Microservice-tailored Generation of Session-based Workload Models for Representative Load Testing

Henning Schulz, Tobias Angerstein
Novatec Consulting GmbH, Germany

Dušan Okanović, André van Hoorn
University of Stuttgart, Germany

***Abstract*—Load tests are commonly used to assess the performance of an application system. A representative load test uses workload characteristics according to the user behavior in production. Session-based systems have special workload characteristics as the system is used as sequences of inter-related requests. Approaches exist to automatically extract session-based workload models from production request logs. However, they focus on system-level testing, which is in stark contrast with modern development practices, where one development team is in charge of developing, testing, and deploying a single microservice. Hence, representative session-based workload models for testing single microservices and their integration are desirable.**

To deal with these issues, we propose a concept for tailoring a representative load test workload to target only certain services, instead of targeting the whole system. Our goal is to transform the workload for one or more specified service(s) from the system-level workload collected in production. Using this approach, only a subset of the application's microservices is deployed for a load test, specifically the targeted services and the services they depend on. We propose two algorithms. The log-based algorithm deals with extracting the workload for a specific service from collected production traces. The model-based algorithm performs the workload tailoring on the level of the workload model.

In an experiment series with a representative microservice application, we compare both algorithms with system-level and request-based workoad models. The results show that when load testing a set of services, the tailored workload models outperform untailored workload models in terms of test duration and the capacity of the test infrastructure, and outperform request-based workload models in terms of representativeness.

I. Introduction

The user satisfaction and business value of application systems is highly dependent on runtime quality properties such as performance. Performance evaluation is an integral part of the software development cycle. A common approach for performance evaluation is load testing [24]: a load driver generates synthetic requests to the system under test according to a previously defined workload model. Representativeness is a common requirement for load tests, i.e., the characteristics of the generated workload — e.g., arrival rates and parameters of requests — correspond to the characteristics of the production workload [11]. Different approaches for extracting workload models from production request logs have been proposed [30], [39], [41]. Fig. 1 depicts a typical workflow for the extraction of session-based workload models and load tests: ① Request or trace information is collected in respective logs; ② Session logs are extracted from the request/trace logs by grouping the requests by the session ID; ③ The session log records are clustered to obtain a workload model; ④ The workload model is parameterized [37] and transformed into the resulting load test.

Modern application systems comply to the microservices architectural style [33], and are developed and maintained according to DevOps practices [42]. Such systems are composed of a potentially large number of microservices, being independently developed, tested, deployed, and operated by different teams. This paper focuses on representative load testing of individual microservices. According to component level tests in other contexts [9], in the best case, only the microservice under test is deployed. In this context, the application of the session-based workload extraction and load testing approaches is limited. For example, they assume the deployment and test of the entire system. However, this is not desirable for several reasons, e.g., the large amounts of computing capacities to be provisioned for the load test environment. Moreover, the session-based workload models target the system interface. It seems obvious to use request-based [5] workload extraction approaches on the service level. However, they do not consider the session characteristics of the workload.

In this paper, we adopt the concept of representative session-based load testing from the system level to the microservice level. Based on the workload extraction workflow in Fig. 1, we propose two algorithms for extracting microservice-tailored workload models from system-level workload and microservice architectural information. The first algorithm (*log-based tailoring*) transforms the request logs to generate tailored inputs for the subsequent extractions steps that produce a tailored workload model. The second algorithm (*model-based tailoring*) transforms the system-level workload model based on the architectural information to obtain a microservice-specific workload model.

We evaluate our approach in an experimental study using a microservice-based application. Both algorithms are compared to two baseline algorithms with respect to the representativeness and the costs of the load tests. The results show that the tailoring approaches slightly reduce the representativeness compared to a system-level workload model, but significantly increase it compared to request-based approaches. Furthermore, they can — under certain conditions — reduce the required test execution time and especially reduce the number of microservices to be deployed for the test. The model-based approach performs better than the log-based approach in terms of the representativeness and qualitative characteristics of the generated models, even though the log-based approach might

To appear as: H. Schulz, T. Angerstein, D. Okanović, and A. van Hoorn. Microservice-tailored generation of session-based workload models for representative load testing. In Proceedings of the 27th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2019).

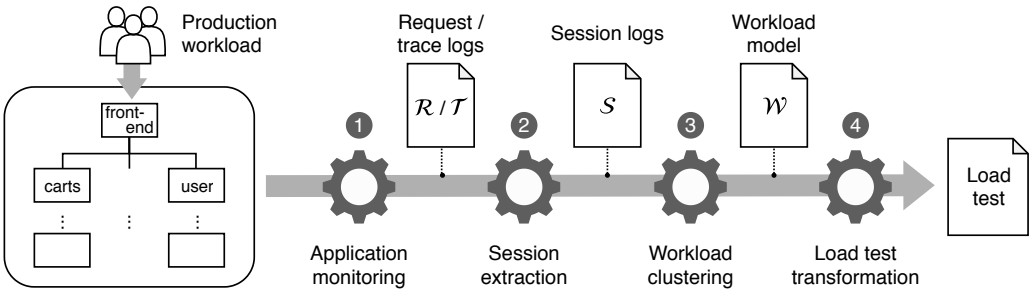


Fig. 1: Load test extraction process.

become preferable for large-scale applications.

The remainder of this paper is organized as follows. Section II details the background of this work and emphasizes the addressed problem. Section III discusses related work. Section IV presents our approach, particularly detailing the two algorithms. Section V includes the experimental evaluation. Section VI concludes the paper and outlines future work.

II. Background and Problem Statement

A typical example for session-based microservice applications are web shops for products that can be browsed and purchased by end users. As illustrated as part of Fig. 1, users interact with such an application via an entry point, e.g., a *front-end* microservice. The response to a user request will then be generated through an interplay of other microservices such as *carts* and *user*. According to a session-based workload [20], [30], user requests are followed by subsequent requests based on the respective reply from the system. Following DevOps practices [42], each microservice is typically developed, (load) tested, and operated by an individual team, with an individual continuous integration and delivery (CI/CD) pipeline. To prepare for the expected production workload, a load test simulates real users by sending requests to the endpoints (e.g., REST endpoints). For load testing the whole application, i.e., the workload arriving at the *front-end* microservice, existing approaches support a tester by extracting representative load tests from user sessions captured in production (e.g., using application performance management (APM) tools) [10], [30], [39], [41]. *However, these approaches are not applicable to test a single microservice such as* carts *individually, because it is not directly invoked by end users and thus, is missing a session context. For ensuring the team's individuality and minimizing the required amount of microservices deployed for the load test as part of the pipeline, we extend the existing WESSBAS approach [41] to fill this gap.*

WESSBAS uses a domain-specific language (DSL) to specify Markov-chain-based workload models, which can be transformed to load tests (④ in Fig. 1). Among other artifacts, a workload model consists of one or several *behavior models* describing how the end users behave. For a set of available endpoints of the application, each of them utilizes a Markov chain with the endpoints as states to describe the order in which requests are submitted. A load test can then submit requests to the application by following the transitions of the chain with the specified transition probabilities and by waiting for additionally specified *think times* per transition. The goal of having multiple behavior models is to increase the representativeness of the simulated users by separating different user behavior, e.g., users mostly browsing vs. users purchasing a lot. The behavior models are automatically extracted by clustering recorded sessions (③ in Fig. 1), with a *behavior mix* describing the relative frequency of each behavior model. *Here, a crucial information is the session ID of the users, allowing to group multiple observed user requests to a session of subsequent related requests (② in Fig. 1).*

Requests to microservices such as *carts* and *user*, which are not directly requested by end users but only by other microservices, do not contain a session ID. Hence, the extraction process in Fig. 1 cannot generate load tests targeting these services. As a workaround, we can change the workload model to a request-based model, which simply defines how often each endpoint is requested per time interval. *However, such a workload model does not take into account the actual request order, which can also influence the microservice's behavior.*

Therefore, we extend the load test extraction process by tailoring certain recorded artifacts to specific microservices for extracting Markov-chain-based, microservice-tailored load tests. For that, we collect traces alongside the request logs (① in Fig. 1) for reusing the session IDs from the user requests. Traces are commonly recorded by monitoring tools and represent the control flow through the application as nested requests [34]. The root request is the entry point to the application, i.e. a request to the *front-end* service. Further requests are invoked by the root request such as to the *carts* service. Furthermore, each request holds timing information, such as a time stamp and response time, and meta information including the session ID. This information can be used to determine the impact of a user request to an endpoint of the *front-end* on the microservices to be tested.

III. Related Work

Related work can be found in the fields of performance testing and, more specific, load testing. We also consider the performance modeling approaches applied in the design phase of software development.

