Benchmarking the Performance of Microservice Applications

Martin Grambow
TU Berlin & ECDF
Mobile Cloud Computing Research Group
Berlin, Germany
mg@mcc.tu-berlin.de

Erik Wittern
IBM
Hybrid Cloud Integration
Hamburg, Germany
erik.wittern@ibm.com

David Bermbach
TU Berlin & ECDF
Mobile Cloud Computing Research Group
Berlin, Germany
db@mcc.tu-berlin.de

ABSTRACT

Application performance is crucial for end user satisfaction. It has therefore been proposed to benchmark new software releases as part of the build process. While this can easily be done for system classes which come with a standard interface, e.g., database or messaging systems, benchmarking microservice applications is hard because each application requires its own custom benchmark and benchmark implementation due to interface heterogeneity. Furthermore, even minor interface changes will easily break an existing benchmark implementation.

In previous work, we proposed a benchmarking approach for single microservices: Assuming a REST-based microservice interface, developers describe the benchmark workload based on abstract interaction patterns. At runtime, our approach uses an interface description such as the OpenAPI specification to automatically resolve and bind the workload patterns to the concrete endpoint before executing the benchmark and collecting results. In this extended paper, we enhance our approach with the capabilities necessary for benchmarking entire microservice applications, especially the ability to resolve complex data dependencies across microservice endpoints. We evaluate our approach through our proof-of-concept prototype OpenISBT and demonstrate that it can be used to benchmark an open source microservice application with little manual effort.

CCS Concepts

•**Information systems** → **RESTful web services;** •**Software and its engineering** → **Software performance;** *Specification languages;* •**Applied computing** → *Service-oriented architectures;*

Keywords

Benchmarking; API; microservice; QoS; RESTful; Open API Specification; Swagger; SLA

1. INTRODUCTION

The complexity of today's software systems and their requirements are growing continuously. Thus, modern applications often rely on a microservice architecture to ease the development process(es), the deployment, and the operation of complex software systems with many components [27]. Instead of one large monolithic system, the business logic of an application is distributed across many small services which execute their specific tasks according to the UNIX-philosophy "Make each program do one thing well" [29].

While the functional requirements of individual microservices can be checked by specifying unit and integration tests, there are some challenges in ensuring non-functional requirements. State of the art live testing techniques coupled with monitoring include canary releases [20] or dark launches [14], which deploy a new version of the service in the production environment and assess its functionality and non-functional characteristics on a (small) share of actual traffic. However, live testing is not possible (i) when there is no production system (e.g., in early development stages), (ii) when testing for non-current workloads (e.g., testing the Christmas traffic of the shopping cart service in July), or (iii) when exposing even a fraction of actual traffic to a new, untested service is too risky. Thus, an alternative but complementary approach to live testing or monitoring is to benchmark new microservice releases. In contrast to live testing, benchmarking evaluates a microservice in a well-defined and isolated testbed which can also include related services. Benchmarking, thus, allows developers to evaluate of non-functional requirements – such as performance – of microservices in specific environments, for specific workloads, and over time. For example, repeatedly running the same benchmark in a controlled environment while the microservice evolves over time can identify performance regressions (or improvements) [18, 38, 10]. Benchmarking, however, can be costly to perform. Besides the setup procedure for the testing environment, workloads have to be defined, the benchmark run has to be monitored, and finally the results must be analyzed to decide whether the requirements have been met.

In other domains such as database benchmarking, there are a variety of tools, e.g., YCSB [9, 3, 13], which leverage the common interface of database systems to achieve repeatability and portability. For microservice applications, however, interfaces and supported operations vary with every

service, making it impossible to find a general interface for microservice benchmarking. Furthermore, as microservices evolve over time, interface changes will break ad-hoc benchmarks that a developer might have implemented.

In previous work [19], we proposed an approach for REST-based microservices in which developers can specify their benchmark workload through abstract interaction patterns. At runtime, these patterns are then automatically resolved and bound to a concrete microservice leveraging its REST characteristics. This allows developers to reuse both the abstract workload description and the benchmark execution environment [3] across microservice releases and the latter also across microservices. This significantly reduces the effort for developers while still ensuring that benchmarks fulfill important characteristics, namely that they are relevant, repeatable, portable, verifiable, and economical [7]. In our previous approach, we focused on single microservices – full microservice applications or dependencies across microservices were neither considered nor supported.

