Tool to Evaluate Microservice Software Architectures

Extended Abstract

João Jese da Costa
Instituto Superior Técnico
Lisboa, Portugal
joaojcosta@tecnico.ulisboa.pt

ABSTRACT

The microservice architecture offers numerous advantages for large-scale distributed systems, such as differentiated scalability, resilience and autonomous optimization of each subsystem. These advantages depend on the quality of building microservices, which is a complex process and fundamentally presents architectural and operational challenges. The system-building based on microservices results in the combination of several factors (communication protocols, replication, coordination, frameworks, and others). The combination of these factors results in a variety of platform alternatives, whose decision on which to choose and how to evaluate the impact of this decision on a common basis is no trivial task.

This dissertation proposed a Benchmark tool that allows evaluating the performance of underlying platform variants of microservices-based systems. Today's microservice benchmarking tools focus on monolithic systems, for example, TPC-W (a recognized benchmark for online commerce database evaluation) and many other tools, are not easily applied to microservice systems due to their characteristics. Our tool allows you to evaluate and compare, in terms of throughput and latency, these different implementations (both monolithic and microservices-based) regardless of the specific implementation of the application interface. Our tool delivered good results in terms of correction and performance compared to TPC-W and itself when compared to the geo-replicated deployment of the Shopping service. The tests performed showed that our tool evaluates with good accuracy and good performance.

KEYWORDS

Tool; Benchmarking; Benchmark; Microservice Architecture; Performance Evaluation.

1 INTRODUCTION

Traditionally applications are designed based on a monolithic architecture, Figure 1. In this architectural model, applications have tightly coupled modules, deployed as a single package and executed as a single process (Figure 1.(a)) [7, 20]. All components are combined in a single program, e.g., server-side operations, business logic, database tier, and integration are all interconnected and interdependent [1].

This architecture has several limitations for large-scale distributed systems, such as lack of resiliency, limited scalability, module deployment dependencies, and both code quality and agility are reduced with application growth [15, 20, 22].

Due to strong coupling, Figure 1.(a), any update requires the creation of a new application package. The complexity of the code

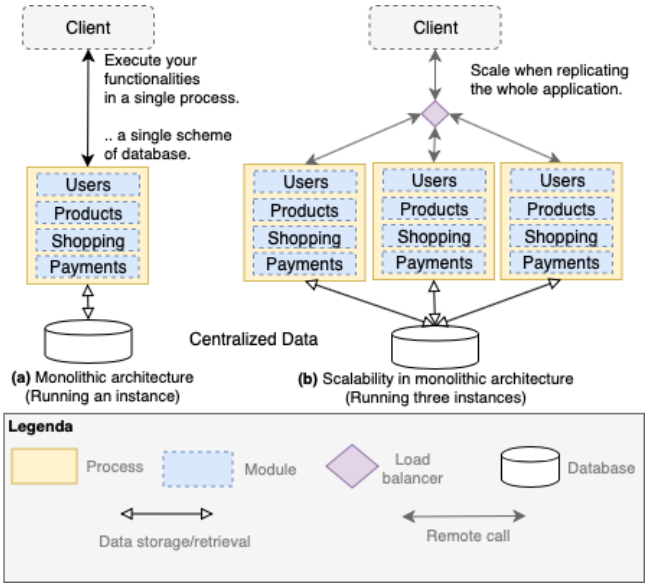

Figure 1: Monolithic Software Architectures - Their Characteristics.

gradually increases as a result of modifications, i.e., it is difficult to maintain a good modular structure. It is impossible to combine different technologies, e.g., to combine two database technologies to exploit the best of each. The application is not resilient, i.e. it is unable to resist the failure of one of the modules. The application only scales horizontally when running multiple instances of the application behind a load balancer, Figure 1.(b).

Due to these limitations, the microservice architecture, Figure 2, has now been adopted for the development of large-scale distributed applications [20].

Microservice enable a division of the application into autonomous, highly cohesive, loosely coupled units, Figure 2, so that these units can be independently developed, distributed, executed and maintained [11, 15].

These units are application components that are each executed in their own process communicating through Web Services or other similar mechanisms. While in monolithic applications libraries are defined as program-bound components and invoked using in-memory function calls [1], in microservices architecture the application components are microservices [11, 15].

Splitting the application into microservices enables differentiated scalability, data isolation, technological freedom, easy deployment, continuous delivery, organizational alignment, code reuse through

MEIC-A, Extended Abstract
2019.

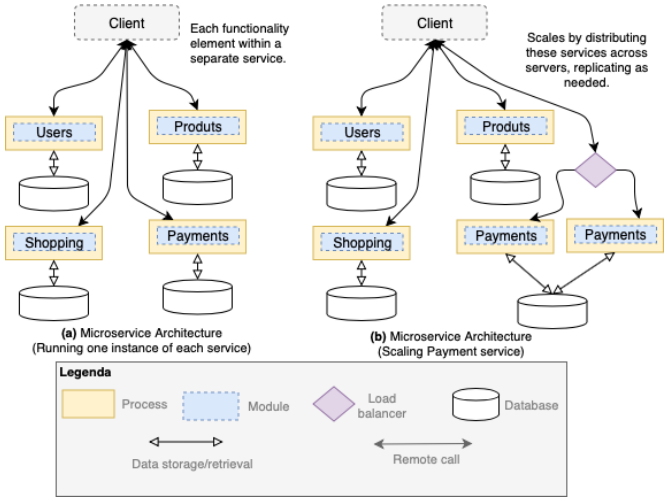

Figure 2: Microservice software architectures - their characteristics.

service calls and resilience as well as optimizing each subsystem for demand or business domain expansion [15, 22**?**].

