Automated performance prediction of microservice applications using simulation

Clément Courageux-Sudan, Anne-Cécile Orgerie, Martin Quinson

▶ **To cite this version:**

Clément Courageux-Sudan, Anne-Cécile Orgerie, Martin Quinson. Automated performance prediction of microservice applications using simulation. MASCOTS 2021 - International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, Nov 2021, virtually, France. pp.1-8, 10.1109/MASCOTS53633.2021.9614260 . hal-03389508

HAL Id: hal-03389508
https://hal.science/hal-03389508

Submitted on 21 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Automated performance prediction of microservice applications using simulation

Clément Courageux-Sudan*, Anne-Cécile Orgerie*, Martin Quinson*
*Univ. Rennes, Inria, CNRS, IRISA, France
Email: {clement.courageux-sudan, anne-cecile.orgerie, martin.quinson}@irisa.fr

***Abstract*—Microservices transform monolithic applications into simple, scalable, and interacting services. It allows for faster development and fine-grained deployments. However, the cooperation of several services leads to intricate dependencies, hindering the detection of performance bottlenecks. Current microservice performance analysis methods require real deployments, a costly process both in time and resources, while performance prediction through simulation relies on models that are complex to develop and instantiate. In this paper, we propose a microservice performance analysis approach based on simulation. Our contribution first introduces a microservice performance model requiring few instantiation parameters. We then propose a methodology to automatically derive model instantiation values from a single execution trace. We evaluate this methodology on two benchmarks from the literature. Our approach accurately predicts the deployment performance of large-scale microservice applications in various configurations from a single execution trace. This provides valuable insights on the performance of an application prior to its deployment on real platform.**

***Index Terms*—Microservice, Modeling and simulation, Performance evaluation, Web services**

I. Introduction

Microservices allow for splitting large and complex applications into sets of simple, independent, and scalable services. Microservice architecture has many advantages over monolithic applications: separation of concerns allows disjoint teams to develop independently part of an application without complete system knowledge. When deployed, microservices run in separate lightweight containers, allowing for better reliability and adaptation to variable workloads thanks to fault detection mechanisms and autoscaling policies at the scale of individual services. Compared to microservices, monolithic applications have to be managed as a whole, reducing their scalability. As a result, microservices are used by some of the largest internet actors, such as Twitter, Netflix, and Uber [21].

Despite their advantages, microservices require complex interactions to fulfill requests. This makes the analysis of microservice applications on real infrastructures more complex than monolithic applications. Microservice applications can be composed of hundreds of services and serve huge workloads. Because of this complexity, optimizing the deployment settings of microservices is challenging: Given an application and a constrained infrastructure budget, one must evaluate and answer the following questions to obtain the best performances: **Q1.** How will the execution times of a microservice react to a variable load? **Q2.** How would a CPU upgrade improve the maximum sustainable load? **Q3.** Will distributing microservices on more than one node increase performances? **Q4.** How to optimize the location of services in a computing cluster to obtain the best performances?

Currently, microservice developers answer such questions by deploying their applications on real platforms and monitoring their performances. This requires a combination of several tools. First, the application is deployed using container orchestration systems, such as *Kubernetes* or *Docker-swarm*. Using them allows defining resource and location constraints on services, so that the experimenter can deploy the application in a specific configuration. The deployed application then has to be benchmarked to obtain information on its performance in this setup. Gathering and processing applications metrics can be done through the use of *Distributed Tracing*. This approach is used in most microservice applications [16], allowing to obtain both service-specific and end-to-end execution metrics. Standards such as *OpenTracing* [20] and *OpenTelemetry* [19] help developers at instrumenting their application, while trace visualization tools like *Jaeger* [14] and *ZipKin* [26] unify the process of observing application bottlenecks.

Using orchestration systems and distributed tracing tools, microservice developers can observe whether a set of deployment settings meets their *QoS* requirements. However, this approach requires combining complex tools, a costly process both in time and resources. Also, the process must be repeated for all configurations. Obtaining *a priori* performance estimations for various configurations without real deployments would be useful. In this paper, we focus on simulation techniques adapted to microservice performance studies. Our analysis of previous contributions about microservice simulation shows a costly process to transpose applications into their simulated twins. There is a need for a methodology to ease performance simulation of real applications.

Our contributions are the following:

- We propose a model for microservice-based applications. This model is simple enough to be calibrated without extensive work while being complete enough to simulate real applications.
- We propose a methodology to instantiate our model and accurately study the performances of real applications.
- We validate the accuracy and scalability of our approach using a set of microbenchmarks as well as real use cases based on existing microservice benchmarks.

Section II introduces our microservice abstraction. Section III describes a methodology to simulate real applications

by using our service model. We validate our contribution in Section IV while Section V compares our contribution to other approaches from the literature. Finally, Section VI concludes this work.

II. A Simple Microservice Model

Modern applications are composed of many services interacting together to fulfill requests. To understand the performance of a complete microservice application, one must first understand the behavior of single, isolated microservices. A microservice offers a well-defined interface, constantly listening for incoming requests. When a request is received, it triggers internal functions computing a result that is then returned to the initiator of the request.

