of user requests for which the maximum CPU consumption of all worker nodes does not exceed 50%. We took a snapshot of the CPU utilization $U_{cpu}(i)$ of the POD of the $i$th microservice and set $R_{cpu}(i) = U_{cpu}(i)/DesMV$. The milli CPUs required for services `s0, s4, s8, s29` turned out to be 1118m, 1385m, 1260m, 745m, respectively. As shown in Fig. 7b (topology A), the other microservices are practically unloaded and we set a minimum value $R_{cpu}(i) = 50m$ for them. The idea behind this tuning approach is that when the CPU utilization of the nodes is below a certain threshold that we set at 50%, the cluster is unloaded therefore the HPA should not replicate any microservices [7].

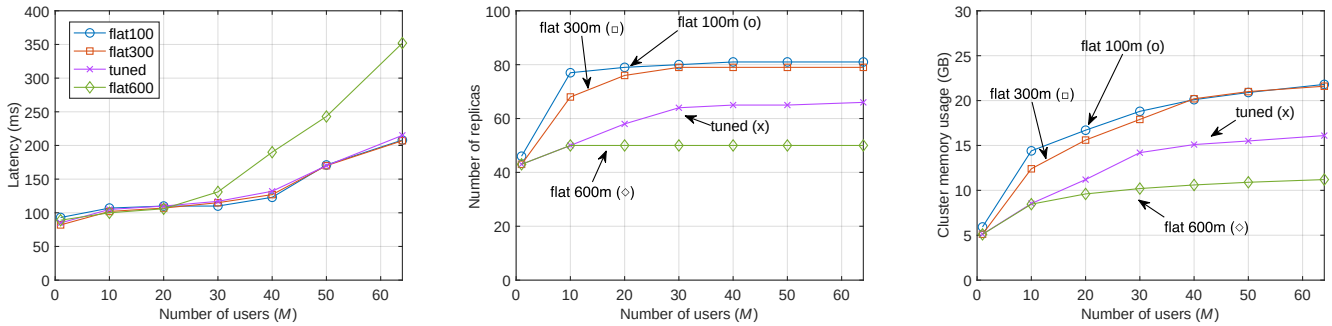
Fig. 9 shows the performance evaluation for flat experiments with a constant value of CPU request per POD of 100, 300 and 600 milli CPUs, and for tuned experiments. In all experiments, we set the maximum number of replicas per POD to 10, since we are using a cluster of 10 nodes. The flat experiments with 100 and 300 milli CPUs are representative of resource underprovisioning for critical microservices `s0, s4, s8, s29`. Consequently, the HPA highly increases the number of replicas of the critical services up to the limit of 10 replicas that we set. The number of replicas in the 300 milli CPU case tends to be a bit lower because the underprovisioning level is low. In both situations, the high number of replicas allows the application to take advantage of all the resources in the cluster, thus achieving low delay, but with a greater impact on memory. The case of 600 milli CPUs is representative of overprovisioning the resources of noncritical microservices in which too much CPU is reserved thus preventing critical microservices from scaling out enough. Consequently, the latency is higher, even though the memory footprint is smaller. As expected, the tuned configuration offers the best trade-off between latency performance and memory consumption.

## 5 Conclusions and lesson learned

µBench is an open-source tool for generating benchmark microservice applications that run in a Kubernetes cluster. Unlike demo applications that cannot be changed, applications created by µBench can be highly customized by varying key parameters, such as the service mesh, the number of microservices involved, and the resources used by them. This high configurability allows, on the one hand, the implementation of extensive sensitivity analyses of cloud solutions aimed at improving the execution of microservice applications and, on the other hand, a better understanding of the phenomena governing the performance of such applications to make informed choices at the design stage. As a proof of concept, we have used µBech in this regard, and the main lesson learned is as follows.

Microservice implementations exploit distributed resources of cloud infrastructures, thus reducing latency and improving throughput as compared to monolithic implementations, which are limited by the performance they can achieve in a single host.