The attractiveness of microservices for large-scale distributed applications has motivated companies such as Amazon [2], Microsoft [12], and Netflix [14] to adopt this architectural model and many others to migrate their systems to microservices architecture [18, 20].

The benefits offered by the microservice architecture depend on the build quality of the microservices. Building microservices is a complex process and presents fundamentally architectural and operational challenges resulting from the interaction between the various distributed components that compose the system [3, 15**?**].

The combination of several factors (protocols, framework and tools), [3, 9, 13], results in a large set of platform alternatives, whose decision on which to choose and how to evaluate the impact of this decision on a common basis is not a trivial task. A tool that allows comparing the various combinations and alternatives of solutions on a common basis is capable of evaluating the performance of these systems in order to minimize the complexity in evaluating the different design alternatives on a common basis.

One of the easiest and most straightforward ways to evaluate different systems in terms of performance is through benchmarking (a process by which one solution is compared in quantitative terms to a standard solution) [5, 10, 16].

Current microservice benchmarking tools focus on monolithic systems, for example TPC-W [6] (a recognized benchmark for e-Commerce database evaluation) and many other tools [17, 19], are not easily applied to microservice systems due to the characteristics of this architectural model.

The main objectives of this work is the following:

- Introducing a new benchmark tool that allows you to compare different implementations (both monolithic and microservice based) in terms of throughput and latency, and allows you to evaluate design alternatives regardless of the specific implementation of the application interface.
- We developed a version of a TPC-W microservice-based system under test.
- We performed a series of experiments to test the correctness and performance of our benchmark tool.

This document is organized as follows. Section 1 introduces the work. In section 2, a comparative analysis of the related works is done. In section 3, we present our solution by describing the architecture, execution workflow and some implementation details. In section 4, we evaluate our benchmark tool. Finally in section 5, the conclusion is done.

2 RELATED WORK

Benchmark SPEC Web 2009 [19], TPC-W [6] and RUBis [17] are transactional web service benchmark. Your workloads run in a managed environment that simulates the activities of a business-oriented transactional application server.

These benchmarks follow the three-tiered architectural pattern resulting from a significant number of monolithic cloud or on-premise applications prior to the emergence and popularization of microservices. Part of them are discontinued. Today they are adapted to study microservice-based application characteristics in various application domains, e.g.,[8].

These tools differ in terms of specific objectives, but in terms of overall purpose (application server performance evaluation), they are similar. The common server performance metric for these Benchmarks is throughput under response time constraints. The computation of this metric varies depending on the objectives and workload of each benchmark. They are all transactional web service application benchmarks. And in general, these benchmarks try to understand how an application server's performance - measured in response time and throughput - varies as more users access a website or web page.

All of these benchmarks are designed for monolithic systems, i.e. typically the focus is an application server integrated with a database server. None of the benchmarks presented is able to measure the performance of different implementations of microservices on a common basis simply.

3 SOLUTION

Introducing a new benchmark tool that performs measurements and benchmarking of both monolithic and microservice based systems. These operations are performed during three phases:

- **Pre-test**, which consists of the system configuration to guarantee the initial conditions of the experiments;
- **Test**, which consists of monitoring experiments and collecting data;
- **Post-test**, which consists in reducing data and comparing marks.

Our main concern during the design of our benchmark tool was to ensure two main qualities, namely (1) correctness and (2) performance of our benchmark tool during measurements and benchmarking. From these concerns, we have defined the following requirements:

- The tool shall function to ensure that the different phases do not overload each other, i.e., that there is no interruption

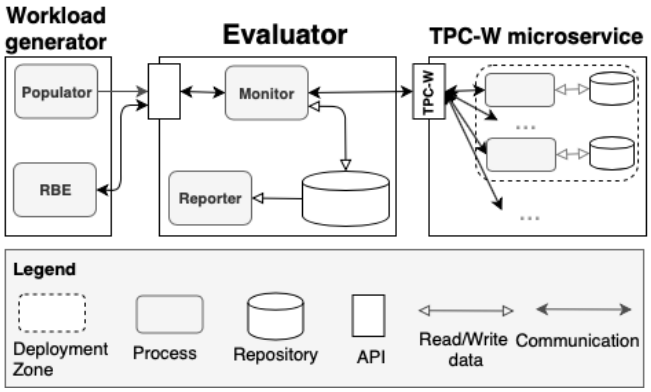

Figure 3: Architecture of our benchmark tool.

or overlap of tasks (database population, measurements or benchmarking);
- To minimize the impact of overload that can be introduced during measurements as well as validate measurements;
- The system under test must be simple, but have the characteristics relevant for evaluation, which produce results with good accuracy.