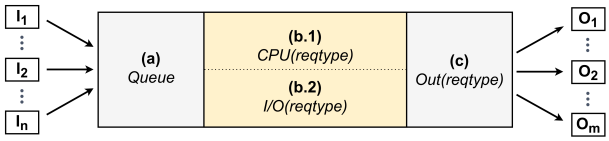

Fig. 1. Intra-service execution pipeline.

We introduce a model representing microservice request executions. Our design goal for this model is to be as simple as possible to enable fast calibrations while being accurate enough to represent real microservices. We model the execution of requests within a microservice using a simple 3-stages pipeline, as shown in Figure 1. A request executed by a service spends some time in each execution stage depending on both the state of the service when the request is received (queuing, resource overload, etc.), and intrinsic service properties such as its degree of parallelism. The total execution time of request r in service s is given by:

$$D_{exec}(r,s) = d_{queue}(r) + d_{CPU}(r) + d_{IO}(r) + d_{comm}(r)$$

where $d_{queue}(r)$ is the queuing time of request r before starting its execution, $d_{CPU}(r)$ is the CPU time to execute request r, $d_{IO}(r)$ the amount of time waiting for I/O operations, and $d_{comm}(r)$ the time spent communicating with external services. We now detail these factors.

(a) Queuing time: When a request is sent to a service, it can experience variable queuing times depending on the service's state. Most services limit their maximum amount of concurrent requests to avoid resource overloads and performance degradation due to system's context switches. In our model, a service comprises a waiting queue where incoming requests are stored until their execution starts. The time spent by a request in the reception queue is dependent on both the input load and the scale of the deployed application. Through vertical scaling, the service is deployed on more efficient resources, leading to reduced execution times during step **(b.1)**, and an increased maximum load capacity. Horizontal scaling does not improve the execution time of single requests, but allows for more requests to be executed in parallel through the use of several service instances in parallel during step **(b)**.

(b.1) CPU usage: A request can start being executed once a free execution slot is available in an instance of the service. We describe the execution of a request by its CPU usage: the machine executing the request has a limited amount of CPU resources (in *flops/s*) and shares them between all active requests in one of the execution slots of the host. The time spent processing a request $d_{CPU}(r)$ is dependent on both the cost of executing r and the amount of requests that are executed concurrently. The cost associated to a request execution further depends on its type. A single microservice can offer more than one function through its interface, each of them leading to different execution times. Because of this, the CPU usage of a service is described by the following values: the provisioned capacity of the executing node, a mapping of request type to CPU costs, and the maximum amount of concurrent executions.

(b.2) I/O idle time: A service execution does not only consist in CPU processing but also in I/O operations. In some cases, I/O can be overlapped with CPU executions. In other cases, I/O can result in periods where the CPU remains *IDLE*. We define the time spent in I/O by using an *active ratio* that represents the time spent doing I/O compared to pure CPU execution. Currently, we model I/O as a simple delay, thus we do not take disk contention into account.

(c) Service output: Most microservices request data from other services to compute a final result. Using our model, a request can be forwarded to other services once it has finished both executing the request during **(b.1)** and waiting for I/O to finish in **(b.2)**. The type of the request defines the list of services to be invoked. If the output services called during this step are running on different computing nodes than the current service, a network communication is initiated to forward the request. This enables the observation of performance bottlenecks due to network limitations. Microservices linked through inter-service communications form a *Directed Acyclic Graph (DAG)* for each request type, where nodes are service executions and edges communications between services. As shown in Figure 1, we separate the execution of a request from outer communications. It is a simplification compared to real microservices that often interleave executions and communications. Our approach does not represent the execution of a request at such a fine grain but conserves the overall execution time as well as communication dependencies.

This model splits the execution of a microservice request in no more than the 3 steps described above. Table I gives a summary of the values used to instantiate our model.

We implemented our model on top of the SimGrid simulation framework. Our implementation (available online [1]) allows to define a microservice application using a simple interface with the parameters from Table I. It can be used to manually define real applications in order to study and evaluate their performances in various configurations. It is also possible to define autoscaling policies, dynamically adapting the amount of resource of each service depending on the

[1] https://github.com/klementc/microservices_simgrid_reproducibility

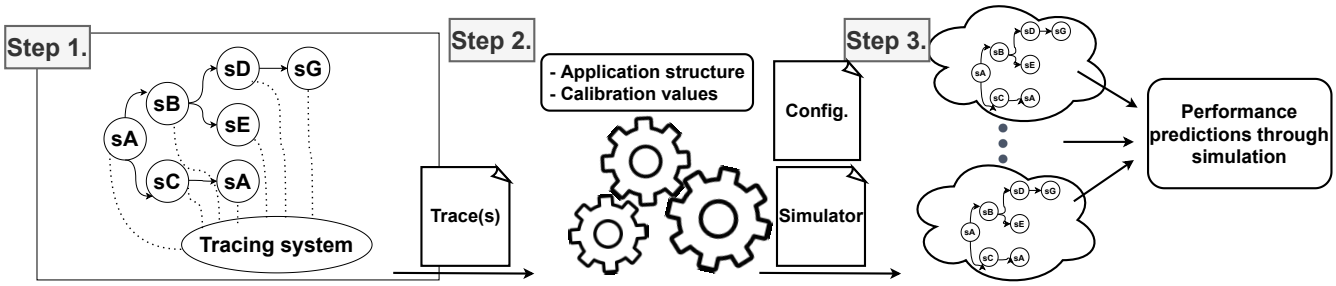

Fig. 2. Transformation of an execution trace in an executable simulator.