7. In this paper we do not want to propose any scaling strategy but only to highlight through µBench the effect of fine-tuning of POD resources compared to a flat approach.




(a) Request latency vs the number of users (b) Number of POD replicas vs the number of users (c) Cluster memory usage vs the number of users

Fig. 9: Perfromance of the microservice applications using topology A in Fig. 4a, sequential downstream calls in a cluster of $K = 10$ nodes with k8s HPA

Distributing the application across multiple hosts inevitably introduces computational and network overheads that increase with the number of microservices involved. However, decomposing the application into too few microservices leads to not fully utilizing the cluster resources. Consequently, although the choice of how to decompose an application is motivated by heterogeneous reasons, in terms of performance, a viable choice is to decompose the application into the minimum number of microservices that allow full utilization of cluster resources. Moreover, whenever possible, it is better to make downstream requests in parallel to reduce latency, although this advantage tends to fade out as the load grows.

Another aspect that dramatically impacts performance concerns the presence of bottlenecks. Indeed, microservice applications can be likened to pipelines that process user requests. Therefore, their performance is limited by those microservices that behave as bottlenecks [47]. To reduce the presence of bottlenecks, we can optimize the application architecture, such as by choosing a more distributed service mesh; otherwise, we can scale out the resources of bottleneck microservices by increasing their number of replicas. However, the replication must be properly controlled to avoid unnecessary consumption of CPU or memory for running an excessive number of replicas. For k8s cluster, this is achieved with a fine configuration of the resource requested by microservice PODs.

## Acknowledgment

This work is supported in part by the Italian MIUR PRIN Liquid_Edge project and PNRR RESTART program.

## References

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[2] J. Thönes, "Microservices," *IEEE software*, vol. 32, no. 1, pp. 116–116, 2015.

[3] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[4] X. Feng, J. Shen, and Y. Fan, "Rest: An alternative to rpc for web services architecture," in *2009 First International Conference on future information networks*. IEEE, 2009, pp. 7–10.

[5] A high performance, open source universal rpc framework. [Online]. Available: https://grpc.io/

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[7] P. Dobbelaere and K. S. Esmaili, "Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper," in *Proceedings of the 11th ACM international conference on distributed and event-based systems*, 2017, pp. 227–238.

[8] A. Singleton, "The economics of microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 16–20, 2016.

[9] D. Merkel *et al.*, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[10] Kubernetes (k8s): Production-grade container orchestration. [Online]. Available: https://kubernetes.io/

[11] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site reliability engineering: How Google runs production systems*. " O'Reilly Media, Inc.", 2016.

[12] Prometheus open-source monitoring solution. [Online]. Available: https://prometheus.io/

[13] Jaeger: open source, end-to-end distributed tracing. [Online]. Available: https://www.jaegertracing.io/

[14] Istio - simplify observability, traffic management, security, and policy with the leading service mesh. [Online]. Available: https://istio.io/

[15] C. Guerrero, I. Lera, and C. Juiz, "Genetic algorithm for multi-objective optimization of container allocation in cloud architecture," *Journal of Grid Computing*, vol. 16, no. 1, pp. 113–135, 2018.

[16] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach," *IEEE Transactions on Cloud Computing*, 2020.

[17] F. Rossi, V. Cardellini, and F. L. Presti, "Self-adaptive threshold-based policy for microservices elasticity," in *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2020, pp. 1–8.

[18] C. K. Rudrabhatla, "A quantitative approach for estimating the scaling thresholds and step policies in a distributed microservice architecture," *IEEE Access*, vol. 8, pp. 180 246–180 254, 2020.

[19] C. T. Joseph and K. Chandrasekaran, "Intma: Dynamic interaction-aware resource allocation for containerized microservices in cloud environments," *Journal of Systems Architecture*, vol. 111, p. 101785, 2020.

[20] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, pp. 122–1225.

[21] N. Kratzke, "About microservices, containers and their underes-

timated impact on network performance," in *CLOUD COMPUTING*, 2015.