In this paper, we extend this work to microservice applications with multiple services and cross-microservice dependencies through a significantly more complex application-wide pattern matching process. As in our earlier work, we do this based on the machine-readable interface descriptions of the respective microservices and resolve the actual workload at benchmark runtime. Our extended algorithm identifies links between microservice operations and resolves the specified abstract patterns into application-specific interaction sequences which can span multiple microservices.

In this regard, we make the following contributions:

1. We propose an extended approach for benchmarking of REST-based microservice applications based on abstract and reusable interaction patterns.

2. We present a proof-of-concept prototype implementing our pattern-based benchmarking approach.

3. We evaluate our approach by benchmarking an open-source microservice application to demonstrate how little manual effort is needed for this.

Please, note that providing a complete pattern catalog is beyond the scope of this paper, but applying our approach in practice obviously requires a comprehensive set of interaction patterns.

The remainder of this paper is structured as follows: After outlining relevant background in Section 2, we present our pattern-based benchmarking approach in Section 3 and its evaluation in Section 4. We discuss our approach in Section 5 and present related work in Section 6 before concluding in Section 7.

2. BACKGROUND

This section outlines relevant designs principles and paradigms used in this paper.

2.1 Microservices

Lewis and Fowler [27] describe microservices as independently deployable and scalable components. In contrast to a monolithic system which combines all application logic in a single artifact, the microservice architecture splits the logic into a suite of services that communicate with one another over network. Within an application, individual services[1] can be written in different programming languages or use different storage technologies, resulting in a heterogeneous environment. Being separate deployment units, individual services can independently be shut down, replaced or updated at will, or new service instances can be deployed at runtime to counteract performance bottlenecks. Given these characteristics, all services must be designed to tolerate failures, as no service can expect correctly typed data or assume that a required service is always available. A challenge for microservice architectures is the lack of debugging and logging capabilities, especially in complex setups including a multitude of services.

2.2 REST APIs

Microservices communicate over the network, relying on networked application programming interfaces (APIs). APIs can differ in the communication protocols (e.g., TCP, HTTP) and data formats (e.g., JSON, binary data, XML) they rely on. In this work, we focus on APIs following the REST architectural style. Being heavily inspired by HTTP, REST APIs evolve around resources being identified by hierarchical URLs, and use HTTP methods to interact with these resources (e.g., POST to create one or GET to receive one). REST APIs do not rely on client state (stateless), and evolve around the communication of resource representations (typically in JSON or XML) between clients and servers [34].

Richardson's maturity model [16] divides REST APIs into three levels: While level 0 APIs use HTTP only to tunnel requests to an endpoint, level 1 introduces resources which can be addressed following hierarchical URIs. Level 2 additionally demands that APIs use HTTP verbs to indicate whether to create (POST), get (GET), update (PUT or PATCH), or delete (DELETE) a resource. Finally, level 3 inserts links (URIs) to corresponding services and or resources into the server responses at runtime, realizing *RESTful* APIs. In this paper, we assume APIs to comply at least with level 2 of this maturity model – specifically, we rely on the use of HTTP methods for defining abstract operations.

2.3 Interface description

In addition to human-readable API documentation targeting (client) developers, REST APIs are often described in a machine-understandable way using description files such as OpenAPI [2] or RAML [3]. For the sake of simplicity, we decided to only consider OpenAPI in our work as possible interface contract.[4] OpenAPI files are written in YAML or JSON and describe where to reach an API, available oper-

[1] In the following, we will refer to microservices as either 'services' or 'microservices' interchangeably.

[2] `https://swagger.io/docs/specification/about/`

[3] `https://github.com/raml-org/raml-spec/`

[4] Translations between formats are possible, using for example `https://apimatic.io/transformer`.

ations, its expected inputs, and possible outputs. Although the current version, OpenAPI 3.0, supports so-called link definitions to express relationships between two requests, they are not designed to describe complex interaction patterns which we develop and present in this paper.