3.1 Architecture

On the left of Figure 3, is the Workload Generator. It is the part of the tool responsible for terminal emulation, users, and requests to be sent to Evaluator to exercise the SUT(System Under Test). This part of the tool implements the TPC-W specification to drive the workload at appropriate intervals used to exercise the SUT. It consists of two main components, namely (1) Populator which is a software component that populates databases with a TPC-W workload and (2) RBE(Remote Business Emulator) which is a software component that drives the TPC-W workload.

- Populator is responsible for ensuring the initial conditions for experiment.
- is responsible for terminal emulation, users, and workload.

On the center of Figure 3, is the Evaluator. It is the part of the tool responsible for measuring, recording marks and reporting results. Evaluator uses the workload generated by Workload Generator to exercise the SUT and collect data while monitoring the experiment. Evaluator has two main components and a third datastore component:

- Monitor: is responsible for the measurements and monitoring of the experiment.
- Reporter: Responsible for reducing data, comparing marks and reporting results.
- Repository: It is responsible for persistently storing the received metrics, experiment start-up and intermediates as well as all relevant information related to the state of the Evaluator.

On the right of Figure 3, is the SUT. It is the microservice version of the TPC-W SUT. The SUT is e-commerce retailing. This system deals with the business logic involved in order processing and

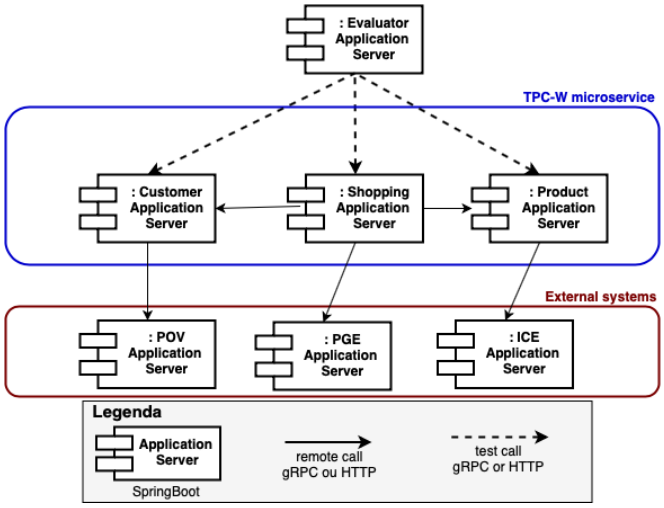

Figure 4: Architecture of our SUT.

retrieving items from the product catalog. Its business model is focused on the B2B(Business-To-Business) scenario.

The SUT has the following B2B interactions:

- New Customer
- Change Customer Payment Method
- Create Order
- Shipping
- Stock Management Process
- Order Status
- New Products
- Product Detail
- Change Item

The division of the monolithic into microservice, Figure 4, was based on the DDD(Domain-Driven Design) approach [4, 21], which consists of identifying entities, services, aggregations and modules in a systematic way. The SUT consists of three modules, namely Customer, Product and Shopping. The POV(Purchase Order Verification) module, the PGE(Payment Gateway Emulator) module and the ICE(Inventory Control Emulator) module are external systems to the SUT. All modules are associated with a database with their respective data model. The Evaluator component in Figure 4, serves to illustrate the interaction between the SUT and the Evaluator.

3.2 Execution Workflow

Workflow follows several steps, some of which occur before measurements, during and after measurements. In summary the typical workflow has the following life cycle:

(1) Scaling and database population, this step occurs before measurements.
(2) Activation of measurements, this step initializes the measurements.
(3) Performing measurements, this step performs measurements and registration of marks.
(4) Ending Measurements, this step ends the measurements.
(5) Report presentation, reports the results obtained.

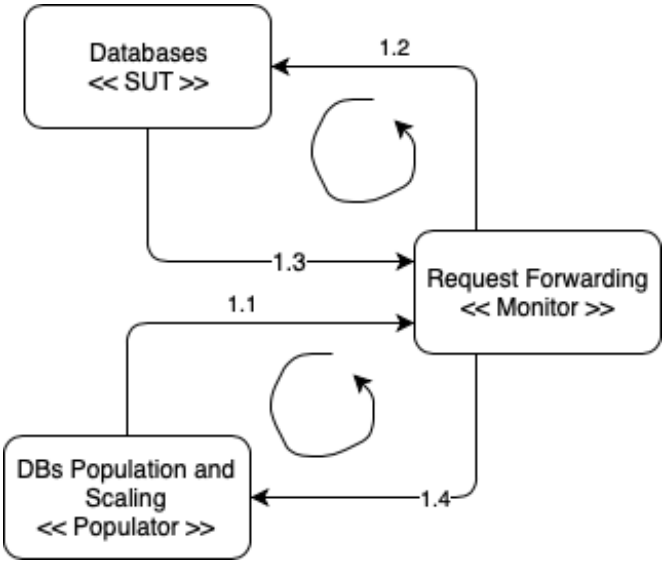

Figure 5: Database scaling and population.

