Extracting Reusable Service Demands for TeaStore

Elijah Seyfarth
s210942@th-ab.de
Aschaffenburg UAS, Germany

Sebastian Frank
sebastian.frank@uni-hamburg.de
University of Hamburg, Germany

Jóakim von Kistowski
joakim.vonkistowski@th-ab.de
Aschaffenburg UAS, Germany

Abstract

Software developers and system operators can use simulators to predict the performance and resilience of a software application. However, the accuracy of the prediction heavily depends on the calibration of the performance model, i.e., the service demands of the system to be modeled. As calibration is a tedious process, this paper shares the extracted CPU service demands of the widely used TeaStore reference microservice application, incl. the corresponding approach and challenges. We extracted the service demands by measuring utilization during experiments and deriving service demands for each of the application's operations using the service demand law. We exemplify the use of the extracted service demands for simulating TeaStore with the MiSim resilience simulator.

1 Introduction

Several performance and resilience simulators, such as MiSim [11] and the Palladio Component Model [4], enable quality predictions in what-if scenarios. These tools support architectural decision-making without requiring full implementation. For instance, MiSim helps to evaluate and configure resilience mechanisms, such as retries, timeouts, and circuit breakers.

Although simulations are less costly than real experiments, they trade off accuracy and require calibrated performance models. Calibration typically depends on *service demand* data (sometimes referred to as *service demands* [5]), often estimated using specialized techniques [6]. However, obtaining accurate service demands is challenging, complicating model setup — both in industrial contexts and research, e.g., with reference architectures like TeaStore [8]. In the absence of complete public datasets, researchers must perform measurements and calibrations themselves.

This work contributes (i) a description of the setup to calibrate a MiSim performance model of TeaStore, (ii) insights gained during the process, and (iii) the associated service demands and artifacts. These results support both practitioners aiming to calibrate their own systems and researchers using TeaStore.

In our approach, we used the service demand law [1] to extract the CPU service demands from the TeaStore. Beforehand, we measured the CPU utilization of the TeaStore under load. The extracted service demands, including the scripts necessary for reproducing our experiments, can be found on GitHub[1] and Zenodo[2].

To evaluate the extracted service demands and demonstrate their use, we applied them to an architectural description of TeaStore and executed MiSim.

2 Related Work

The literature related to this work can be grouped into two main categories:

Service Demand Estimation Multiple general approaches for service demand estimation have been proposed [3, 5, 7, 10]. An evaluation of several approaches [6] shows the service demand law [1] to deliver the most precise results. Thus, we decided to use this approach in our calibration process.

Characterizing TeaStore Performance Other works performed similar experiments on the TeaStore to characterize its performance [9, 12].

However, previous works on TeaStore either did not extract any service demands or did not publish them. In contrast to these existing works, we focus on the extraction process and publish service demands for reuse in any performance or resilience modeling context.

3 Process

To obtain the service demands needed to calibrate a performance model, we measured the CPU utilization of the TeaStore under load. Afterwards, we were able to calculate the service demands by applying the service demand law [1] to the measured utilization and system throughput.

[1] https://github.com/TH-Aschaffenburg-Software-Design/teastore-service-demands

[2] https://doi.org/10.5281/zenodo.16959789

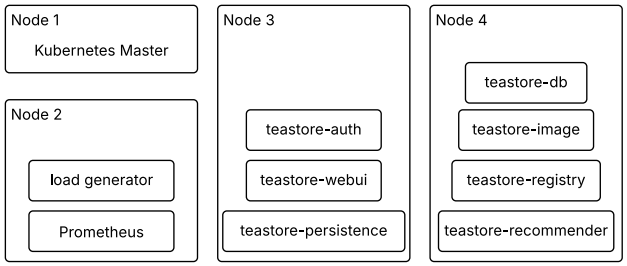

Figure 1: Distribution of components on the virtual machines in the Kubernetes cluster

3.1 Setup

We set up the TeaStore [8], a microservice reference application for benchmarking. We deployed TeaStore in a Kubernetes cluster of four virtual machines, each having two virtual 2.60 GHz CPUs and 8 GB of memory as shown in Figure 1. To rule out contention over the CPU resource, we limited the CPU of each of TeaStore's pods to 0.6 cores, except for the registry with 0.1 cores. Since TeaStore is a Java application that uses a Just-In-Time compiler, we warmed it up for at least 4 hours using TeaStore's *Buy Profile* to guarantee realistic behavior. We monitored the TeaStore using kube-state-metrics and stored the metrics in a Prometheus database. To load the system, we used the k6 load generator, as it offers advanced scripting and provides an open workload model [2], which is crucial for controlling the utilization during the measurement.