2.4 Benchmarking

Benchmarking "is the process of measuring quality and collecting information on system states" [7] and can be applied to compare different software versions, configurations, system alternatives, or deployments. In benchmarking, a measurement client runs an application-driven workload multiple times against a system or service under test (SUT), typically in a non-production environment, and evaluates the outputs in a subsequent offline analysis to determine its quality of service (QoS) while complying with various general benchmark requirements, e.g., [21, 7]. Benchmarking requires a very high degree of control over the SUT to make results reproducible. In contrast to monitoring, which is about non-intrusive and passive observation of a (production) system, benchmarking aims to answer how a system will react on specific changes or stresses, and is about comparison of systems or deployments.

3. PATTERN-BASED BENCHMARKING

Our pattern-based benchmarking approach relies on the observation that there are sequences of interactions with resources in REST APIs which recur across APIs. One common example is to list resources of a specific type (e.g., by performing *GET .../customers*), to then retrieve information about one specific resource (e.g., by performing *GET .../customers/1*), and finally deleting that resource (e.g., by performing *DELETE .../customers/1*). Based on this observation, we argue that it is possible to automatically generate benchmarking workloads from

- an abstract description of such patterns and
- a description of how to interact with each of the microservices' APIs (e.g., in OpenAPI format).

In this section, we start by describing the challenges in generating such a pattern-based workload. Next, we introduce our pattern-based solution in detail and finally give an overview of our approach's system design.

3.1 Challenges

We have identified the following three major challenges facing our approach:

(A) The first challenge is to define patterns and workloads for arbitrary microservice applications, including the total number of requests and their distribution across patterns.

(B) Once defined, the individual patterns must be mapped to the individual services and their respective operations. Here, an abstract pattern composed of multiple operations (e.g., *listResources*) must be linked to service-specific resources (e.g., a list of *customers* or *catalog items*) and its operations (e.g., *GET ../customers* and *GET ../catalogue*).

(C) Finally, the abstract requests must be filled with concrete parameter values depending on the interface definition which is, especially for request sequences, hard because parameter values may depend on the outcome of previous requests (potentially to a different service).

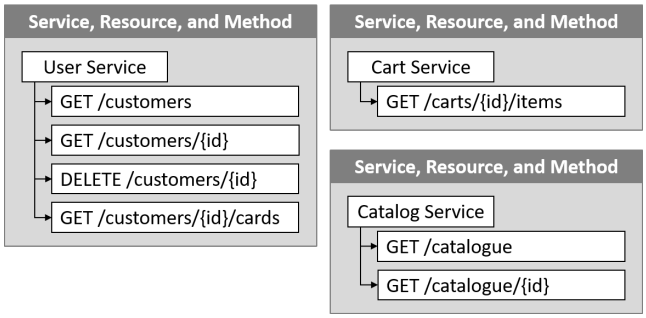

Figure 1: Example application used to explain our matching process.

3.2 From Abstract Interaction Patterns to Service-Specific Workloads

The key idea of our approach is to define an abstract workload separately from the services in an application itself and to resolve the actual service-specific workload at runtime. To address the challenges outlined above, which also have interdependencies, we divide this process into six steps (described in detail later). While challenge A is solved in the first two steps, steps 3 and 4 aim to cope with the difficulties described in Challenge B. Finally, challenge C, the actual workload generation, is covered in the steps 5 and 6.

1. **Pattern definition:** Define abstract interaction patterns.
2. **Workload definition:** Enhance pattern definition and specify frequency and ratio of requests.
3. **Binding definition:** Optionally, overwrite default binding behavior.
4. **Binding enactment:** Bind abstract interactions to concrete microservices and their operations.
5. **Workload generation:** Create application-specific workload.
6. **Benchmark execution:** Run the workload against the SUT and substitute values at runtime.

In the following, we will explain these steps using the example application listed in Figure 1. In the example, the customer and cart service use the same ID field.

Step 1 – Pattern definition: The first step is to define abstract interaction patterns that are independent of the microservices but still applicable to them.

Table 1: List of currently supported abstract operations which are combined to form abstract interaction pattern.

Operation	Description
CREATE	Creates and returns an item.
READ	Reads an item based on some filter (e.g., an ID) and returns the requested item.
SCAN	Reads multiple items based on some (optional) filter (e.g., a keyword) and returns the results.
UPDATE	Modifies an item based on some filter.
DELETE	Deletes an item based on some filter.

Table 2: Abstract interaction pattern which requests multiple resources, reads one random item from the resulting list, and finally deletes the selected item.