According to the testing pyramid proposed by Cohn [12], testing can be applied at different levels of an application. Unit tests, which focus on testing of particular functions,

are at the bottom like the ones most often applied, while at the top of the pyramid are manual U/I tests and testing in production, as an emerging trend. In the middle are component, integration, and system tests. A similar pyramid can be derived for performance testing. For example, Horky et al. [23] propose to perform performance unit tests, i.e., to test specific functions under a workload, not only with specific inputs. *In this paper, we focus on the middle part of the performance testing pyramid, i.e., on tailored load tests for performance evaluation of particular components, e.g., microservices, and their integration with other components.*

Becker et al. [6] propose an approach for the model-based generation of performance prototypes, which include resource demands and usage models. The approach from Versteeg et al. [40] generates prototype stubs based on the message traces collected in production to support testing in environments such as DevOps. Baltas and Field [4] propose a combined approach which includes performance stubs when the code is not available, to support continuous testing throughout the development lifecycle. Field et al. [19] developed an approach to test components in virtual time and reduce the testing time. They propose to enrich mock objects with performance models to allow for their realistic behavior. To have a time consistency between the mock objects and the real code, the application execution time has to be virtualized too. *In this paper, we propose a complementary approach, which, instead of performance stubs, uses multiple load drivers. A combination of two approaches would allow for a complete isolation of a microservice during the testing.*

To deal with the time required for testing in order to support faster release cycles, some authors propose test case selection and prioritization. *However, they are dealing with functional testing* (as shown by Kazmi et al. [25]), *and do not take into account performance testing specifics, such as operational profile or changes in the incoming workload.*

Some authors argue that the behavior of complex (microservice-based) applications cannot be evaluated at design time [35], i.e., the results are not realistic. Therefore, operational profiles are relevant in software testing, not only for performance, as we show in this work, but also reliability [32], [28] and scalability [3].

The problem we are dealing with in this paper is also known in other fields, e.g., workload decomposition in performance models. The *Forced Flow Law* from operational analysis [16], represented with the equation $X_i = X \cdot V_i$, where V_i is a visit count for the resource i of a system, describes the relationship between throughput of the entire system X and throughput of a particular component/resource of that system X_i. Koziolek [27] proposed modeling languages as a part of the Palladio Component Model [7] to describe user ("usage modeling language") and component behavior ("resource demanding service effect specification"). A PCM instance containing these descriptions is then mapped to a layered queuing network, to be solved. The concept of workload transformation is required when deriving a workload between different modeling formalisms, e.g., between different levels of description [21]. Several works by Cortellessa et al. [14], [15], [8] deal with deriving queuing networks from annotated UML models, where user behavior is described with e.g., activity diagrams, to assess the performance of particular components.

As a prerequisite for operational testing, workload characterization [18] is essential for software performance evaluation. It summarizes and explains workload properties and furthermore, it allows for the generation of realistic synthetic workloads, for both simulation and benchmarking [11]. An approach based on Markov chains is proposed by Menascé et al. [30], to extract Customer Behavior Model Graphs (CBMGs) from the traces in server logs for load testing of e-commerce web sites. Ruffo et al. [36] extend this approach to all web applications. In both approaches, user sessions are used to identify Markov states, representing the user interaction with the system, and transitions between these states, annotated with user think times and probabilities. Because Markov chains cannot model inter-request dependencies in user sessions, Shams et al. [39] developed an approach which uses Extended Finite State Machines (EFSMs). Vögele et al. [41] propose the WESSBAS approach for workload extraction and characterization, combining Markov Chains and EFSMs, to generate executable workload models based on production session logs. The LIMBO approach [26] by von Kistowski et al. allows for modeling of synthetic workloads with variable intensities, but without the modeling of the user behavior. *These approaches, however, focus on the system level, i.e., they require an entire system to be deployed to perform a test. For complex software systems, this requires a significant amount of resources and time.*

To the best of our knowledge, there is no performance-aware testing approach equivalent to functional component tests and integration tests. Load tests tailored based on collected traces close this gap, by allowing only a determined set of services to be deployed and tested.

IV. Workload Model Tailoring

In this section, we propose two alternative algorithms for tailoring a session-based workload model extracted from collected request logs for a specific set of target microservices. We build on Markov-chain-based extraction approaches such as WESSBAS [41], applying the algorithms at different steps of the extraction process presented in Fig. 1. First, we introduce log-based tailoring, which modifies the collected request or session logs. The second algorithm is model-based tailoring, which modifies the extracted workload model. In the following, Section IV-A introduces the notations used and Section IV-B motivates the selection of the two algorithms. Then, the two algorithms are presented in Sections IV-C and IV-D.

A. Notations

An overview of the notations we use in the following is provided in Table I. Corresponding to Fig. 1 from left to right, we consider an application with n microservices, e.g., *front-end*, *carts*, *user*, etc. Each microservice m_i has a set of

TABLE I: Table of Notations

Notation	Explanation
$M = \{m_1, \ldots, m_n\}$	existing microservices, $\|M\| = n$
$\mathcal{M} \subset M$	microservices of a tailored workload model
$E_i = \{e_1, \ldots, e_{k_i}\}$	endpoints of m_i, $\|E_i\| = k_i$
$E = \bigcup_{i=1}^{n} E_i$	endpoints of all microservices
$\mathcal{E} \subset E$	endpoints of a tailored workload model
$\mathcal{R}$	request logs
$\varepsilon : \mathcal{R} \to E$	called endpoint of request
$\phi : \mathcal{R} \to \mathbb{R}$	session ID of request
$t : \mathcal{R} \to \mathbb{R}$	time stamp of request
$\delta : \mathcal{R} \to \mathbb{R}$	duration of request
$\mathcal{T}$	trace logs
$\tau = (r_\tau, R_\tau, C_\tau) \in \mathcal{T}$	trace tree with root request r_τ, vertices R_τ
$C_\tau \subset R_\tau \times R_\tau$	call relations/edges of a trace τ
$\mathcal{S} \subset \mathcal{R}^*$	session logs ($\mathcal{R}^* = \bigcup_{i=1}^{\infty} \mathcal{R}^i$)
$\mathcal{W} = (\{W_1, \ldots, W_q\}, f)$	workload model with q behavior models
$f : \{W_1, \ldots, W_q\} \to [0, 1]$	relative frequency of behavior model
$W_j = (\mathcal{E}_j, p_j, \Delta_j, \mathcal{S}_j)$	behavior model with endpoints $\mathcal{E}_j \subset E$
$I, \$$	initial and final state of a behavior model
$p_j : \mathcal{E}_j \times \mathcal{E}_j \to [0, 1]$	state transition probability
$\Delta_j : \mathcal{E}_j \times \mathcal{E}_j \to \mathcal{N}(\mu, \sigma)$	think time (normal) distribution
$\mathcal{S}_j \subset \mathcal{S}$	sessions aggregated in W_j
$\alpha\Delta_1 * \beta\Delta_2$	convolution of normal distributions Δ_1, Δ_2

endpoints E_i, e.g., *POST /carts/{}/items*. M and E denote the entirety of all existing microservices and endpoints. We use the script font $\mathcal{M}, \mathcal{E}$ for denoting selected subsets used for tailoring a specific workload model.

Request logs are designated as $\mathcal{R}$ and comprise all end user requests that have been observed at the entry points of the application, e.g., the endpoints of the *front-end* microservice. Each individual request $r \in \mathcal{R}$ is characterized by the called endpoint $\varepsilon(r)$, a user session ID $\phi(r)$, a start time stamp $t(r)$, and a duration $\delta(r)$. Trace logs $\mathcal{T}$ add to $\mathcal{R}$ the additional dimension of the requests internally made by the microservices to process the end user request. Each trace $\tau \in \mathcal{T}$ is defined by a directed tree consisting of the root request $r_\tau \in \mathcal{R}$, further requests (nodes) R_τ to the different microservices, and the call relations (edges) C_τ. For convenience, we also use ϕ, t, and δ for traces: $\phi(\tau = (r_\tau, \cdot, \cdot)) := \phi(r_\tau)$.

Session logs $\mathcal{S}$ are extracted from $\mathcal{R}$ (② in Fig. 1) by grouping the requests by the session ID:

$$\forall s \in \mathcal{S}\ \forall r_1, r_2 \in s : \phi(r_1) = \phi(r_2)$$

and by sorting them according to the time stamp:

$$\forall s \in \mathcal{S}\ \forall r_1, r_2 \in s : r_1 \leq r_2 \iff t(r_1) \leq t(r_2)$$

Session logs $\mathcal{S}$ are clustered by similar user behavior and aggregated in a workload model $\mathcal{W}$ (③ in Fig. 1). A workload model consists of one Markov chain W_j — corresponding to a WESSBAS behavior model — per session cluster and a weighting function f defining the workload mix. Each Markov chain is defined by states $\mathcal{E}_j$, a transition probability function p_j, and a think time distribution function Δ_j. Furthermore, we record the session cluster $\mathcal{S}_j$. Finally, we denote the linear combination (convolution [31]) of two normal distributions Δ_1, Δ_2 as follows:

$$\alpha\Delta_1 * \beta\Delta_2 \sim \mathcal{N}(\alpha\mu_1 + \beta\mu_2, \alpha^2\sigma_1^2 + \beta^2\sigma_2^2) \quad (1)$$

B. Overview of Tailoring Approaches

In order to solve the problem of tailoring a workload model to a set of microservices, we are first given the artifacts $\mathcal{R}, \mathcal{T}, \mathcal{S}, \mathcal{W}$ (Fig. 1). In addition, we use the set of microservices $\mathcal{M}$ to be targeted by the load test, and the endpoints $\mathcal{E}$ corresponding to $\mathcal{M}$. The goal is to generate a workload model tailored to the endpoints $\mathcal{E}$ by using the collected traces $\mathcal{T}$.