TABLE I
SUMMARY OF INTER-SERVICE AND INTRA-SERVICE PROPERTIES USED TO CALIBRATE OUR MICROSERVICE MODEL.

Granularity	Parameters
Request	- Type of the request
Intra-service properties	- CPU costs - I/O ratios - Parallelization degree
Inter-service properties	- Output services (a DAG for each type of request) - Network requests sizes

input load. Even if possible, the manual instantiation of the models for all services of large applications with the correct values remains a challenging task. Both service interactions and individual service behaviors need to be taken into account for possibly hundreds of services. We thus introduce a methodology to simplify the transformation of real applications into simulation models.

III. MODELING REAL MICROSERVICE APPLICATIONS

We propose a methodology to simulate the performance of real microservice applications without extensive efforts. Previous works on microservice simulation such as [25] recommend modifying the code of the application to log additional information that are used as calibration values. However, manual code modification remains burdensome and error-prone for large applications. It is also not evolutive since it has to be made again when the structure of the code is modified. Our goal is to allow developers of microservice applications using state-of-the-art service monitoring techniques to automatically obtain both the description of the structure of their application along with the calibration values for our microservice model. Our approach does not require code modifications provided that the targeted application uses one of the standard service monitoring solutions, as detailed hereafter.

To instantiate our microservice model, we need to gather values for the parameters given in Table I. The *request types* to study depend on the application and on the goal of the experiment. One can study either a specific request or observe several types of request executable in the application.

Intra-service properties need to be observed at both application- and system-levels. Indeed, overall service execution times can be observed easily from the application, but the ratio of active and *IDLE* CPU times requires lower-level information.

Inter-service information can be obtained by observing the network interactions between services running in separate containers.

The required accuracy for the calibration values depends on the target of the simulation. For example, approximate values of network request sizes can be adequate for an experiment studying the performance of an application under limited computing resources, as the network does not constitute the bottleneck of the experiment.

The model parameters can be obtained via various methods. From our experience, *distributed tracing* can provide these values at a low cost. Most real microservice applications are already instrumented to export metrics and information about the state of each of their services. Whereas the typical use case of distributed tracing is to help identifying the services inducing performance issues, leveraging it to calibrate our model is an interesting solution. Traces entails the path followed by requests and the amount of time spent in each service. Each service execution is called a *span*, and the set of all spans linked to the same request forms the *DAG* of the overall execution. The collected data provide both inter-service property DAGs and intra-service execution times. In the following, we use the information contained in these traces to instantiate our microservice models.

As summarized in Figure 2, our goal is to build a simulator exploiting the information contained in a single execution trace per type of studied request. The resulting simulator can then predict the performance of the application under various configurations. Figures 3 and 4 show an execution trace automatically extracted from an application of the literature [9]. In Figure 3, a partial execution trace can be observed, directly taken from the web interface of Jaeger [14], whereas Figure 4 is a DAG representation of the same request. Based on this example, we now show how to calibrate our model by following the steps of Figure 2.

Step 1: Obtain calibration traces. The application modeler first needs to gather execution traces of the requests to study. These traces are created upon request executions, once the application is instrumented with a tracing system such as *OpenTracing* [20], *OpenTelemetry* [19], or *Kieker* [24]. The user can then run the instrumented application on a machine

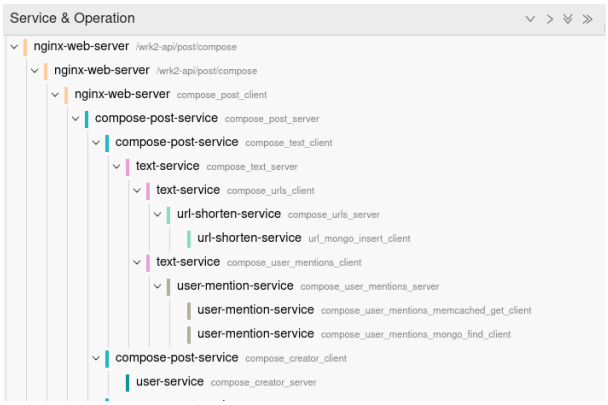

Fig. 3. Part of a trace from the execution of DeathStarBench's [9] social network COMPOSE request extracted from Jaeger [14]. Services implied in the execution of the request as well as the relation between services can be observed.