3.2 Challenges

When designing the experiment for service demand extraction, we faced multiple challenges.

Isolating the operations We were using kube-state-metrics to monitor the CPU utilization of the pods. However, to parametrize a performance model, service demands on a per-operation level are required. Sticking to our initial approach of using the service demand law, we had to extract the utilization of individual REST-operations. To obtain the desired metrics, we ran an individual experiment for each operation. As TeaStore uses REST APIs for inter-service communication, all operations are exposed as endpoints. Thus, we set up the experiment to generate load onto a single endpoint at a time over the experiment period. Afterwards, we retrieved the CPU utilization of the pod that exposed the endpoint. This way, we isolated the utilization of the endpoint under test from the utilization of its dependencies. Note that TeaStore does not have dependencies within the same service, which enables this method.

Parameterizing the endpoints Due to the need for an isolated experiment run per operation, we had to directly call REST APIs of services designed for internal communication and not only the top-level REST API of the *webui* service. These internal APIs use special request bodies like session blobs or entities with their IDs. We prepared these objects in advance, e.g., we obtained a fresh session blob by sending a login request with specific user credentials to the *auth* service. That way, we prepared 100 session blobs for each of TeaStore's users. Apart from user information, a session blob holds the user's shopping cart items. Because of that, we filled the shopping cart of each prepared session blob with 2 items. During the experiment, we varied the concrete arguments of each request, if possible, to get realistic results. Random numbers either served directly as an argument or as an index for selecting an object out of a list. To foster the experiment's repeatability and as a guarantee to use the whole range of possible arguments, we chose sequencing instead of random generation as a number-generation method.

Handling the database Despite TeaStore holding the user's session data inside the session blobs, the event of placing an order is persisted in the database. As a result, the database fills up with each experiment run. We noticed that this has a significant impact on the performance of some retrieval operations. Thus, we had to reset the database using TeaStore's database generation endpoint. As the resulting empty database state is not realistic, we decided to only reset the database once before the experiment, and then run the experiment for each operation subsequently. In addition to that, we placed all operations affecting the database in the beginning to create the database state for the remaining operations.

3.3 Experiment run

We configured the k6 load generator with an open workload and a constant arrival rate. For each operation, we determined an individual arrival rate based on preceding measurements, aiming for a utilization of 50% to prevent unwanted effects from under- and overloading. However, we observed that only a utilization of 25% was reached, despite the CPU of the load generator node was running at full capacity. A possible explanation for this behavior is that TeaStore microservices exhibit bottlenecks typical for microservice applications, such as I/O operations or bottleneck dependencies to other services. As depicted in Figure 2, the load was executed over a period of 125 s per operation. This interval includes a sufficient warm-up period of 20 s, as well as an ending cut of 5 s, leaving a clean measurement period of 100 s. All operation experiments were run subsequently with a load pause of 5 s in between each run.

After the experiment run, we used the service demand law [1] to calculate the service demands out of the measured CPU utilization and the arrival rate, assuming a steady state equilibrium. We repeated the experiment 18 times and provide the mean service demands to reduce the impact of random errors.