Based on the load test extraction process, we identify four approaches to reach this goal. The simplest one is to record only the requests arriving at $\mathcal{E}$ and to simply specify the rate at which each endpoint is requested. Such a request-based workload model will be able to replay the mean request rates correctly, but misses relevant session information, which is required for preserving the request orders and inter-request time distributions. Therefore, we aim at developing more elaborate algorithms, but use the request-based workload model as a baseline in our evaluation.

The first proposed algorithm is based on the request logs $\mathcal{R}$, measured at the endpoints requested by the end users and thus, containing the session IDs $\phi(\cdot)$. By using the trace corresponding to each request $r \in \mathcal{R}$, we can replace r by all requests targeting $\mathcal{E}$ while preserving the session ID $\phi(r)$. Such modified request logs only target $\mathcal{E}$ and allow to reuse the remaining steps of Fig. 1.

A second algorithm could address the session logs $\mathcal{S}$ and replace each request similarly to the first algorithm. However, this would lead to exactly the same result. Therefore, we do not differentiate between these two algorithms and only focus on the first one. We denote it as *log-based tailoring*.

Finally, an algorithm can modify the workload model $\mathcal{W}$, which we refer to as *model-based tailoring*. In doing so, the algorithm needs to replace each state — which aggregates several requests to the respective endpoint — in each Markov chain W_j by the aggregate control flow caused by such a request at the endpoints $\mathcal{E}$. For instance, an end user request *POST /orders* to the *front-end* can cause a *POST /orders* request to the *orders* microservice and several requests to the *user* microservice, in potentially varying sequences. This call behavior needs to be reflected in the replacement of the original Markov state.

C. Log-based Tailoring

The log-based tailoring algorithm takes as input recorded trace logs $\mathcal{T}$ — which implicitly contain $\mathcal{R}$ as root requests — and the endpoints $\mathcal{E}$ of a set of microservices $\mathcal{M}$, which the resulting load test should target. The output are request logs $\mathcal{R}$ tailored to $\mathcal{E}$. As described earlier, it is also possible to use the session logs in $\mathcal{S}$ and run the procedure on them, with the same results.

The algorithm works as formalized in Algorithm 1 and illustrated in Fig. 2. We represent three exemplary traces with root requests r_1, r_2, r_3 and further nested requests as layered

Algorithm 1 Extract request logs tailored to endpoints $\mathcal{E}$

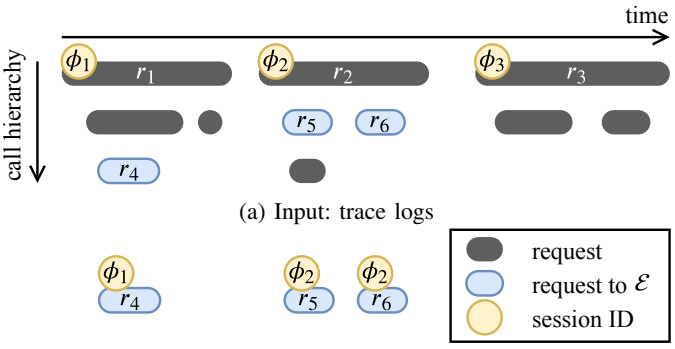
```
1: function TAILORREQUESTLOGS(𝒯, ℰ)
2:     ℛ ← ∅
3:     for all τ ∈ 𝒯 do
4:         ℛ′ ← TAILORREQUEST(τ, ℰ)
5:         φ(ℛ′) ← φ(τ)
6:         ℛ ← ℛ ∪ ℛ′
7:     end for
8:     return ℛ
9: end function
```

Algorithm 2 Tailor root request of trace τ to endpoints $\mathcal{E}$

```
1: function TAILORREQUEST(τ, ℰ)
2:     ℛ′ ← ∅,   T̄ ← {τ}
3:     while T̄ ≠ ∅ do
4:         τ̄ = (r̄, R̄, C̄) ∈ T̄          ▷ arbitrarily selected
5:         T̄ ← T̄ \ {τ̄}
6:         if ε(r̄) ∈ ℰ then
7:             ℛ′ ← ℛ′ ∪ {r̄}
8:         else
9:             T̄ ← T̄ ∪ {(r′, R̄, C̄) | r′ ∈ R̄ ∧ (r̄, r′) ∈ C̄}
10:        end if
11:    end while
12:    return ℛ′
13: end function
```

bars with the time on the horizontal axis and the call hierarchy on the vertical axis. We furthermore consider three requests r_4, r_5, r_6 to target an endpoint in $\mathcal{E}$. The algorithm processes all traces and from each, it collects the requests targeting $\mathcal{E}$ into $\mathcal{R}'$ (line 4). For instance, the trace with root request r_1 results in a single-element set $\{r_4\}$ and r_2 is tailored to $\{r_5, r_6\}$. Furthermore, the session IDs of the collected requests are set to the root requests (line 5), e.g., $\phi(r_5) = \phi(r_6) := \phi_2$.

The request collection from every individual trace is implemented in Algorithm 2. It takes one trace $\tau \in \mathcal{T}$ and the endpoints $\mathcal{E}$ as inputs and performs a breadth-first search for collecting all relevant requests from τ. The algorithm processes a set $\overline{T}$ of sub traces, initialized with τ (line 2) and continues until no more elements are contained in $\overline{T}$. In each iteration, one trace $\overline{\tau}$ is selected and removed from $\overline{T}$ (lines 4 and 5). Then, it is checked whether the root request $\overline{r}$ of $\overline{\tau}$ targets one of the endpoints in $\mathcal{E}$ (line 6). If so, $\overline{r}$ is added to the resulting set of request logs (line 7). Otherwise, all sub traces are extracted from $\overline{\tau}$ and added to $\overline{T}$ (line 9). As an example, the trace with root request r_2 in Fig. 2 has two sub traces with root requests r_5 and r_6. The breadth-first search ensures that only the highest-level requests that target $\mathcal{E}$ are collected, e.g., if r_5 contained another nested request r_7 targeting $\mathcal{E}$ as well, it would not be collected. This is important because, in the load test, the requests to the lower-level requests will be made by the microservices targeted by the higher-level requests and should not be duplicated by the load test.



(a) Input: trace logs

(b) Output: tailored request logs

(c) Legend

Fig. 2: An example of tailoring requests r_1, r_2, r_3 to $\mathcal{E}$ (Algorithms 1 and 2).

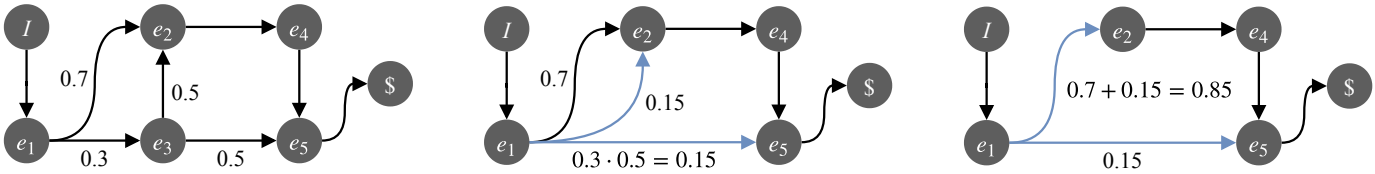
Algorithm 3 Tailor workload model $\mathcal{W}$ to endpoints $\mathcal{E}$

```
1: function TAILORWORKLOADMODEL(𝒲, 𝒯, ℰ)
2:     for all W_j = (ℰ_j, p_j, Δ_j, 𝒮_j) ∈ 𝒲 do
3:         for all e ∈ ℰ_j, e ∉ ℰ do
4:             𝒯′ ← {(r_τ, R_τ, C_τ) ∈ 𝒯 |
                       ∃s ∈ 𝒮_j ∃r ∈ s : r_τ = r ∧ ε(r_τ) = e}
5:             φ(𝒯′) ← {1, ..., |𝒯′|}
6:             ℛ′ ← TAILORREQUESTLOGS(𝒯′, ℰ)
7:             if ℛ′ = ∅ then
8:                 REMOVESTATE(W_j, e, δ(𝒯′))
9:             else
10:                ℛ′ ← ℛ′ ∪ {(I′, φ(τ), t(τ), 0) | τ ∈ 𝒯′}
11:                ℛ′ ← ℛ′ ∪ {($′, φ(τ), t(τ) + δ(τ), 0) | τ ∈ 𝒯′}
12:                REPLACESTATE(W_j, e, ℛ′)
13:            end if
14:        end for
15:    end for
16: end function
```