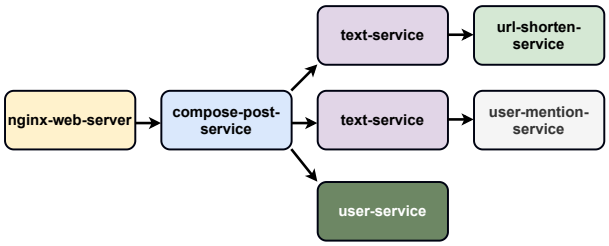

Fig. 4. Graph representation of the request sample from Figure 3.

representative of the targeted platform. Finally, the traces are used to calibrate our model. They contain information that will be used in the following steps. A particular care must be taken in the selection of the traces to ensure that they are representative of the behavior of the application. Using average execution times helps at avoiding outliers. In the case of Figure 3, the application from DeathStarBench [9] is natively instrumented with *OpenTracing* and we run it with a medium workload for several minutes to ensure that we select a trace corresponding to the application's behavior in steady-state: none of the services are overloaded, the caches are warm, databases are populated, etc.

Step 2: Process the traces to obtain calibration values. The second step consists in obtaining the values required for the instantiation of our microservice model. Tracing systems used in Step 1 are usually combined with generic trace processing tools such as *Jaeger* [14], *Zipkin* [26], or Kieker's tools [24]. These tools collect application traces and process them to obtain metrics about the performance of end-to-end executions. They provide enough information to calibrate our model even if low-level metrics may be missing. As shown in Figure 4, using Jaeger, we obtain from the OpenTracing trace a DAG of services execution connected to each other. Thus, we can infer how services interact during request executions. The size of network requests is not provided in this case, but an instrumentation with *OpenTelemetry* [19] could for example allow us to send request sizes to *Prometheus* [22]. Intra-service properties are also partially included in our trace. We obtain an accurate estimation of the duration of the request for each service execution. Still, our example trace must be extended regarding the ratio of I/O. Complementary solutions such as Docker monitoring could be used to retrieve I/O ratios.

Step 3: Building and configuring the simulator. The values obtained during the previous step allow instantiating one microservice model for each service of the target application. We can then observe the duration of end-to-end request executions through simulation. Simulations can be configured to explore the performance of the application under various deployment settings. Because obtaining calibration values is relatively easy thanks to existing monitoring systems, it becomes possible to adjust calibration values after the modification of a service's implementation.

To profit from the ability of distributed tracing at providing the intra and inter-service properties of an application, we designed a tool to automatically produce the code of a simulator with calibrated microservices given Jaeger traces. This allows for a semi-automatic performance simulation of compatible instrumented applications.

IV. Experimental Validation

To evaluate the accuracy and scalability of our contribution, we rely on both a set of microbenchmarks and published microservice benchmarks use cases. We compare the performance predicted by models instantiated using our approach to real application executions. Based on those results, we show that our contribution can help microservice performance analysis by answering the four questions we stated in Section I.

Experimental setup: All experiments have been launched on the Grid'5000 testbed [2]. We run our experiments on a cluster composed of nodes with 2 $\times$ *Intel Xeon E5-2630 v3* with 8 cores/CPU, 128 GiB of memory, and 2 $\times$ 10 Gbps network interfaces. They run Debian 10 under kernel *4.19.0-16-amd64*. Services run within *Docker containers* and multi-node deployments are done with *Docker-swarm*.

Source code and reproducibility: The source code of our contribution is available online at https://github.com/klementc/microservices_simgrid_reproducibility along with the scripts used to obtain the results. We also provide notebooks with the code used to generate the figures of this paper and additional experimental results not presented here.

A. Microbenchmarks

Before predicting the performance of real large microservice applications, we need to ensure that our microservice model allows for accurate execution time predictions at the scale of single services. This first experiment aims at showing the ability of our microservice model to reproduce request execution times accurately under a dynamic load.

We launch a microservice application that executes a fixed amount of CPU work for each request. The microservice fetches incoming requests through a RabbitMQ queue, and it is possible to chain multiple services to obtain multi-steps executions. The results detailed hereafter consist of a

single service sending its results directly to a sink. We ran microbenchmarks using 2 chained services leading to similar results but not shown here due to space constraints.

To simulate this application using our model, we proceed in two steps. First, we obtain the execution time of a request given its CPU cost by sending calibration requests. These calibration values represent the execution duration when a service is not overloaded. Through linear regression, we estimate the duration of request executions for any CPU cost assigned to the service. We then calibrate the service model to use these values before running simulations and comparing the results to an execution on a real platform.

We generate a synthetic load using LIMBO, an HTTP load model and generator [23]. It consists of requests spanning over 5 minutes with between 1 and 40 requests per second and three activity spikes for a total amount of around 4,500 requests as shown in Figure 5. We launch this experiment 5 times for each configuration. Configurations vary by the quantity of work to be executed and the maximum amount of parallel executions.

Figure 6 shows the results obtained during the microbenchmark execution with one service, an execution time of approximately 25ms per request and a concurrency degree of either five or ten. The concurrency degree refers to the maximum number of parallel requests for a given service. For each request executed during the experiment, Figure 6 shows the estimated execution time of request obtained using our model in SimGrid and the execution times of a real deployment.