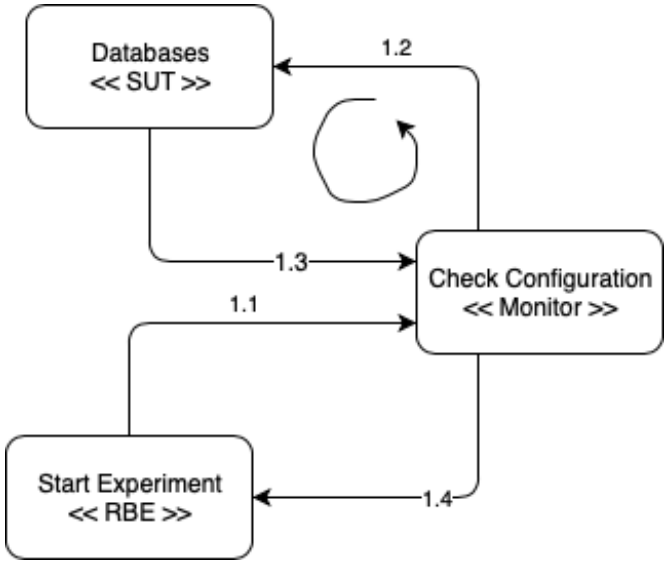

Figure 6: Activation of measurements.

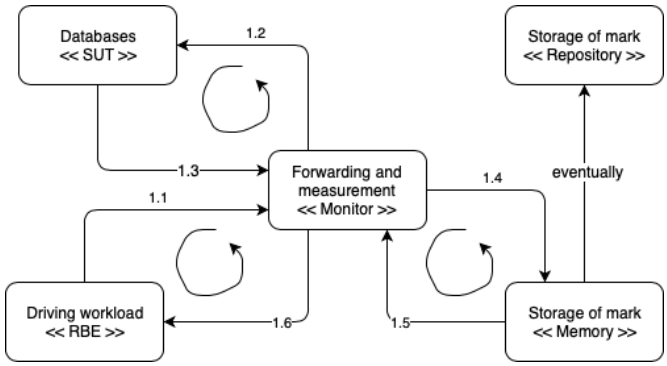

Figure 7: Performing measurements.

1. Scaling and database population To get started, is necessary to ensure the initial conditions for the experiment. To do this, Figure 5, Workload Generator runs Populator. Popular loads the settings and initializes the scalability parameters of the databases. It then connects with Evaluator to initiate transactions related to the initial population. When you initialize Evaluator, it recreates the tables in each SUT database. For this reason, Populator does not include table deletion and table creation operations. For each transaction made by Populator, Evaluator performs interactions for each Endpoint in the georeplication matrix associated with its B2B interaction. Database population is performed for all SUT modules sequentially. Finally, after completing the population process, Populator requests database state statistics to verify that all operations were performed successfully. The main statistical information is the cardinality of the databases, which is the main argument of the Populator checker.

2. Activation of measurements Prior to measurements, it is necessary to verify the initial conditions for the experiment and activate the measurements. To do this, Figura 6, Workload Generator runs RBE which initially loads the settings (includes the type of test, which may be normal or georeplication and, if georeplicated; the type of interaction, which may be random or defined), sends a special start experiment request to Evaluator. Evaluator via Monitor loads the configurations (response time constraint matrix and other configurations), checks the initial state of the database and, if necessary, checks the georeplication matrix configuration if they satisfy the requested test type. If the check is satisfied, the Monitor activates the measurements and returns the status to RBE, the status includes the list of available zones and regions, database cardinalities, and whether or not to start the experiment.

3. Performing measurements As illustrated in Figure 7, with the initial conditions met and approved for the start of the experiment, RBE initiates the clients and then performs interactions with Evaluator with a probabilistic reflection time. Evaluator receives the order through Monitor and counts it. The Monitor performs the request mapping, measures the operation time, checks whether the measured time for the operation satisfies the constraint matrix, satisfies stores the trademark, counts response, and returns the response to the respective RBE.

4. Ending Measurements To end the measurements, Figure 8, RBE is sent a special request to Evaluator. Evaluator through Monitor checks that all orders have been completed, all completed successfully, calculates SIPS(Service Interactions Per Second) gloabal, stores persistently in Repository, terminates measurements (next orders will not be executed) and returns a success message to RBE. If there are incomplete orders of a given operation type, check if 90% of this order type was successfully completed, calculate SIPS gloabal, persistently store in Repository, close measurements (next orders not will be executed) and returns a success message to RBE. If the completed order number is below 90%, Monitor logs a global Repository error for the current experiment and returns an error to RBE. The error returned to RBE means that the experiment must be repeated.