D. Model-based Tailoring

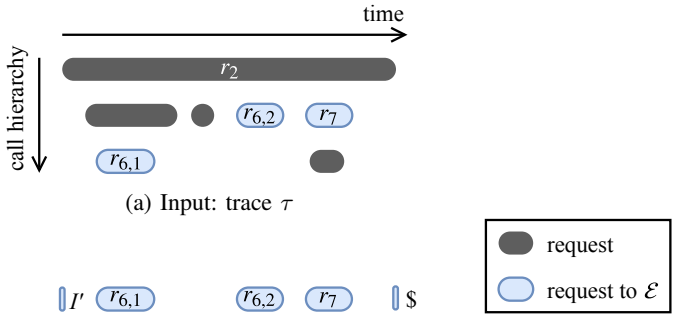
Instead of performing tailoring on the collected traces, the second algorithm targets the workload model $\mathcal{W}$. While there are several workload modeling formalisms in the literature ([11], [17]), we chose to use the WESSBAS DSL [41], which models $\mathcal{W}$ as multiple Markov chains (behavior models) W_j with a relative frequency $f(W_j)$. Each state in a chain represents requesting a certain endpoint, as shown in Fig. 3a. Transitions in a chain are characterized by the transition probability, e.g., $p_j(e_1, e_2) = 0.7$, and a normally distributed think time $\Delta_j(e_1, e_2)$ (not shown in the figure).

Algorithm 3 implements the model-based tailoring. It takes the workload model $\mathcal{W}$, the collected traces $\mathcal{T}$, and the set of target endpoints $\mathcal{E}$ as input and modifies each Markov chain W_j of $\mathcal{W}$. Based on the traces and the session logs $\mathcal{S}_j$ corresponding to the Markov chain, it replaces all states that are not contained in $\mathcal{E}$ by new endpoints from $\mathcal{E}$. A replacement needs to model the application's control flow that is caused by a request of the original state in another Markov chain. For that, the algorithm iterates over all such



(a) Input: Markov chain W_j (b) Step 1: concatenate transitions via e_3 (c) Step 2: merge duplicate transitions

Fig. 3: Illustration of REMOVESTATE(W_j, e_3, δ) (Algorithm 4).



(a) Input: trace τ (b) Output: request logs with I' and \$′ (c) Legend

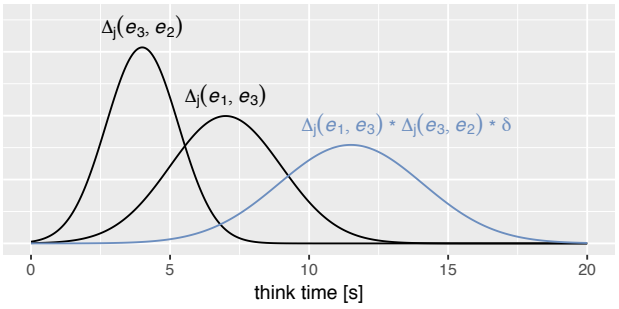
Fig. 4: Extracting tailored request logs from a trace in Algorithm 3.

states (line 3) and collects all traces from $\mathcal{T}$ whose root requests are contained in a session in $\mathcal{S}_j$ (line 4). Then, it sets the session IDs of these traces to unique values (line 5) and reuses Algorithm 1 for extracting request logs tailored to $\mathcal{E}$, as illustrated in Fig. 4. Given a trace with root request r_2 is processed and there are nested requests $r_{6,1}, r_{6,2}, r_7$ to endpoints in $\mathcal{E}$, the resulting request logs will consist of $\{r_{6,1}, r_{6,2}, r_7\}$. Because of the changed session IDs, $\mathcal{R}'$ represents the control flow at the endpoints in $\mathcal{E}$ from the perspective of the original state rather than an end user's perspective. Depending on $\mathcal{R}'$, the original state is either removed if $\mathcal{R}'$ does not cause any requests on $\mathcal{E}$ (line 8), or replaced by a Markov chain extracted from $\mathcal{R}'$ (line 12). In the former case, the aggregated duration of the requests to the original state — represented by a normal distribution $\delta(\mathcal{T}')$ — is passed as a parameter for preserving the delay introduced by the state. In the latter case, placeholder requests I' and \$′ are added (lines 10 and 11; Fig. 4b), which will ensure that the new Markov chain will take the same time as the original state does. Both are important for preserving the overall timing of W_j. In the following, we describe the algorithms for removing and replacing states.

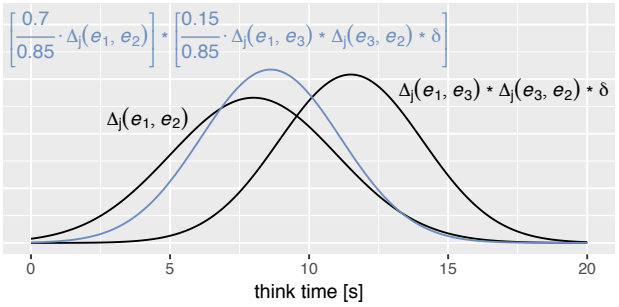
Removing a state e with aggregate duration δ from a Markov chain W_j is implemented in Algorithm 4 and illustrated in Fig. 3. In this example, we want to remove state e_3. In a first step, the algorithm calculates the expected number of steps the Markov chain loops in the state e — given that such a cycle exists — based on the geometric series $\sum_{i=1}^{\infty} p_j(e,e)$ (line 2). Because e will be removed, we need to ensure that the time spent in the loop will be preserved in the chain.

Algorithm 4 Remove state e with normally distributed duration δ from Markov chain W_j

1: **function** REMOVESTATE($W_j = (\mathcal{E}_j, p_j, \Delta_j, \mathcal{S}_j), e, \delta$)
2: $\alpha \leftarrow \frac{1}{1-p_j(e,e)} - 1$ ▷ geometric series [22]
3: **for all** $e' \in \mathcal{E}_j$ **do**
4: $p_j(e,e') \leftarrow \frac{p_j(e,e')}{1-p_j(e,e)}$
5: $\Delta_j(e,e') \leftarrow \Delta_j(e,e') * \alpha \cdot \Delta_j(e,e)$
6: **end for**
7: **for all** $e', e'' \in \mathcal{E}_j$ **do**
8: $p' \leftarrow p_j(e',e'') + p_j(e',e) \cdot p_j(e,e'')$
9: $\Delta_j(e',e'') \leftarrow \left[\frac{p_j(e',e'')}{p'} \cdot \Delta_j(e',e'')\right]$
$* \left[\frac{p_j(e',e) \cdot p_j(e,e'')}{p'} \cdot \Delta_j(e',e) * \Delta_j(e,e'') * \delta\right]$
10: $p_j(e',e'') \leftarrow p'$
11: **end for**
12: $\mathcal{E}_j \leftarrow \mathcal{E}_j \setminus \{e\}$
13: **end function**

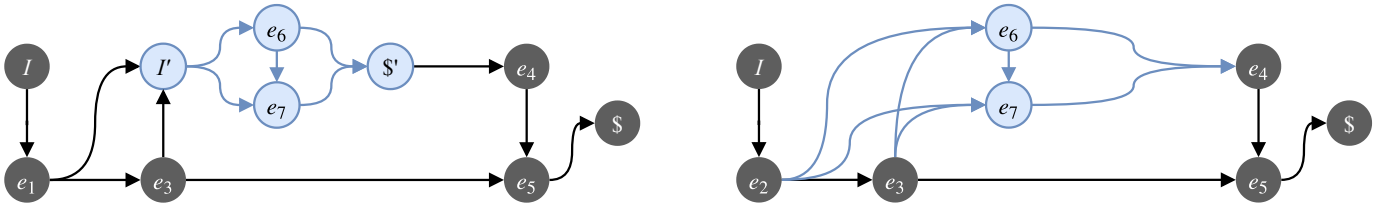


(a) Step 1: concatenate (e_1, e_3) and (e_3, e_2)



(b) Step 2: merge the two transitions (e_1, e_2)

Fig. 5: Illustration of the think time calculation in REMOVESTATE$(W_j, e_3, \delta \sim \mathcal{N}(0.5, 1))$ (Algorithm 4).



(a) Step 1: replace e_2 with Markov chain

(b) Step 2: remove I' and $\$'$ using REMOVESTATE

Fig. 6: Illustration of REPLACESTATE$(W_j, e_2, \mathcal{R}')$ (Algorithm 5).