We make two observations. First, both SimGrid predictions and real-world results have higher execution times during request arrival spikes, which happen at 20s, 150s, and 250s. This is caused by the processing of several requests in parallel and queuing. This shows the ability of our service model to reproduce processing times of requests under a dynamic load.

Second, we observe that the execution time per request changes with the concurrency degree of the application. With a concurrency degree of 5, the maximum request execution time (125ms) is approximately two times lower than the maximum request execution time with 10 parallel requests. A smaller concurrency degree decreases single requests execution times at the cost of increased queuing times. Executing an important amount of parallel requests on a single CPU core will also lead to overheads due to operating system thread switching. The execution model of SimGrid does not take into account context switching costs, thus it might underestimate execution

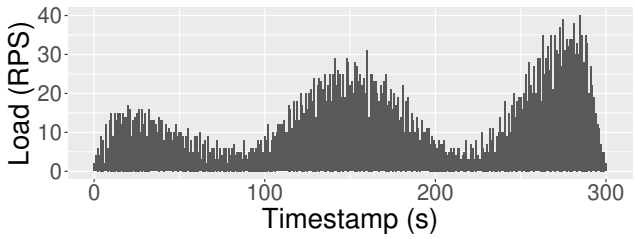

Fig. 5. Synthetic load generated for microbenchmark experiments: Requests Per Second (RPS) during 5 minutes with 3 load spikes.

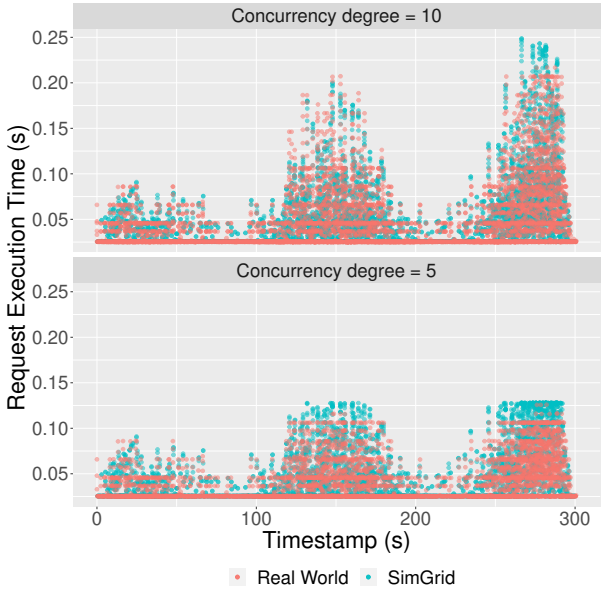

Fig. 6. Comparison between our model's prediction and a real execution with two different concurrency degrees under a synthetic load.

times when the number of parallel requests is much higher than the number of CPU cores to execute them. Yet, in the results of this paper, this effect does not impact our observations.

These results show that our microservice model implementation in SimGrid can accurately predict the performance of simple CPU-intensive microservices under variable loads, and thus it answers the first question:

Q1. How will the execution times of a microservice react to a variable load?

B. Use-case 1 : TeaStore login requests

We now observe the performance when modifying the resources (i.e. number of cores) dedicated to the execution of an application within a single computing node. This question is of importance for real deployments to estimate the cost/gain ratio of different hardware options. Our goal is to obtain the same results between real observations and our simulations when changing the number of resources to be used. To evaluate the versatility of our approach, we only rely on one calibration experiment detailed hereafter to instantiate our service models.

For this evaluation, we run TeaStore [8], a microservice application benchmark used in microservice performance evaluation literature [11]. We focus on the *LOGIN* request of this application. This request involves 4 different services running in separate Docker containers. We study the maximum sustainable load (in *Requests Per Second, RPS*) of the application under different resource configurations and compare real results to simulated predictions.

TeaStore is natively instrumented with Kieker [24]. We use the average request execution duration of each service to calibrate our service models within SimGrid by doing a real execution under a low load-profile. From this execution we extract a trace with Kieker, providing us the calibration values.

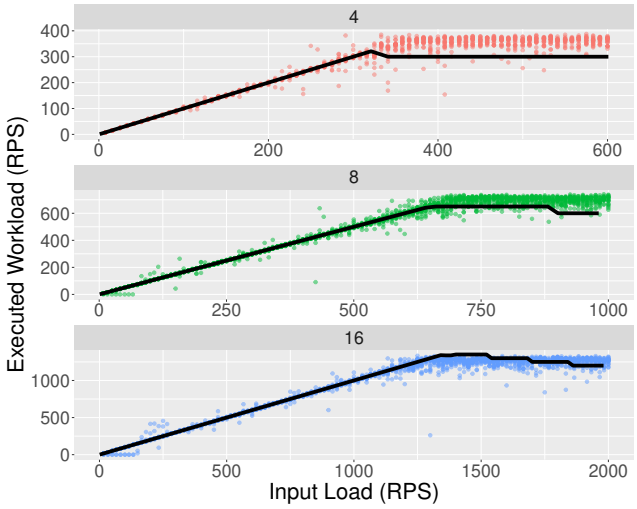

Fig. 7. Comparison of teastore's LOGIN performance between SimGrid predictions (black line) and real world executions (colored dots) for 4, 8 and 16 cores dedicated to the execution of the application.