Step	Operation	Input	Selector	Output
1	SCAN	-	-	list
2	READ	list	RAND	item
3	DELETE	item	-	-

As defined by the second level of Richardson's maturity model, the typical REST CRUD operations can be mapped to HTTP methods: A new resource can be created at a resource endpoint by calling the *POST* HTTP method and accessed following a path structure at that endpoint. Individual resources can be read (HTTP *GET*), updated (HTTP *PATCH* or *PUT*), and deleted (HTTP *DELETE*). Finally, multiple resources can be listed by sending an HTTP *GET* to a list operation (e.g., *GET ../search*) which potentially may return multiple items. In the very first step of our approach, we use these basic interactions to define the abstract operations shown in Table 1 which we will later use to bind abstract patterns to concrete service resources.

While almost all of these interactions refer to a specific single resource, read operations can request either a single resource or multiple resources; we therefore distinguish the two read operations *READ* (single) and *SCAN* (multiple). Most operations require some filter information about the item to read, update, or delete. These do not only include an ID or key of the requested resource, but also further domain-specific values if multiple items should be read (SCAN). Furthermore, we introduce selectors as part of these filter information: If a list of items serves as input for an operation, the selector determines which item to pick from that list (e.g., first, last, or random item).

Our abstract operations already cover the common CRUD interactions with REST services. If necessary, our approach can be extended with additional basic operations. Using the basic operations from Table 1, we can now compose an interaction pattern as a sequence of abstract interactions. Thereby, each interaction is linked to a microservice, an operation, an abstract resource, and optional filter information. Moreover, it must define where to store output values of an interaction and from where to obtain input values.

Table 3: Abstract interaction pattern which queries a list of resources and then requests all associated resources for one random item.

Step	Operation	Input	Selector	Output
1	SCAN	-	-	list
2	SCAN	list	RAND	sublist

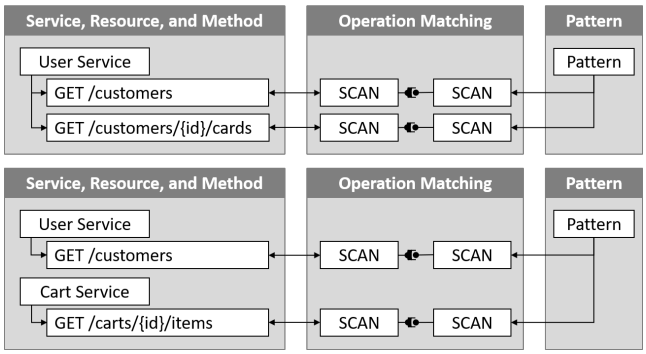

Figure 2: Matching the pattern from Table 3 to two different interaction sequences.

The complete interaction pattern for the abstract example from the beginning of this section is shown in Table 2: First, a SCAN operation determines all available resources on a service resource endpoint and stores the resulting values in a variable called *list*. Next, an individual value is selected by a random selector from this list, the corresponding resource is read (possibly from another microservice), and stored into a variable called *item*. Finally, the selected item is deleted.

Table 3 motivates a more complex example with sub-resources. The interaction starts again with a *SCAN* of available resources and the outcome is written to a variable called *list*. Applied to a microservice application, this could, e.g., request a list of users which each have a number of sub-resources. Next, a random item from that list serves as input for a subsequent *SCAN* operation which requests all (sub-)resources for the chosen item, e.g., a list of all credit cards for the selected user or a list of all items in the user's shopping cart (see Figure 1). Figure 2 shows how the binding of the pattern shown in Table 3 would later be resolved for the two example sub-resources: In the upper part, the selected customer ID is used to retrieve all credit cards of the selected user from the same microservice as sub-resources. In the lower part, the ID is used to list all items in the selected user's cart via a second microservice.

Step 2 – Workload Definition: The next step is to specify the actual workload which should be executed against the SUT. Similar to the business transactions in BenchFoundry [3], a pattern definition can include optional conditions for individual patterns (e.g., waiting times between operations to mimic realistic user behavior). Comparable to YCSB [9], which defines a workload based on a total number of operations as well as the respective share of each database operation, we define a workload based on three pieces of information: first, the list of all patterns which shall be used;

second, the total number of pattern invocations; third, the share or weight of each pattern. At execution time, multiple such patterns are usually executed in parallel.