Algorithm 5 Replace state e of Markov chain W_j with a Markov chain derived from tailored requests $\mathcal{R}'$

```
1:  function REPLACESTATE(W_j = (E_j, p_j, Δ_j, S_j), e, R')
2:      S' ← GROUPTOSESSIONS(R')
3:      W' = (E', ·, ·, ·) ← AGGREGATE(S')
4:      E_j ← E_j ∪ E'
5:      for all e' ∈ E_j do
6:          p_j(e', I') ← p_j(e', e)
7:          Δ_j(e', I') ← Δ_j(e', e)
8:      end for
9:      for all e' ∈ E_j do
10:         p_j($', e') ← p_j(e, e')
11:         Δ_j($', e') ← Δ_j(e, e')
12:     end for
13:     E_j ← E_j \ {e}
14:     REMOVEMARKOVSTATE(W_j, I', 0)
15:     REMOVEMARKOVSTATE(W_j, $', 0)
16: end function
```

For that, we remove the cycle by appending the think time to all outgoing transitions from e and by normalizing the probabilities such that they sum to 1 (lines 3-6). Then, we concatenate all incoming and outgoing transitions to/from e (lines 7-11). This is done in two logical steps. First, the transitions are concatenated, e.g., (e_1, e_3) and (e_3, e_2) to (e_1, e_2) and (e_1, e_3) and (e_3, e_5) to (e_1, e_5) (Fig. 3b). The think times are convolved accordingly, as illustrated in Fig. 5a, furthermore including the original state's duration δ. In the second step, potential duplicate transitions are merged, e.g., between e_1 and e_2 (Fig. 3c). The think times are convolved again, weighted by the relative transition probabilities (Fig. 5b). Please note that the such resulting think time deviation is not a stochastically valid junction of the original think times. However, as the proper junction is not a normal distribution but the workload model requires it, this is the closest we can achieve. At least, the think time mean will be correct.

When a state is to be replaced, a new (sub-)chain is to be inserted instead of the original state. This is implemented in Algorithm 5 and illustrated in Fig. 6 at the example of replacing state e_2 (see Fig. 3a). The algorithm takes as input the Markov chain W_j, the state e to be replaced and the tailored request logs $\mathcal{R}'$ prepared in Algorithm 3. First, it groups the request logs into session logs and aggregates them into a new Markov chain W' (lines 2 and 3). Considering request logs related to our previous example (Fig. 4b) and assuming requests $r_{6,\cdot}$ target endpoint e_6 and r_7 targets e_7, the new chain will contain states $\{I', e_6, e_7, \$'\}$. Then, this chain replaces e in two steps. In the first step, e (in the example e_2) is replaced by the new chain by changing the target and source states of the transitions of e (lines 4-8; Fig. 6a). In the second step, the artificially inserted states I' and $\$'$ are removed using Algorithm 4 (lines 13-15; Fig. 6b). As a result, the incoming and outgoing transitions of the new chain will contain the duration of the original request as think times.

V. EVALUATION

We evaluate the two introduced workload model tailoring algorithms in an experiment series. We use the Sock Shop Microservices Demo,[1] because Aderaldo et al. [1] have argued it to be representative for microservice applications. In doing so, we address the following research questions:

RQ1 How representative are the workloads generated by the tailored load tests compared to an untailored and a request-based test?

The first question aims at evaluating potential differences of the workload arriving at the microservices under test due to the tailoring of the workload models. For measuring the representativeness, we use a distance metric based on the Kolmogorov-Smirnov statistic. As the goal is to generate representative tailored tests, the difference should be small. As baselines, we use an (untailored) system-level test generated by WESSBAS and the request-based load tests described in Section IV-B.

RQ2 To which degree do the tailored load tests impair the performance metrics of the tested microservices?

Instead of evaluating the workload characteristics directly, we compare performance metrics such as response time, CPU utilization, and memory consumption, which are influenced by the workload.

RQ3 How much can tailoring reduce the test execution time until measured performance metrics are stable?

When testing a set of microservices in isolation rather than in combination with the remainder of the application, the overall complexity of the system is reduced. This indicates that a shorter test duration might be feasible for reaching a certain

[1] https://microservices-demo.github.io/

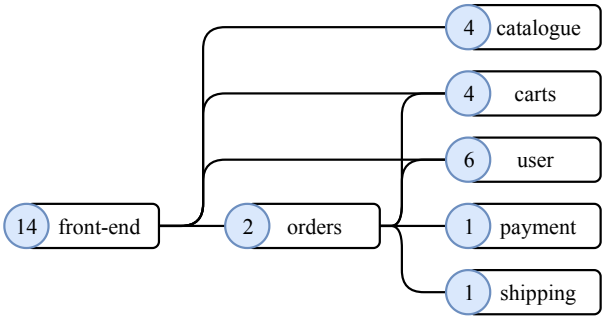


Fig. 7: Dependencies between the Sock Shop microservices (Ⓧ = number of endpoints of the respective microservice).

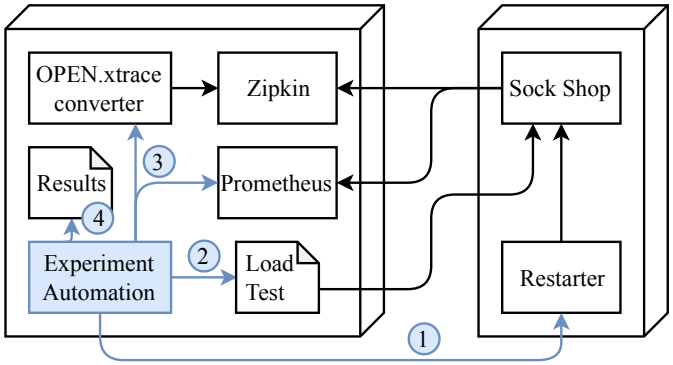


Fig. 8: Experiment setup for load test execution.

stability of measured metrics compared to an untailored test. Thus, we evaluate the ability of the tailoring approaches to reduce the required test duration, assessed by a metric based on the variation of measured response time medians.

RQ4 Which qualitative differences of the tailored workload models exist?

Finally, we analyze the qualitative differences of the log-based and model-based tailored workload models generated during the experiment. As they are generated differently, attributes such as understandability and complexity can differ.

In the following, we introduce the system under test (SUT) in more detail, present the methodology of the experiment series, introduce the metrics used in the evaluation, present the results, and discuss the results according to the research questions. Finally, we discuss the threats to validity. A replication package can be found online [38].

A. System under Test

In our experiment series, we use the aforementioned Sock Shop Microservices Demo (Sock Shop) as SUT, on which we execute the generated load tests. The Sock Shop constitutes a web shop that allows browsing, adding to a cart, and purchasing. User account management is also available. The following microservices exist (with databases for some): *front-end*, *catalogue*, *carts*, *orders*, *payment*, *shipping*, and *user*. The dependencies between the microservices are depicted in Fig. 7. In addition to the Sock Shop itself, we utilize the open-source monitoring systems Zipkin[2] for trace collection, a Java service converting the Zipkin traces into OPEN.xtrace [34] to allow for monitoring tool independence, and Prometheus[3] for collecting performance metrics.

As illustrated in Fig. 8, we deploy the Sock Shop on a bare-metal machine with 80 cores (2 threads each) at 2300 MHz, 896 GB RAM, and a magnetic disk with 15 000 rpm. Each microservice is deployed as a Docker container with two isolated CPU cores and 4 GB of RAM. In addition, the machine hosts a lightweight Java service for restarting the application remotely. Zipkin, the OPEN.xtrace converter, and Prometheus are deployed on a second machine with 24 cores (2 threads each) at 2300 MHz and 32 GB RAM, connected via a shared 10 Gbit/s network infrastructure. The second machine also hosts the JMeter[4] load tests that are executed and a script automating the experiment (see below). JMeter has a heap size of 512 MB, except for the untailored test, which needs 2 GB.

B. Methodology

Our evaluation consists of an experiment series in three steps, which we describe in the following.

1) Simulate Production Workload: As representative load testing uses production monitoring data for generating load tests, we need to simulate production workload first. For that, we designed a load test mimicking three different types of users, namely users that visit products (80 users), browse and buy products (60 users), and visit the status of their orders (60 users). An experiment automation script executes this load test as depicted in Fig. 8. First, it restarts the Sock Shop to ensure that the load test is executed in a clean and comparable environment ①. Then, the load test is executed ②. During the load test, the Sock Shop sends traces to Zipkin and CPU and memory metrics to Prometheus. After the load test has finished, the metrics and traces — via the OPEN.xtrace converter, which retrieves the Zipkin traces — are collected ③ and stored into a results folder for later analysis ④. In the following, we will refer to the workload (model) of this load test as *reference workload*.