As recommended by TeaStore's documentation, the load is generated by LIMBO [23]. We benchmark the application under 3 different configurations. For the real experiments, the application is deployed on a machine with either 4, 8, or 16 cores dedicated to the execution of the services. We execute each configuration 20 times. A summary of the results is shown in Figure 7.

The goal of this experiment is to detect the breaking point after which the application is overloaded. This breaking point is detected by our model, for example around 320 requests per second in the 4 core configuration, as shown in Figure 7. Regarding the accuracy of our model, we observe the *mean relative error (MRE)* between SimGrid predictions and real world values. Over the different workloads shown in Figure 7, we observe an average MRE of 11.8%, 4.9% and 3.6% with maximum values of 21%, 17% and 14.9% in the 4, 8 and 16 cores configurations respectively. While the maximum error observed is non-negligible, especially in the 4 cores configuration, the predictions of our model allow observing trends, and comparing the advantages of one configuration over another. This experiment allowed us to answer the second question:

Q2. How would a CPU upgrade improve the maximum sustainable load?

C. Use-case 2 : DeathStarBench's social network

The next step consists in evaluating the performance of an application taking advantage of several physical servers, one of the main assets of the microservice architecture. Yet, finding the best partitioning of services is a complex task.

We chose to study one of the most realistic published microservice benchmarks to our knowledge, the social network from DeathStarBench [9]. We reproduce the most complex request of this application, the *COMPOSE* request that submits a publication to the social network. It consists of more than 30 spans across 12 different services. We deploy the social network using *Docker-swarm*, and vary the location of each service and the number of replicas. We compare the maximum sustainable request throughput obtained with our simulator against real executions with 10 real runs per configuration.

TABLE II
SERVICE CONFIGURATIONS FOR THE SOCIAL-NETWORK APPLICATION.

	Node 1	Node 2
Config 1	nginx_web_server, compost_post_service, unique_id, media_service, user_service, text_service, user_mention, home_timeline, social_graph, user_timeline, post_storage_service, url_shorten	
Config 2.a	nginx_web_server, compose_post, unique_id, user_service, text_service, user_mention, home_timeline, social_graph, url_shorten, user_timeline	media_service, post_storage_service
Config 2.b	nginx_web_server, compost_post, home_timeline, social_graph, user_timeline, post_storage_service	unique_id, media_service, user_service, text_service, user_mention, url_shorten

Table II shows the server allocations for the microservices required to execute the *COMPOSE* request in each of the three studied configurations. With configuration 1, all services are using the resources of a single node. It should be the least efficient configuration due to fewer resources available for each service. Configurations 2.a and 2.b divide the 12 services into two randomly selected groups, each running on different nodes. For all configurations, other services of the application, not involved in executing the *COMPOSE* request, are running on a separate node not considered here. The SimGrid calibrated simulator is generated using the Jaeger trace partially shown in Figure 3 and the code generation script described in Section III. We run the experiment with various CPU constraints. Figure 8 shows the result obtained with 10 cores per node to execute the application.

We observe one limitation of our approach during this experiment as our model does not capture the communication overhead due to *Apache Thrift* under high load. We choose to reduce this overhead (that is known to the authors of [9]) by executing two instances of each service, as would be done in a real deployment to improve the application throughput. Such fine-grain overheads could be considered in future work.

Figure 8 shows the maximum request throughput estimated by SimGrid and obtained through real-world executions. We observe that SimGrid accurately detects the breaking point where the application becomes saturated, at around 1,500 RPS with *1×10 cores* for configuration 1, and 1,750 RPS and 2,200 RPS under configurations 2.a and 2.b with *2×10 cores*. We observe similar results when comparing the average latency of requests. A non-proportional maximum throughput between the configurations can be observed. Indeed, configuration 2.a presents an unbalanced grouping of services among the nodes

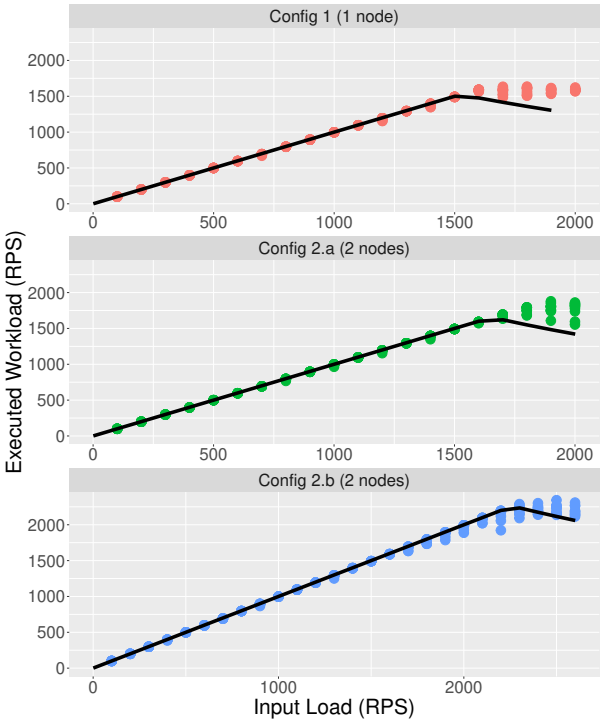

Fig. 8. Comparison of social network COMPOSE request between SimGrid predictions (black line) and real world observations (colored dots) with 10 cores per node and either 1 (config. 1) or 2 (config. 2.a and 2.b) nodes.