[22] L. Funari, L. Petrucci, and A. Detti, "Storage-saving scheduling policies for clusters running containers," *IEEE Transactions on Cloud Computing*, 2021.

[23] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. 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)*. IEEE, 2018, pp. 223–236.

[24] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 3–18.

[25] Sock shop: Microservices demo. [Online]. Available: https://microservices-demo.github.io/

[26] Istio bookinfo application. [Online]. Available: https://istio.io/latest/docs/examples/bookinfo/

[27] µBench GitHub website. [Online]. Available: https://github.com/mSvcBench/muBench

[28] DeathStarBench. [Online]. Available: https://github.com/delimitrou/DeathStarBench

[29] Online boutique. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[30] R. Jung and M. Adolf, "The jpetstore suite: a concise experiment setup for research," in *Symposium on Software Performance*, 2018.

[31] Distributed version of the spring petclinic - adapted for cloud foundry and kubernetes. [Online]. Available: https://github.dev/spring-petclinic/spring-petclinic-cloud

[32] A. Sriraman and T. F. Wenisch, "$\mu$ suite: a benchmark suite for microservices," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 1–12.

[33] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi, "Benchmark requirements for microservices architecture research," in *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*. IEEE, 2017, pp. 8–13.

[34] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović, and A. van Hoorn, "Microservices: A performance tester's dream or nightmare?" in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 138–149.

[35] S. Caculo, K. Lahiri, and S. Kalambur, "Characterizing the scale-up performance of microservices using teastore," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 48–59.

[36] V. Rao, V. Singh, K. Goutham, B. U. Kempaiah, R. J. Mampilli, S. Kalambur, and D. Sitaram, "Scheduling microservice containers on large core machines through placement and coalescing," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2021, pp. 80–100.

[37] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *2016 IEEE international symposium on workload characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[38] Acme air in nodejs. [Online]. Available: https://github.com/acmeair/acmeair-nodejs

[39] Cluster trace microservices v2021. [Online]. Available: https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021

[40] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 412–426.

[41] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, J. He, and C. Xu, "An in-depth study of microservice call graph and runtime performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3901–3914, 2022.

[42] V. Podolskiy, M. Patrou, P. Patros, M. Gerndt, and K. B. Kent, "The weakest link: revealing and modeling the architectural patterns of microservice applications," in *30th Annual International Conference on Computer Science and Software Engineering*. ACM, 2020.

[43] J. Gibbons, "Unbounded spigot algorithms for the digits of pi," *American Mathematical Monthly*, vol. 113, 06 2004.

[44] Apache jmeter. [Online]. Available: https://jmeter.apache.org/

[45] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*. IEEE, 2018, pp. 000 149–000 154.

[46] Horizontal pod autoscaling. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[47] G. Gill, V. Gupta, and M. Singh, "Performance estimation and slack matching for pipelined asynchronous architectures with choice," in *2008 IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 2008, pp. 449–456.

**Andrea Detti** is a professor of Wireless Networks and Cloud Computing at the University of Rome "Tor Vergata". His current research activity spans on different topics in the area of computer networks and cloud computing. He is co-author of more than 80 papers on journals and conference proceedings, and participated to several EU funded projects with coordination and research roles.

**Ludovico Funari** is a researcher at the University of Rome Tor Vergata. He received the master's degree in "ICT And Internet Engineering" in October 2019. His research activity includes IoT, Cloud and Edge computing. He has worked as a CNIT (Italian National Inter-University Consortium for Telecommunications) researcher for the EU H2020 "Fed4IoT" project. He is currently working for the "Liquid_Edge" MIUR research project at Univ. of Rome Tor Vergata.

**Luca Petrucci** received the Master's Degree in Computer Science Engineering in April 2019, from the University of Rome Tor Vergata. He is a PhD student at the Univ. of Rome Tor Vergata. From April 2016 to December 2019 he is a researcher at CNIT (Italian National Inter-University Consortium for Telecommunications), where he had developed his bachelor thesis and master thesis, respectively concerning the EU projects BEBA and 5G-PICTURE.