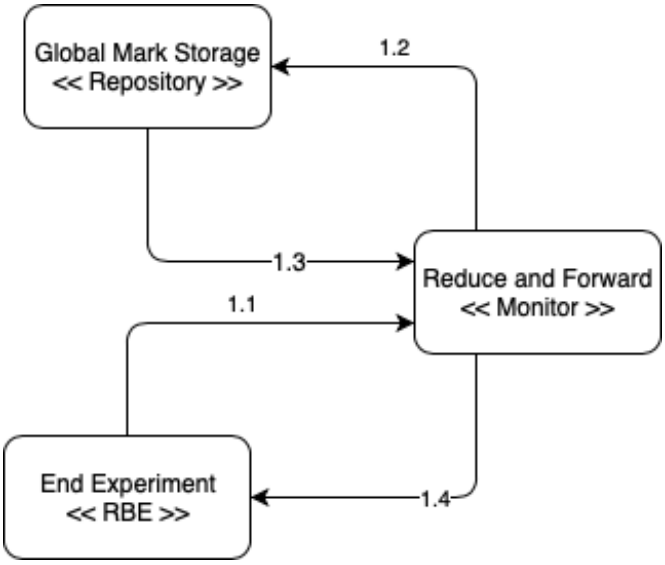

Figure 8: Ending Measurements.

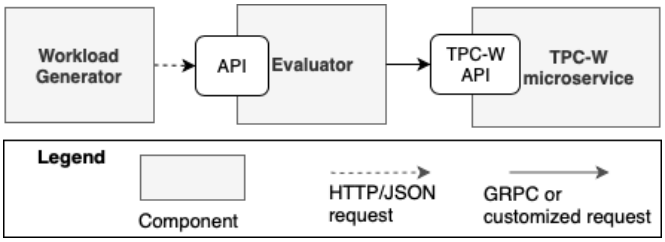

Figure 9: Communication between tool components.

5. Report presentation For reporting purposes, Reporter retrieves the experiments in Repository, presents a summary of the retrieved experiments, and provides the toolset. The main tools are: (1) list marks in terms of SIPS and (2) compare marks.

3.3 Implementation

Our tool was implemented using Java, MySQL, Docker and a set of frameworks such as gRPC, Spring Boot and Spring Cloud, due to the cost-effective performance versus simplicity of implementation that these tools provide together.

Among its many attractions, Java offers good documentation and support, offers a robust platform for development, and has a very high number of third party cloud-related tools for microservices. MySQL is a well-performing relational database used on critical business systems and packaged applications. Docker allows you to package code and all its dependencies so that the application runs quickly and reliably from one computing environment to another through a container, as well as orchestrating services at runtime. And finally, the various frameworks used allowed for unmatched agility in building the modules.

Communication

As illustrated in Figure 9 , messages exchanged between Workload Generator and Evaluator have been standardized using the JSON format. Separate Workload Generator from Evaluator - Workload Generator works as remote terminal TPC-W and combined with Evaluator forms our tool's remote terminal emulator, putting it in a separate process, uses HTTP / JSON to perform communication required. This allows (1) the database starter (Populator) and business emulator interactions to be customized to the intended experience and (2) to be developed in any available programming language.

For communication between Evaluator and the system under evaluation, we built RPC and RESTful APIs using the gRPC and Spring REST frameworks. This allows (1) the user to have alternatives of implementation, in terms of model and / or communication technology, of the system under evaluation and to (2) easily extend it to others, for example RMI.

Configuration

The main configuration file in Evaluator plays an important role for loading and testing the system under evaluation. This file provides information about: (a) the client type and (b) operations and endpoint set associated with it . The information in the file has been standardized using the YAML format, which is a data serialization language and has a human readable structured data format.

Workload Generator also has a configuration file, this file plays an important role for scalability, database population and the interactions of the experiment being tested. We put in this file information about: (a) georeplication parameter, i.e., mode of interaction for replicas and geo-distribution scope, (b) item number, (c) number of business emulators and (d) distribution percentage for the interaction mix.

Main Components

Evaluator performs interactions with Workload Generator and SUT, Figure 9, with or without measurement capability. An experiment is delimited by two special requests, one indicating the start of the emulation phase, i.e., the start of measurements and another indicating the start of the post-emulation phase, i.e the end of measurements and the start of the measurement. report generation. Thus, we control the experiments and allow Evaluator to function in two modes, (1) as a mediator in the pre-emulation phase, that is, to allow the databases to be popular, since only he knows the communication interface of the SUT and (2) as a remote terminal with measuring capability during the emulation phase.

In the pre-emulation phase, Evaluator receives requests produced by Populator to build the test databases, including their initial population and all indexes present during the test. Evaluator acts as a mapper translating requests into SQL-DDL operations on MySQL databases through Spring Data Repositories and later on interactions within each microservice to populate their test databases.

In the Emulation phase, the process begins with the measurement activation step. In this step, Evaluator receives test settings, which is used to validate interactions. The execution of the measurements is implemented as illustrated in Figure 10. Upon receipt of the request, Mapper retrieves a nonempty set of Endpoints (the 2 Algorithm presents the strategy used for mapping), which is used by Measurer to route the request and take measurements. After the measurement, Validator checks whether the time constraint per operation policy has been met. If satisfied temporarily stores the

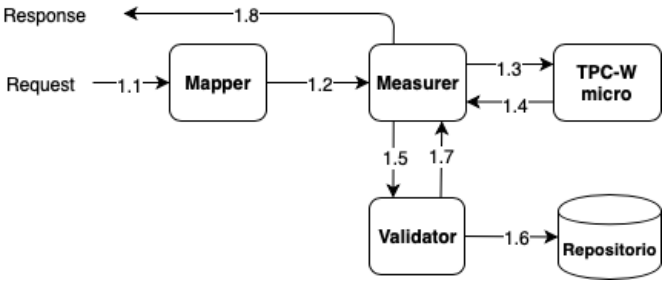

Figure 10: Performing Measurements - Implementation Details.

brand and persists eventually. If the constraint was not met, then repeat the measurement.