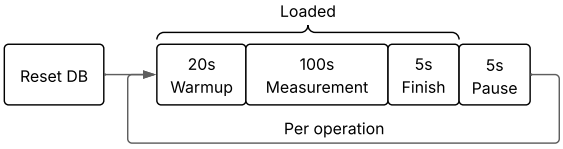

Figure 2: Experiment flow

3.4 Garbage Collection

We observed significant response time peaks in regular intervals. As we consider this unrealistic for actual production usage, we tried to tune the garbage collector to lower the peaks. We achieved this by decreasing the `MaxGCPauseMillis` parameter to 100 for all of TeaStore's services. For completeness, we extracted service demands for three different configurations *default*, *tuned*, and *Z Garbage Collector*. We proved a statistically significant difference of service demands in 25 of the 32 operations at a 95% confidence level using a one-factor ANOVA with a Bonferroni correction.

4 Applying the service demands

The service demands can be used to parametrize performance and resilience models. To demonstrate this, we calibrated an architectural description of the TeaStore with the obtained service demands and performed a simulation in MiSim [11]. We defined the experiment as an open model workload with a constant arrival rate of 120 rps, distributed on the eight *webui* endpoints, over an interval of 200 s. For comparison, we set up a final experiment to measure the response time of TeaStore with the same parameters. Over 10 replications of this experiment, we observed a mean response time ranging from 0.0067 s to 0.0074 s. In contrast, the simulation produced a deterministic mean response time of 0.0201 s. This discrepancy stems from the fact that MiSim can only be parametrized with one set of service demands. This limits the performance model to focus on one single resource, while TeaStore is influenced by several resources in reality.

5 Conclusion

The provided service demands enable researchers to run simulations of TeaStore without the application setup. This allows for a quick evaluation of a performance model, skipping the tedious process of calibration. Our scripts and learnings can be used to easily reproduce the experiments on different hardware, which may be necessary for direct comparison of TeaStore and its simulation.

In future work, we are planning on repeating the experiments using more powerful hardware to further evaluate the results and extract service demands of additional resources. We also aim to build on the setup to thoroughly evaluate and improve the prediction capabilities of MiSim for resilience scenarios.

References

[1] D. A. Menascé, V. A. F. Almeida, and L. Dowdy. *Performance by Design: Computer Capacity Planning by Example*. Prentice Hall Professional, 2004.

[2] B. Schroeder, A. Wierman, and M. Harchol-Balter. "Open Versus Closed: A Cautionary Tale". In: *NSDI '06*. San Jose, CA, May 2006.

[3] X. Wu and M. Woodside. "A Calibration Framework for Capturing and Calibrating Software Performance Models". In: *Computer Performance Engineering*. Berlin, Heidelberg: Springer, 2008, pp. 32–47.

[4] S. Becker, H. Koziolek, and R. Reussner. "The Palladio Component Model for Model-Driven Performance Prediction". In: *JSS*. Special Issue: Software Performance - Modeling and Analysis 82.1 (Jan. 2009), pp. 3–22.

[5] S. Spinner et al. "LibReDE: A Library for Resource Demand Estimation". In: *ICPE '14*. New York, NY, USA: ACM, Mar. 2014, pp. 227–228.

[6] S. Spinner et al. "Evaluating Approaches to Resource Demand Estimation". In: *Performance Evaluation* 92 (Oct. 2015), pp. 51–71.

[7] S. Eismann et al. "Modeling of Parametric Dependencies for Performance Prediction of Component-Based Software Systems at Run-Time". In: *ICSA '18*. IEEE, Apr. 2018, pp. 135–13509.

[8] J. Von Kistowski et al. "TeaStore: A microservice reference application for benchmarking, modeling and resource management research". In: *MASCOTS '18'*. IEEE. 2018, pp. 223–236.

[9] S. Eismann et al. "Microservices: A Performance Tester's Dream or Nightmare?" In: *ICPE '20*. Edmonton AB Canada: ACM, Apr. 2020, pp. 138–149.

[10] M. Mazkatli et al. "Incremental Calibration of Architectural Performance Models with Parametric Dependencies". In: *ICSA '20*. IEEE. Mar. 2020, pp. 23–34.

[11] S. Frank et al. "MiSim: A simulator for resilience assessment of microservice-based architectures". In: *QRS '22*. IEEE. 2022, pp. 1014–1025.

[12] J. S. Grohmann. "Model Learning for Performance Prediction of Cloud-native Microservice Applications". PhD thesis. JMU Würzburg, Mar. 2022.