In the following, we will refer to the abstract interaction patterns defined in Step 1 and the workload definition as *pattern configuration*.

Step 3 – Binding Definition: While our matching algorithm automatically identifies links between requests, there are several cases in which these automatic bindings should be suppressed or require additional information. In this optional adjustment step, developers can manually exclude specific service operations from the matching algorithm or define links between microservices. If used, this provides additional information to the default binding process described in Step 4 below.

As one usage scenario, microservice applications sometimes provide multiple resource endpoints (e.g., */customers* and */catalogue*) which can be used by the benchmarking client for an interaction pattern. By default, all possible resource endpoints and operations are used by the benchmark. When, however, the automatic binding from pattern to resource and operation should be suppressed (e.g., in case that only the */customers* resource endpoint should be benchmarked), this can be achieved by excluding the respective service endpoint for a subset of the patterns.

As another usage scenario, a manual definition can also be used to link parameters of two services with the goal of enabling interaction sequences which span multiple microservices of the same application. For example, if the resources of the user service can be accessed through a parameter called *id* and another service uses the same IDs to maintain the shopping carts for the respective user, then our binding algorithm needs this manual link definition to match an abstract pattern to these two services, i.e., to clarify that the second ID is indeed the user ID and not the cart ID. Here, semi-automatic approaches based on text similarity measures can support developers to define these links. For simplicity of presentation, however, we will focus only on manual links between services in this paper.

Step 4 – Binding Enactment: As already described above, our approach for automated binding enactment relies on a number of key ideas: First, REST operations can directly be mapped to the corresponding HTTP methods, e.g., a *CREATE* is mapped to an HTTP *POST*. Second, a microservice which complies with the second level of Richardson's maturity level exposes its operations in a way that is compliant with the REST operation semantics, e.g., creating a new user will always be exposed as a *CREATE* which can then be mapped to *POST*. Third, the input and output of these operations as well as the corresponding data schema are described in the interface description file, i.e., in our case, the OpenAPI file, so that we can link the output of one operation to the input of another. This allows us to create the cross-operation links in our interaction patterns. Finally, the interface description also provides information on where to find the microservice, hence, we can actually invoke it once we have completed all the mappings as described above. Based on the reasoning above, we can automatically generate a binding between an interaction pattern and the actual sequence of HTTP calls – subject to the conditions above, e.g., that creating a new user is not exposed as a *PUT*.

Algorithm 1: Generate Interaction Sequences

```
   Input: pattern - abstract interaction pattern
   Input: specs - list of interface descriptions
   Result: sequences - supported interaction
           sequences
 1
 2 /* Initialization                                  */
 3 AO = IdentifyAbstractOperations(specs)
 4 root = CreateRootNode()
 5
 6 /* Matching                                        */
 7 for i ← 0 to pattern.size do
 8 |   a = pattern[i]
 9 |   candidate = FindByAbstractOperation(a, AO)
10 |   foreach node ∈ GetNodesByLevel(i,root) do
11 |   |   foreach o ∈ candidate do
12 |   |   |   if AreDependenciesResolvable(node, o)
   |   |   |    then
13 |   |   |   |   AddChildNode(node, o)
14 |   |   |   end
15 |   |   end
16 |   end
17 end
18
19 /* Sequence Extraction                             */
20 sequences = ExtractSequences(root)
```

A pattern can be mapped to a microservice application if there is at least one supported interaction sequence for this pattern within the application. As shown in Figure 2, this can either affect a single service only or an operation result can be used to interact with another service, thus, enabling a cross-service benchmark run.

Algorithm 1 describes the algorithm we use to match and generate the application-specific interaction sequences; it has the three phases Initialization, Matching, and Sequence Extraction.

Step 4.1 – Initialization: First, we analyze the interface description of all services and determine the abstract operation for all resource paths and operations (line 3). While it is easy to determine the operation for creation, modification, and deletion as they directly map to an HTTP method by convention, this is more difficult for the SCAN and READ operation. Here, we have to inspect the resource path a bit further and check if it ends with an input parameter. If so, the parameter refers to a key or an ID and supports the READ operation; If the resource path does not end with a parameter and returns an array of items, it can be bound to the SCAN operation.