Mapper identifies Endpoints for both the georeplication scenario and the single data center scenario(Algoritmo 1 e Algoritmo 2).

Data: *request*, *test*
Result: *endpoint*
endpoint ⟵ *null*;
index ⟵ −1;
if *test.getMode*() == *Mode.GEO* **then**
  if *test.getType*() == *Type.RAND* **then**
    index ⟵ *getRandomPosition*();
  else
    index ⟵ *getProbabilisticPosition*();
  end
  endpoint ⟵ *matrix*(*index*);
else
  return *endpoint*
end
;

Algorithm 1: GetEndpointGeo: Estratégia para obter próximo endpoint - teste de georeplicação.

Data: *request*, *test*
Result: *list*
host ⟵ *GetEndpointGeo*(*request*, *test*);
list ⟵ {}
if *host* == *null* **then**
  list ⟵ *getAllEndPoint*();
else
  list.add(*host*);
end
return *list*;

Algorithm 2: Mapper: Estratégia para mapeamento de interação.

In the post-emulation phase, Reporter accesses the database and makes comparisons against a defined reference or between retrieved tags if the retrieved data is the result of a successfully completed experiment.

4 EVALUATION

Our evaluations and discussions focused on the main concerns, namely the correctness and performance of our tool.

4.1 Settings

We evaluated our Benchmark tool in two deployment scenarios, namely (1) in a single zone and (2) geographically distributed. We use Docker, Docker Compose, and Docker Machine to deploy each service on an Amazon Web Services micro EC2. The Shopping service was the only one replicated and it was only in two regions where we added a master-slave configuration. We use the Amazon CloudWatch monitoring and observation service tool to check the state of test configurations before testing begins.

Testing for correctness was performed on a single availability zone, while testing for performance used two regions.

Two parameters were configured for database scaling, namely the number of Emulated Business(), *NUM_BS*, and the cardinality of the table ITEM, *NUM_ITEMS*. Database scalability is defined by the size of the supported EB population. The cardinality of the table ITEM we keep fixed, *NUM_ITEMS* = 100000. The cardinality of the STOCK and AUTHOR tables is a function of the ITEM table. The initial cardinality of the other tables is a function of the configured EB number. Thus, only the number of EB was varied, i.e the number of active users.

Each EB contains an interaction mix matrix, Table 1, to execute within a BSL(Business Session Length). A WIRT(Web Service Interaction Response Time) constraint, Table 2, is applied, where 90% of successful interactions must have a WIRT less than the constraint. In this way the interaction mix as well as the WIRT constraint is configured.

Table 1: Matriz de mistura de interações.

Id	Interação	Mix
A	New Customer	0, 01
B	Change Payment Method	0, 05
C	Create Order	0, 50
D	Create Status	0, 20
E	New Products List	0, 10
F	Product Detail	0, 12
G	Change Item	0, 02

4.2 Correctness

We have checked the correctness of our tool by subjecting TPC-W and our tool to the same similar workload and test environment to observe result compliance in terms of request / response matching versus database consistency. and in terms of performance versus number of EB.

Table 2: Matriz de restrição de WIRT

	A	B	C	D	E	F	G
Restrição	3	3	4	4	1	1	1

In terms of request / response matching versus database consistency, we performed an experiment with 10 EB and a minimum reflection time of 7 seconds and for the same WIRT interaction and constraint mix matrix, we observed that equality in the request / response number and consistency of operations. To check the consistency of the operations, we built SQL(Structured Query Language) queries that, in addition to returning the record amount, relate the various reference keys. And we find that the operations are carried out consistently.

We also checked in terms of performance versus EB number. The check consisted of checking that for each number of active users the SIPS satisfies the inequality 1 for the minimum reflection time. Where X represents the number of active users (EB), Y the average think time and Z the SIPS debit. This inequality sets the throughput limits for a given number of active users operating at a given average reflection time. Guarantees acceptable result given the definition of these parameters.

$$\frac{X}{Y} < Z < \frac{X}{Y \times 2} \tag{1}$$

As can be seen from Figure 11, and as expected meets the inequality, i.e., respects the SIPS limits for the number of active users and the average think time.

In Figure **??**, we can see that our tool for the same parameters, although presenting an inferred rate, satisfies the inequalities, i.e., the reported SIPS rates meet the established limits.

Both systems satisfy the correctness conditions defined as an evaluation criterion, so we can conclude that our tool has a correct SIPS according to the main criteria defined by the TPC to verify the TPC-W implementation correction.

4.3 Performance

To evaluate the performance of our Benchmark tool, we submit to our tool a test workload similar to that submitted to the TPC-W. In this section, we will present and discuss the results of these tests. We also present and discuss the result of the georeplication test workload.