2) Generate Load Tests: After the reference workload has been executed, the collected traces are used to generate tailored load tests by using the log-based and model-based approaches, which we implemented as part of the ContinuITy project. The code is available online [13]. As baselines, we also generate an untailored (system-level) load test using the plain WESSBAS and request-based load tests simply replaying all requests at a rate extracted from the reference workload. We apply the request-, log-, and model-based approaches for generating load tests for the following combinations of microservices, considering the dependencies shown in Fig. 7:

- catalogue — 4 endpoints
- user — 6 endpoints

[2]https://zipkin.apache.org/
[3]https://prometheus.io/
[4]http://jmeter.apache.org/

- carts, orders, payment, shipping, user — 10 endpoints
- catalogue, user — 10 endpoints
- catalogue, carts, orders, payment, shipping, user — 14 endpoints
- catalogue, carts, user — 14 endpoints
- catalogue, carts, payment, shipping, user — 16 endpoints

Please note that we always include dependent microservices, i.e., for the *orders* service, we also include *carts*, *payment*, *shipping*, and *user*. If *orders* is to be tested in isolation, load-test-ready stubs need to replace the dependent services, which can however influence *order*'s performance. Therefore, we use the actual services as "perfect" stubs.

3) Execute Load Tests: The last step is the execution of the generated load tests. Again, we utilize the setup depicted in Fig. 8. The experiment automation takes care of executing all tests for 30 minutes and restarting the Sock Shop before each test to yield comparable results.

C. Metrics

For measuring several attributes of the experiment results, we utilize existing and newly introduced metrics, which we present in the following.

1) Workload Distance Metric: For measuring the representativeness of a generated load test (RQ1), we introduce a distance metric comparing the test's results with the reference workload. Even though we focus on session-based workloads in this paper, the tailored load tests target microservices that do not depend on sessions. Thus, we cannot use session information in the distance metric. Instead, we use the arrival rate of requests or rather its reciprocal, the inter-arrival time.

Given we collected two samples $X_{\text{ref},e}$ and $X_{\text{gen},e}$ of inter-arrival times for an endpoint e for the reference and generated workload, we calculate the Kolomogorov-Smirnov statistic D_e for the significance level α. D_e is a measure for the distance of the inter-arrival time distribution for e. The Kolmogorov-Smirnov test rejects a null hypothesis $H_0 : F_{X_{\text{gen},e}}(x) = F_{X_{\text{ref},e}}(x)$, if the following holds for the critical value $c(\alpha)$ [29]:

$$D_e > c(\alpha) \cdot \sqrt{\frac{|X_{\text{ref},e}| + |X_{\text{gen},e}|}{|X_{\text{ref},e}| \cdot |X_{\text{gen},e}|}}$$

Hence, we yield an aggregate measure for all endpoints $\mathcal{E}$ involved in one load test by calculating the weighted average of the D_e:

$$D := \frac{1}{|\mathcal{E}|} \sum_{e \in \mathcal{E}} D_e \cdot \sqrt{\frac{|X_{\text{ref},e}| \cdot |X_{\text{gen},e}|}{|X_{\text{ref},e}| + |X_{\text{gen},e}|}}$$

2) Performance Metrics: For assessing the influence of potentially less representative load tests, we compare performance metrics (RQ2). Precisely, we analyze the response times of the requests, the CPU utilization, and the memory consumption during each load test. We extract the response times from the collected traces. The CPU and memory metrics are collected by Prometheus in 5 second granularity.

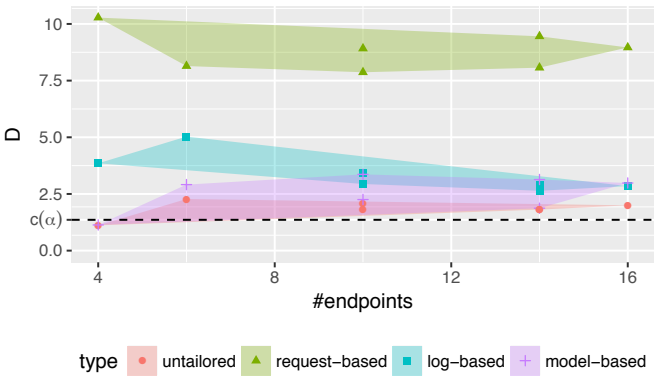


Fig. 9: Aggregated Kolmogorov-Smirnov statistic D.

3) Test Duration Metric: For measuring the test duration that is required for reliable results (RQ3), we base on the performance metrics. Presuming a steady-state load, aggregate measures such as the median response time at test end are of interest. Therefore, we determine the required test duration by comparing the median response time calculated at a certain time stamp with the final value. Given a sample of median response times $(x_1^{(e)}, \ldots, x_n^{(e)})$ per second for endpoint e, we calculate the distance to the finally determined median response time $x_n^{(e)}$ at index i:

$$\widetilde{x}_i^{(e)} := \frac{|x_i^{(e)} - x_n^{(e)}|}{x_n^{(e)}}$$

Then, we define the required test duration for $\beta = 0.01$ and tested endpoints $\mathcal{E}$ as

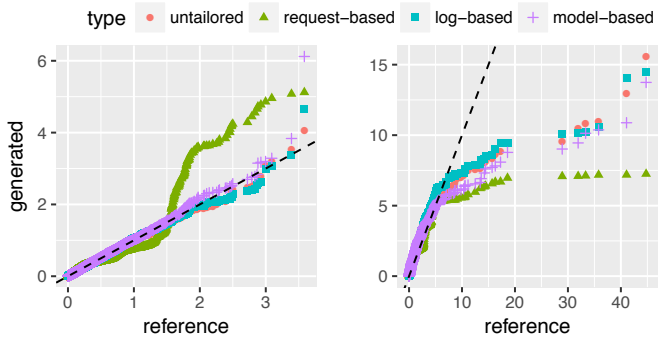
$$\max_{e \in \mathcal{E}} \left(\min\{i \mid \forall j \geq i : \widetilde{x}_j^{(e)} \leq \beta\} \right)$$

D. Results

In this section, we present the results of the experiment, separated by research question. For convenience, we denote the load tests transformed from a workload model by the used tailoring approach, e.g., *log-based load test* or *untailored load test*.

1) Representativeness: For assessing the representativeness of the executed load tests, we calculate the distance metric D for each of them. In addition, we distinguish between the number of endpoints involved in the load test, because we expect it to influence D. The resulting values including the convex hull per tailoring approach and a line for the critical value $c(\alpha) = 1.36$ [29] for $\alpha = 0.05$ are shown in Fig. 9. It can be seen that mostly all values are greater than $c(\alpha)$, indicating a significant difference. However, for our approaches, D is only slightly greater than for the untailored test, but clearly less than for the request-based tests, which increase D by a factor between 3.57 and 9.22 compared to the untailored test.

In general, the model-based load tests generate more representative results — they increase the distance by a factor in the range of 1.02 to 1.86 — than the log-based tests with factors between 1.42 and 3.46. However, the log-based approach



(a) *catalogue GET /catalogue/size* (b) *orders POST /orders*

Fig. 10: Q-Q plots of the inter-arrival times compared to the reference workload for different endpoints.

becomes better with an increased number of endpoints, while the model-based approach has a slight but ambiguous upward trend. When testing 16 endpoints, the difference between the two approaches is negligible.

For investigating the difference of the inter-arrival time distributions in more detail, we investigate quantile-quantile (Q-Q) plots (Fig. 10). For shorter inter-arrival times, e.g., in case of the *GET /catalogue/size* endpoint when testing the *catalogue* microservice individually (4 endpoints), it can be seen that the request-based test has a clearly different distribution than the reference test. The other generated tests are close to the reference test. For longer inter-arrival times such as the *POST /orders* endpoint measured when testing the *orders* microservice in combination with the dependent ones, the tail of all distributions is different than the one of the reference test. However, the tails of the untailored, log-based, and model-based tests appear similar.

2) Performance Metrics: Table II provides a summary of t-tests and Cohen's d applied to the response times, CPU utilization, and the change of used memory per second. Each t-test compares the measurements of a tailored test with the untailored one with $H_0 : F_X(x) = F_0(x)$ and $H_A : F_X(x) \neq F_0(x)$. The numbers count the occurrence of a certain combination of a significant difference detected by the t-test and an effect size. For all metrics, there are cases where H_0 is rejected and others in which no significant difference is detected. The most differences — including larger effect sizes — are detected in the CPU utilization, which can be attributed to the naturally large fluctuation of this metric. For the response times and the memory, the most frequent effect size is negligible. Between the three tailoring approaches, no clear difference can be seen, even though for the response times, there are more cases with a significant difference for the request-based test.

3) Required Test Duration: Fig. 11 shows the test duration required until the final median response time is reached except for an error of 1 % in relation to the number of tested endpoints. We use the untailored test as a baseline and include the convex hull for illustrating the overall relation.