which leads to the overloading of one node while the other is still able to process requests. A very naive service model would predict an improvement factor of two between the configurations 1 and 2.a, and would predict identical performance for configurations 2.a and 2.b. Since our model takes into account the processing cost of each service, it is thus able to show that configuration 2.b performs better than configuration 2.a, as in the real executions.

Regarding the accuracy of our predictions, the configurations 1, 2.a and 2.b have an average MRE of respectively 3.5%, 3.7% and 1.4%, whereas the maximum measured errors are 21%, 20% and 6.6%. The maximum error values can be explained, as with the *TeaStore* experiment, by a different behavior of our model compared to the real application once the breaking point is reached. Before that point, the error remains very low between the simulated predictions and real measurements, while the breaking point is observable in Figure 8, at 1500, 1700 and 2250 requests per second in scenarios 1, 2.b and 2.b.

These experimental results show that our approach, based on a microservice model and code tracing tools, provides accurate estimations of the performance of microservice applications with different configurations, thus answering both questions:
Q3. Will distributing microservices on more than one node increase performance?
Q4. How to optimize the location of services in a computing cluster to obtain the best performance?

V. Related Work

To improve the performance of their applications, microservice developers can, among others: use automatic scaling policies based either on system thresholds [13] or application metrics [10], detect and correct application faults [7] or optimize service placements [1].

To evaluate the advantages of using such methods, two main approaches exist in the literature around microservice performance evaluation. On one hand, real-world experiments [4], [13]. Despite giving meaningful insights on the behavior of an application, because of time and real platform constraints, these methods often limits the validity of their results to a narrow set of contexts. They also do not allow for easily reproducible results. The second approach uses purely analytical models [3], [7]. This approach allows observing some theoretical properties of an application but is not adapted to study complex applications composed of many services.

To overcome the lack of evaluation in various contexts through real experiments and the difficulty of adapting formal methods to real systems, a third approach resorts to simulation. With simulation, one can observe the behavior of an application under several configurations without real deployments and reproduce the obtained results using only commodity hardware. We propose a brief overview of existing simulation software for microservice applications and their limits.

The authors of BigHouse have obtained performance predictions of cloud applications using statistical methods based on a discrete-event simulator [18]. Unfortunately, more recent works have shown that the accuracy of BigHouse is limited when applications are composed of several services due to its very high-level representation of applications, not taking into account some specificities of microservices [25].

Based on the limits of BigHouse, the authors of [25] used a finer-grained representation of microservice applications with μqsim. μqsim allows for a detailed representation of internal microservice executions using sets of execution stages for each type of request a microservice can receive along with communication dependencies between the different services. The relation between services is modeled as a DAG where nodes are services and edges represent the path followed by individual requests processed by the application. With a correct calibration of the different execution stages of each service and their interactions for different types of requests, the authors managed to reproduce the behavior of a complex microservice benchmark [9]. With μqsim, the accuracy of the simulation depends on the description of the different processing stages inside microservices and of service dependencies. Whereas this calibration can be done easily by hand with small applications, the lack of a proper calibration methodology leads to a tedious and error-prone process with large applications. This approach also requires to re-instrument a service each time its code is modified in order to fit its new behavior. Compared to μqsim, our approach introduces a simplified representation of intra-microservice processing by using a fixed set of execution stages (i.e. 3 stages as shown on Figure 1). But, these stages

are automatically calibrated using standard execution traces obtained through well-known distributed tracing systems.

Other tools allow for microservice simulation studies. Simulators dedicated to *Fog and edge computing systems* often model their applications as DAGs of tasks, like YAFS [15], fogTorch [5], and IFogSim [12]. Yet, these simulators are very specific to the context of *Fog*, and still require to manually describe most of the dependencies and properties of the simulated applications. For instance, the authors of [6] provide a code generation pipeline with Fog specific properties, such as geographical coordinates, and Fog applications, such as dataflow processing which do not correspond to microservice architectures although both employ DAG representations.

Evaluating the performance of an application based on execution traces is not a new approach [17]. Our approach is a new step towards the seamless transposition of real applications into simulations. Similar approaches require manual descriptions of intra-service and inter-service properties that is both costly and error-prone, especially for large applications.

VI. Conclusion and Future Work

Microservice applications trade monolithic complexity for intricate interactions between simple services, hindering the system performance evaluation.