As can be seen in Figure **??** and Figure **??**, there is a very strong approximation between the results reported by both tools. For a smaller number of active users the SIPS rate reported by the higher than reported by our tool, but the situation reverses by significantly increasing the number of active users.

The above situation, Figure **??** and Figure **??**, led us to observe the average response of each tool, Figure 15. When we compare, we find that the average response time of our tool for a small number of active users is higher compared to the average response time of TPC-W. And the situation is reversed as the number of active users increases.

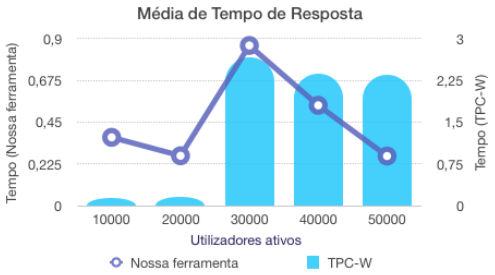

Figure 15: Average Response Time: TPC-W and Our Tool

Thus, we have observed that (a) for a small number of active users, a monolithic application is able to fulfill requests faster than the microservice version. Communication between different servers impacts the performance of the microservice version. (b) While for a larger number the situation is reversed due to single point overhead, ie the monolithic application has a single access point and all interactions are directed at it, while the microservice version has multiple servers and the load is distributed among them.

SIPS throughtput reported by our tool, such as that reported by TPC-W, indicated that our benchmark tool performs well.

5 CONCLUSION

Microservices enable differentiated scalability, data isolation, technological freedom, easy deployment, code reuse, resilience, as well as optimizing each subsystem to meet business demand or business agile expansion. These advantages depend on the build quality of microservices, which is a complex process and fundamentally presents architectural and operational challenges resulting from the interactions between the various distributed components that make up the system. Current microservice benchmarking tools focus on monolithic systems and are not easily applied to microservice systems.

This work developed a benchmark tool that evaluates the performance of underlying platform variants of microservices based systems on a common basis, developed a microservice version of the SUT TPC-W and performed a series of experiments to test the correctness and the performance of our tool.

Our tool performed well in terms of correctness and performance compared to TPC-W and itself when compared to georeplicated deployment of the Shopping service.

REFERENCES

[1] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. 2004. *Web Services - Concepts, Architectures and Applications (Data-Centric Systems and Applications)* (1st ed.). Berlin, Heidelberg : Springer Berlin Heidelberg. 368 pages. https://doi.org/10.1007/978-3-662-10876-5

[2] AWS. 2017. Microservices. (2017). https://aws.amazon.com/microservices/?nc1=h{_}ls

[3] Shahir Daya, Nguyen Van-Duy, Kameswara Eati, Carlos Ferreira, Dejan Glozic, Vasfi Gucer, Manav Gupta, Sunil Joshi, and Valerie Lampkin Marcelo Martins. 2015. Microservices from Theory to Practice Creating Applications in IBM Bluemix Using the Microservices Approach. (2015).

[4] Eric Evans. 2003. *Domain Driven Design: Tackling Complexity in the Heart of Software.*

[5] Paul Fortier and Howard Michel. 2013. *Computer Systems Performance Evaluation and Prediction.*

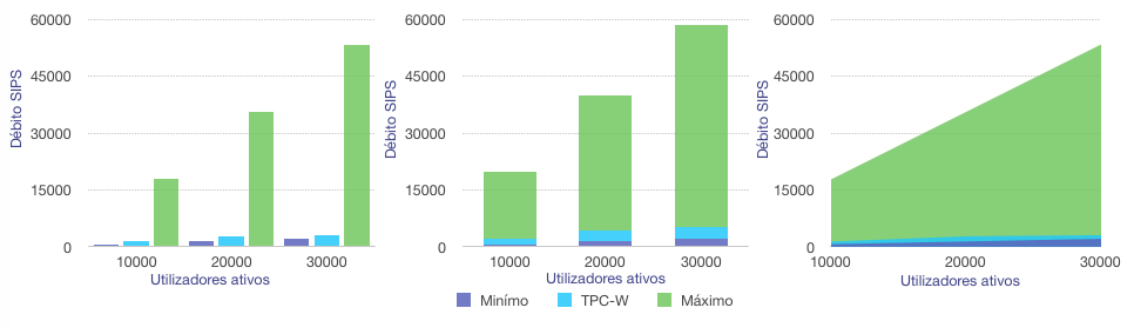

Figure 11: SIPS Limit Satisfaction Check for TPC-W.