TABLE II: Summary of the Statistical Tests

t-test Cohen's d	not sign. negligible	sign. negligible	sign. small	sign. medium	sign. large
response time					
request-based	19	31	24		
log-based	28	20	26		
model-based	29	23	22		
CPU utilization					
request-based	7		7	7	28
log-based	12		6	9	22
model-based	9		8	7	25
memory change per second					
request-based	38	3	8		
log-based	33	10	6		
model-based	36	8	5		

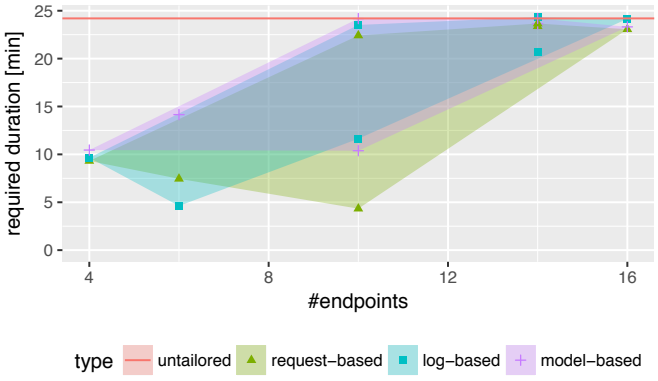


Fig. 11: Required test duration per number of tested endpoints and test type.

We calculate the duration based on the response times of the tested endpoints, i.e., for the untailored test, we consider the endpoints of the *front-end* service, while we use the endpoints of the respective microservices for the tailored tests. It can be seen in the figure that the untailored test requires the longest execution time with 24.2 minutes, except for the log-based test for one service combination (*catalogue, carts, orders, payment, shipping, user*), which needs 24.33 minutes. In general, less endpoints under test reduce the required test duration. The shortest duration of 4.33 minutes could be reached by the request-based test for the *catalogue* and *user* service (10 endpoints), followed by the log-based test for the *user* service (6 endpoints) with 4.62 minutes. Furthermore, among the tailoring approaches, the request-based approach tends to require the shortest duration while the model-based approach requires the longest duration. However, the difference becomes small for an increasing amount of endpoints.

4) Qualitative Differences: Analyzing the behavior model that have been generated during the experiment series, we identify thee major differences between the log-based and the model-based tailoring approaches. First, the number of Markov chains (behavior models) is different: while the model-based algorithm reuses the Markov chains generated by the untai-

lored approach — which are five in our case — the log-based approach generated five chains, too, but only three for the *users* microservice tested in isolation. Hence, the clustering of the session logs is different. In general, even though the number of Markov chains is equal for most of the tests, it cannot be assumed that the clustering is equal.

The second difference is the number of states of each Markov chain. The log-based approach always has as many states as the number of endpoints involved in the test. In contrast to that, the model-based approach can have more endpoints. For instance, the *user GET /customers/id* endpoint is used in the replacements of the *GET /customers/id* and *POST /orders* endpoints of the *front-end* microservice. Hence, it occurs two times in the tailored Markov chain. In our experiments, the chains of the model-based tests have between $25\,\%$ and $50\,\%$ ($34\,\%$ on average) more states than the log-based approach.

Finally, the states of the model-based tailored Markov chains correspond to states of the untailored chains. By tracking the origin of each state — e.g., we keep the original state's name in the name of the new state — this relationship allows for analyzing and changing the Markov chain based on end user behavior, e.g., removing a certain end user request due to a changed API. In contrast to that, the session logs tailored by the log-based algorithm are newly created by clustering, resulting in completely different Markov chains.

E. Discussion

In the following, we discuss the research questions defined in the beginning of this section based on the results. For each research question, we highlight the key message in a box.

1) Representativeness — RQ1:

> Our introduced tailoring approaches slightly reduce the representativeness compared to an untailored load test, but significantly increase it compared to a simple request-based load test. The log-based load tests are less representative than the model-based tests, but become increasingly representative with more tested endpoints.

The reduction of the representativeness is reflected in the aggregated Kolmogorov-Smirnov statistic D, which is however only slightly higher than for the untailored test. In contrast to that, the request-based approach is more than two times less representative. We attribute this finding to the following. In the model-based approach, we merge state transitions of the Markov chains. Because the workload model is based on normal distributions, we need to convolve the think time distributions into another normal distribution (see Fig. 5b), which is an improper operation. While the mean think time remains correct, the variance and distribution function is impaired. Hence, the think time between two requests can be different than in the untailored load test, which influenes the inter-arrival times.

In the log-based approach, all applied operations are valid. However, the session logs that are clustered after the tailoring are different than without tailoring. Thus, the clustering can be different, resulting in different Markov chains. This reasoning is underlined by the fact that the log-based load test for the *user* microservice has only three chains while the untailored test has five. For a larger number of tested endpoints, the difference in the Markov chains appears to be less critical.

2) Performance Metrics — RQ2:

> The performance metrics support the findings of RQ1: there are small differences between the tailored load tests and the untailored one. However, the larger differences of the request-based tests are not reflected.

This was identified by t-tests and Cohen's d. The fact that the request-based tests do not clearly entail more different performance metrics might stem from the fact that the average number of requests per endpoint and per second is the same as in all other tests. This might lead to a very similar behavior of the Sock Shop. Regardless of that, these findings do not falsify the conclusion made for RQ1, but rather support it.

3) Required Test Duration — RQ3:

> A larger number of tested endpoints requires a longer test duration. For a small number of endpoints, the request- and log-based approaches require the shortest durations, but this is at the expense of representativeness, especially for the request-based approach.

We can derive this finding from the calculated test duration metrics, but also need to take into account that there can be individual endpoints that require a longer duration. For instance, the *POST /orders* endpoint of the *orders* service required between 22.42 and 24.33 minutes while the *GET /customers/* endpoint of the *user* service required only between 8 seconds and 4.97 minutes. This finding indicates that the longer duration is not only due to the sheer number of endpoints, but also to the higher likelihood to include an endpoint requiring the longer duration.

In addition, we would like to highlight that the tailoring approaches can only reduce the test duration from the perspective of all tested microservices. If only a certain set of microservices, e.g., *user*, is considered to determine the test duration also in the untailored test, the tailoring approaches cannot reduce it without impairing the representativeness, as it would require the workload arriving at *user* to be changed. In contrast to that, the tailoring approaches can always save resources, because only the tested microservices including its dependents need to be deployed. By combining them with stubbing approaches [6], [4], [19], [40], the dependent microservices can be removed from the deployment as well.

Concluding, the fewer services are included in a load test, the more resources can be saved by only deploying the tested services and potentially stopping the test after a shorter time. For that, we suggest using existing approaches [2].

4) Qualitative Differences — RQ4:

> The model-based tailoring approach generates Markov chains with states that correlate with end user requests and thus, can better explain the effect of the end user's behavior. In contrast to that, the log-based approach does not allow for such analyses. As a drawback, the model-based approach generates Markov chains with 36 % more states in average.

Even though the sizes of the Markov chains in our experiment are not critical, it can become memory-critical for large-scale applications. Furthermore, duplicated states hinder manual maintainability. However, as the Markov chains are generated automatically and not meant to be changed manually in the first place, this drawback is less relevant. Therefore, our results indicate that the model-based approach is preferable over the log-based approach regarding qualitative attributes.

F. Threats to Validity

We identify the following threats to the validity of our work.

1) Conclusion Validity: In the analysis of the performance metrics, we applied the t-test for detecting significant differences between the tailored and untailored tests. These metrics are not necessarily normally distributed, which is an assumption of the t-test. However, as the sample sizes are large with at least 305 entries, the t-test can be used according to the central limit theorem.

Furthermore, performance metrics such as CPU utilization can be fluctuating and thus, unreliable in general. Therefore, the conclusions we have drawn are based on a combination of several performance metrics and also workload metrics.

2) Internal Validity: Our evaluation consists of a series of experiments, which were executed automatically in a sequence. For preventing influences of former test executions, the tested Sock Shop application and the JMeter load driver have been restarted before each execution, including a completely new deployment of the Sock Shop. For preventing interactions between microservices running in parallel, we applied CPU pinning and memory reservation.

Another potential threat is that the metrics measured during the reference workload appeared to bear small inconsistencies. This manifested with the steady increase of the response times of the *carts* microservice. During all generated load tests, no such effect could be observed. Hence, the comparison to the reference test is impaired. However, as the input, i.e., the request logs, for all workload model generation approaches was the same, we presume no side effects regarding the comparability among these tests. Also, we could not detect any trends in the request rates of the reference workload, justifying the steady-state execution of the generated load tests.

3) Construct Validity: In this paper, we assume normally distributed think times in the Markov chains modeling the workload. In general, think times do not need to be normally distributed and workload modeling languages such as the WESSBAS DSL allow for different distribution functions as well. We chose the normal distribution, because it is commonly used in related works. Evaluating other distribution functions is left for future work.

4) External Validity: As the Sock Shop is not an industrial application, it is questionable whether it can represent real-world microservice applications. However, we base on an existing study, which assessed the Sock Shop to be representative [1]. In future work, we plan to evaluate the tailoring algorithms with another industrial application.