In this article, we introduced a microservice simulation model based on a reduced number of calibration values to describe microservice applications. Our contribution is more precise than large grain models, while being easier to instantiate than precise models. We proposed a methodology leveraging distributed tracing systems to instantiate the simulation models of real applications using standard instrumentation solutions.

We implemented this model on top of SimGrid, and applied our methodology on applications instrumented with Jaeger and Kieker. These contributions were evaluated on microservice benchmarks, demonstrating their ability to answer the operational questions **Q.1-4** introduced in Section I on such applications. This could be used in various what-if analyses such as the exploration of performance trade-offs under scarce resources that are common in fog infrastructures. More interestingly, it could even be used to dimension a Fog infrastructure given an application and a workload to serve, an intractable problem with other solutions.

In the future, we will provide models of features that are common among microservice applications, such as the *Kubernetes* autoscaling policies. We also plan to give access to detailed resource usage values, including energy dissipation, through classical instrumentation systems.

Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] H. Arkian, G. Pierre, J. Tordsson, and E. Elmroth, "Model-based stream processing auto-scaling in geo-distributed environments," in *Int. Conf. on Computer Communications and Networks*, 2021.
[2] D. Balouek *et al.*, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, 2013, vol. 367.
[3] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Trans. on Parallel and Distributed Systems*, vol. 30, no. 9, 2019.
[4] A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, and S. Kounev, "Chamulteon: Coordinated auto-scaling of micro-services," in *IEEE Int. Conf. on Distributed Computing Systems*, 2019, pp. 2015–2025.
[5] A. Brogi, S. Forti, and A. Ibrahim, "How to best deploy your fog applications, probably," in *IEEE International Conference on Fog and Edge Computing*, 2017, pp. 105–114.
[6] C. Canali and R. Lancellotti, "Paffi: Performance analysis framework for fog infrastructures in realistic scenarios," in *Int. Conf. on Computing, Communications and Security*, 2019.
[7] F. Dai, H. Chen, Z. Qiang, Z. Liang, B. Huang, and L. Wang, "Automatic analysis of complex interactions in microservice systems," *Complexity*, 2020.
[8] S. Eismann, J. Kistowski, J. Grohmann, A. Bauer, N. Schmitt, and S. Kounev, "Teastore-a micro-service reference application," in *IEEE Int. Workshops on Foundations and Applications of Self* Systems*, 2019, pp. 263–264.
[9] Y. Gan *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2019.
[10] M. Gotin, F. Lösch, R. Heinrich, and R. Reussner, "Investigating performance metrics for scaling microservices in cloudiot-environments," in *ACM/SPEC Int. Conf. on Performance Engineering*, 2018, pp. 157–167.
[11] J. Grohmann, P. K. Nicholson, J. O. Iglesias, S. Kounev, and D. Lugones, "Monitorless: Predicting performance degradation in cloud applications with machine learning," in *Int. Middleware Conf.*, 2019, pp. 149–162.
[12] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments," *Software: Practice and Experience*, vol. 47, no. 9, pp. 1275–1296, 2017.
[13] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. Epema, and A. Iosup, "An experimental performance evaluation of autoscaling policies for complex workflows," in *ACM/SPEC on Int. Conf. on Performance Engineering*, 2017, pp. 75–86.
[14] "Jaeger," https://www.jaegertracing.io, last accessed June 2021.
[15] I. Lera, C. Guerrero, and C. Juiz, "Yafs: A simulator for iot scenarios in fog computing," *IEEE Access*, vol. 7, pp. 91 745–91 758, 2019.
[16] J. Mace, "End-to-End Tracing: Adoption and Use Cases," Brown University, Survey, 2017.
[17] N. Mazzocca, M. Rak, and U. Villano, "The transition from a pvm program simulator to a heterogeneous system simulator: The hesse project," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, 2000, pp. 266–273.
[18] D. Meisner, J. Wu, and T. F. Wenisch, "Bighouse: A simulation infrastructure for data center systems," in *IEEE Int. Symp. on Performance Analysis of Systems & Software*, 2012, pp. 35–45.
[19] "OpenTelemetry," https://opentelemetry.io, last accessed June 2021.
[20] "OpenTracing," https://opentracing.io, last accessed June 2021.
[21] A. Panda, M. Sagiv, and S. Shenker, "Verification in the age of microservices," in *Workshop on Hot Topics in Operating Systems*, 2017.
[22] "Prometheus," https://www.weave.works/features/prometheus-monitoring/, last accessed June 2021.
[23] J. v. Kistowski, N. Herbst, and S. Kounev, "Limbo: a tool for modeling variable load intensities," in *ACM/SPEC Int. Conf. on Performance engineering*, 2014, pp. 225–226.
[24] A. Van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *ACM/SPEC Int. Conf. on performance engineering*, 2012, pp. 247–248.
[25] Y. Zhang, Y. Gan, and C. Delimitrou, "μqsim: Enabling accurate and scalable simulation for interactive microservices," in *IEEE Int. Symp. on Performance Analysis of Systems and Software*, 2019, pp. 212–222.
[26] "Zipkin," https://zipkin.io, last accessed June 2021.