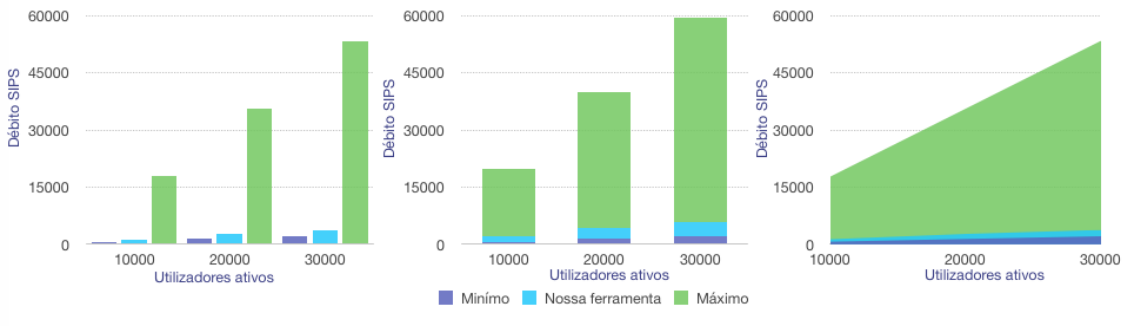

Figure 12: SIPS limits compliance check for our tool.

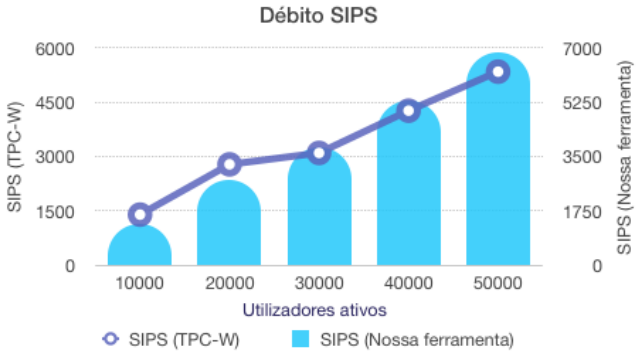

Figure 13: SIPS throughput comparison: TPC-W vs Our tool.

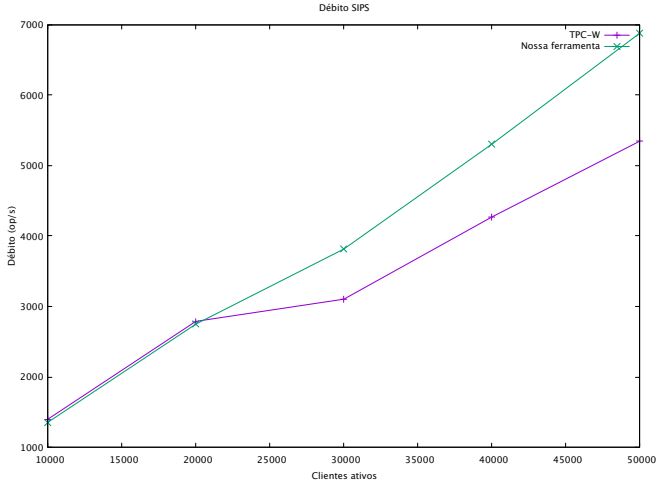

Figure 14: SIPS throughput comparison: TPC-W vs Our tool.

[6] San Francisco. 2003. *TPC BENCHMARK W (Web Commerce).* Technical Report. http://www.tpc.org/tpc{_}documents{_}current{_}versions/pdf/tpcw{_}v2.0.0.pdf

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional. 416 pages.

[8] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, Christina Delimitrou, Y Zhang, D Cheng, A Shetty, P Rathi, N Katarki, A Bruno, J Hu, B Ritchken, B Jackson, K Hu, M Pancholi, Y He, B Clancy, C Colen, F Wen, C Leung, S Wang, L Zaruvinsky, M Espinosa, R Lin, Z Liu, J Padilla, and C Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. *Asplos* (2019). https://doi.org/10.1145/3297858.3304013

[9] Thomas Hunter II. 2017. *Advanced Microservices, A Hands-on Approach to Microservice Infrastructure and Tooling.*

[10] Raj Jain. 1991. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling* (1st ed.).

[11] James Lewis and Martin Fowler. 2014. Microservices - a definition of this new architectural term. (2014), 14. https://martinfowler.com/articles/microservices.html

[12] Microsoft. 2016. .NET Microservices Architecture Guidance. (2016). https://dotnet.microsoft.com/learn/aspnet/microservices-architecture
[13] Vikram Murugesan. 2017. *Microservice Deployment Cookbook*.
[14] Netflix. 2015. Netflix Open Source Software Center. (2015). https://netflix.github.io/
[15] Sam Newman. 2015. *Building Microservices*. O'Reilly Media. 280 pages. https://doi.org/10.1109/MS.2016.64
[16] Mohammad S Obaidat and Noureddine A Boudriga. 2010. *FUNDAMENTALS OF PERFORMANCE EVALUATION OF COMPUTER AND TELECOMMUNICATION SYSTEMS*.
[17] OW2. 2003. RUBiS: Rice University Bidding System. (2003). https://rubis.ow2.org/
[18] Boris Scholl, Trent Swanson, and Daniel Fernandez. 2016. *Microservices with Docker on Microsoft Azure*.
[19] SPEC. 2011. SPECweb® 2009. (2011). https://www.spec.org/web2009/
[20] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. 2017. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing* 4, 5 (2017), 22–32. https://doi.org/10.1109/MCC.2017.4250931
[21] Vaughn Vernon. 2013. *Implementing Domain-Driven Design*.
[22] Eberhard Wolff. 2017. *Microservice: Flexible Software Architecture*.