VI. Conclusion

In this paper, we presented two algorithms — log-based and model-based tailoring — for generating microservice-tailored workload models for load testing. In contrast to existing approaches for representative workload model generation [30], [36], [39], [41], such tailored workload models allow to test a certain subset of all microservices of an application in isolation. Hence, with regard to independent microservice development teams, each development team can test their service without deploying the whole application. Services that are not affected by the test will not be deployed, saving the resources required to run the test. In addition, our approach is automated, in order to simplify an implementation into a continuous delivery pipeline.

Our experimental evaluation with a microservice application showed that the proposed algorithms do not only allow for resource-efficient load testing, but also generate more representative workload than a simple request-based workload model. Furthermore, they could reduce the required test execution time compared to a system-level test, especially for combinations of microservices under test with a low number of endpoints. The model-based approach appeared to outperform the log-based approach concerning representativeness and understandability of the generated workload models, even though the log-based approach became better for larger numbers of tested endpoints.

For future work, we are going to apply the tailoring algorithms to industrial microservice applications. Furthermore, we suggest evaluating other distribution functions for modeling the think times, as they appeared to bear limitations for the model-based tailoring. The WESSBAS DSL allows for that in general, even though we based on the commonly used normal distributions in this paper. Finally, it is of interest to investigate the load tests tailored with the proposed algorithms in combination with their natural complement of performance stubs. In doing so, a full isolation of one microservice from the remainder of the application can be achieved.

VII. Acknowledgements

This work is being supported by the German Federal Ministry of Education and Research (grant no. 01IS17010, ContinuITy) and the European Union's Horizon 2020 research and innovation programme (grant no. 825040, RADON). The authors would like to thank the HPI Future SOC Lab for providing the infrastructure.

References

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

[2] Alghmadi, H.M., Syer, M.D., Shang, W., Hassan, A.E.: An automated approach for recommending when to stop performance tests. In: Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME 2016). pp. 279–289. IEEE Computer Society (2016)

[3] Avritzer, A., Ferme, V., Janes, A., Russo, B., Schulz, H., van Hoorn, A.: A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing. In: Software Architecture - 12th European Conference on Software Architecture (ECSA 2018). pp. 159–174 (2018)

[4] Baltas, N., Field, T.: Continuous Performance Testing in Virtual Time. In: Proceedings of the 9th International Conference on Quantitative Evaluation of Systems (QEST 2012). pp. 13–22. IEEE Computer Society (2012)

[5] Barford, P., Crovella, M.: Generating representative web workloads for network and server performance evaluation. In: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems (SIGMETRICS/PERFORMANCE '98). pp. 151–160 (1998)

[6] Becker, S., Dencker, T., Happe, J.: Model-driven generation of performance prototypes. In: Performance Evaluation: Metrics, Models and Benchmarks. pp. 79–98 (2008)

[7] Becker, S., Koziolek, H., Reussner, R.: The palladio component model for model-driven performance prediction. J. Syst. Softw. **82**(1), 3–22 (2009)

[8] Bernardo, M., Cortellessa, V., Flamminj, M.: TwoEagles: A model transformation tool from architectural descriptions to queueing networks. In: Proceedings of the 8th European Performance Engineering Workshop (EPEW 2011). Lecture Notes in Computer Science, vol. 6977, pp. 265–279. Springer (2011)

[9] Black, R., Van Veenendaal, E., Graham, D.: Foundations of Software Testing—ISTQB Certification. Cengage Learning EMEA, 3rd edn. (2012)

[10] Cai, Y., Grundy, J.C., Hosking, J.G.: Synthesizing client load models for performance engineering via web crawling. In: Proc. ASE 2007. pp. 353–362 (2007)

[11] Calzarossa, M.C., Massari, L., Tessera, D.: Workload characterization: A survey revisited. ACM Computing Surveys **48**(3), 48:1–48:43 (2016)

[12] Cohn, M.: Succeeding with Agile: Software Development Using Scrum. Addison-Wesley Professional (2009)

[13] ContinuITy Project: ContinuITy (2019), https://github.com/ContinuITy-Project/ContinuITy, accessed: 2019-05-27

[14] Cortellessa, V., Mirandola, R.: Deriving a queueing network based performance model from UML diagrams. In: Workshop on Software and Performance. pp. 58–70 (2000)

[15] Cortellessa, V., Mirandola, R.: PRIMA-UML: A performance validation incremental methodology on early UML diagrams **44**(1), 101–129 (2002)

[16] Denning, P.J., Buzen, J.P.: The operational analysis of queueing network models. ACM Comput. Surv. **10**(3), 225–261 (1978)

[17] Draheim, D., Grundy, J., Hosking, J., Lutteroth, C., Weber, G.: Realistic load testing ofweb applications. In: Proceedings of the Conference on Software Maintenance and Reengineering. pp. 57–70. CSMR '06 (2006)

[18] Ferrari, D.: On the foundations of artificial workload design. SIGMETRICS Perform. Eval. Rev. **12**(3), 8–14 (1984)

[19] Field, T., Chatley, R., Wei, D.: Software performance testing in virtual time. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering. pp. 173–174. ICPE '18 (2018)

[20] Goseva-Popstojanova, K., Singh, A.D., Mazimdar, S., Li, F.: Empirical characterization of session-based workload and reliability for web servers **11**(1), 71–117 (2006)

[21] Graf, I.M.: Transformation Between Different Levels of Workload Characterization for Capacity Planning. In: Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1987). pp. 195–204. ACM (1987)

[22] Hildebrandt, S.: Analysis 1. Springer Berlin Heidelberg, 2nd edn.

[23] Horký, V., Libič, P., Marek, L., Steinhauser, A., Tůma, P.: Utilizing performance unit tests to increase performance awareness. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering. pp. 289–300. ICPE '15 (2015)

[24] Jiang, Z.M., Hassan, A.E.: A survey on load testing of large-scale software systems. IEEE Trans. Software Eng. **41**(11), 1091–1118 (2015)

[25] Kazmi, R., Jawawi, D.N.A., Mohamad, R., Ghani, I.: Effective regression test case selection: A systematic literature review. ACM Comput. Surv. **50**(2), 29:1–29:32 (2017)

[26] v. Kistowski, J., Herbst, N., Kounev, S.: Limbo: A tool for modeling variable load intensities. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering. pp. 225–226. ICPE '14 (2014)

[27] Koziolek, H., Reussner, R.H.: A model transformation from the Palladio Component Model to layered queueing networks. In: Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW 2008). Lecture Notes in Computer Science, vol. 5119, pp. 58–78 (2008)

[28] Lyu, M.R. (ed.): Handbook of Software Reliability Engineering. McGraw-Hill, Inc. (1996)

[29] Massey Jr., F.J.: The Kolmogorov-Smirnov test for goodness of fit. Journal of the American Statistical Association **46**, 68–78

[30] Menascé, D.A., Almeida, V.A.F.: Capacity Planning for Web Services: Metrics, Models and Methods. Prentice Hall (2002)

[31] Montgomery, D.C., Runger, G.C.: Applied Statistics and Probability for Engineers. John Wiley & Sons, Inc., 3rd edn. (2003)

[32] Musa, J.D.: Operational profiles in software-reliability engineering **10**(2), 14–32 (1993)

[33] Newman, S.: Building Microservices—Designing Fine-Grained Systems. O'Reilly Media (2015)

[34] Okanović, D., van Hoorn, A., Heger, C., Wert, A., Siegl, S.: Towards performance tooling interoperability: An open format for representing execution traces. In: Proceedings of the 13th European Workshop on Computer Performance Engineering (EPEW 2016). pp. 94–108 (2016)

[35] Pietrantuono, R., Russo, S., Guerriero, A.: Run-time reliability estimation of microservice architectures. In: 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE). pp. 25–35 (2018)

[36] Ruffo, G., Schifanella, R., Sereno, M., Politi, R.: Walty: a user behavior tailored tool for evaluating web application performance. In: Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications,. pp. 77–86 (2004)

[37] Schulz, H., Angerstein, T., van Hoorn, A.: Towards automating representative load testing in continuous software engineering. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering. pp. 123–126. ICPE (2018)

[38] Schulz, H., Angerstein, T., Okanović, D., van Hoorn, A.: Replication package: Microservice-tailored generation of session-based workload models for representative load testing (2019), https://doi.org/10.5281/zenodo.3333367

[39] Shams, M., Krishnamurthy, D., Far, B.H.: A model-based approach for testing the performance of web applications. In: Proc. SOQUA 2006 (2006)

[40] Versteeg, S., Du, M., Schneider, J.G., Grundy, J., Han, J., Goyal, M.: Opaque service virtualisation: A practical tool for emulating endpoint systems. In: Proceedings of the 38th International Conference on Software Engineering (ICSE 2016). pp. 202–211 (2016)

[41] Vögele, C., van Hoorn, A., Schulz, E., Hasselbring, W., Krcmar, H.: WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction - a model-driven approach for session-based application systems. Software and System Modeling **17**(2), 443–477 (2018)

[42] Zhu, L., Bass, L., Champlin-Scharff, G.: DevOps and its practices. IEEE Software **33**(3), 32–34 (2016)