Next, we initialize a virtual root node of a tree data structure which we will use to store intermediate results of pattern matching (line 4). The paths in the tree from root to leaf will later hold our interaction sequences.

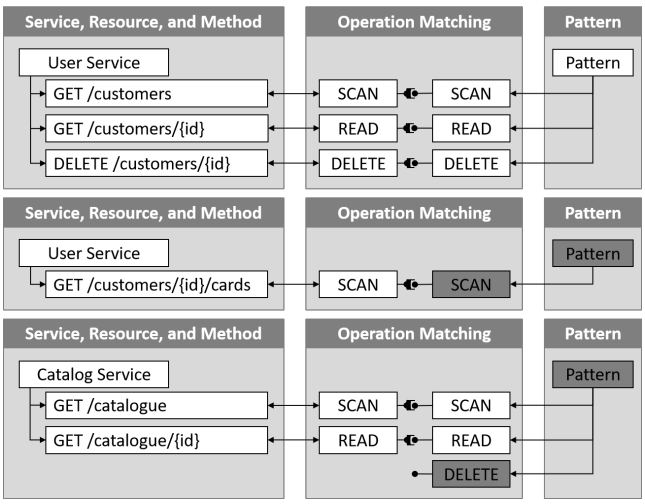

Figure 3: Mapping a pattern to interaction sequences, only one sequence (list all users, read one random user, and delete the selected user profile) supports the example pattern.

Step 4.2 – Matching: In the Matching phase, we iterate over the abstract operations in the pattern and select all service operations of the same abstract type as a list of operation candidates (lines 7-9). For the abstract interaction pattern in Table 2 and our example application, the first operation can be mapped to three service-specific operation candidates: listing all users, listing all credit cards of a user, and listing all items of the catalog service (see Figure 3).

Next, we iterate over all leaf nodes *node* at the specified level in the tree (for the first pattern operation this is the root node) and check for every candidate operation *o* whether its dependencies can be fulfilled on the path from *root* to *node* (lines 10 - 16). Candidate operations that do not require input values (such as listing all users) have no dependencies and can, hence, always be used. In that case, we simply add this operation as a child node to *node*. If, however, there are dependencies, we verify that the required information can be obtained from previous requests on the path from *root* to *node*. In our example, this affects the sequence starting with listing the cards of a user: As it is a candidate for the first operation, there is no information available on the required user ID, see Figure 3. Any operation for which all dependencies can be resolved is added as a child node; all other operations are disregarded (lines 12 - 14). In this step, it is also possible to define additional conditions that an operation needs to fulfill before it can be added as a child node; in our prototype, for instance, we have implemented a filter that only adds operations that target microservices already used on the current path or microservices for which an explicit link has been defined in step 3. We do this to limit the number of interaction sequences, but it is not strictly necessary to do so.

We execute this for all operations in the pattern and, thus, gradually build our tree data structure. In the third example in Figure 3 for instance, we can see that the first two operations of the example pattern can be matched against the catalog service. The third operation (DELETE), however, cannot be matched since the only available service-specific DELETE requires a user ID as input which cannot be obtained from the two previous operations (listing all catalog entries and reading a specific catalog item).

Step 4.3 – Extract Sequences: Finally, we extract the interaction sequences from our tree. For this, every path from the root node to a leaf that – excluding the root node – contains the same number of nodes as the pattern has operations represents an interaction sequence. Shorter paths are incomplete interaction sequences where one or more pattern operations could not be matched; these can be discarded.

When at least one interaction sequence per pattern has been found, the binding is saved and represents the automatic binding for the combination of patterns, binding definition, and interface descriptions. In some cases, the binding algorithm may find sequences that are not relevant or not desired in the respective use case. These sequences can either be deleted or deactivated manually or can be used to create additional load on the SUT.

Step 5 – Workload Generation: With the previously created binding and the interface descriptions, we can finally generate the benchmark workload by building HTTP requests which follow the interaction patterns and the restrictions in the interface description files. First, each pattern operation can be directly resolved to an HTTP method based on the binding. Next, we can fill the required parameters and request body content of each request by inspecting the interface description of the respective microservice and generating random values for all interactions: In OpenAPI, complex parameter values and request bodies are described using JSON Schema.[5] We can use these descriptions to generate the required data items filled with random values. Moreover, we can generate use-case specific values such as product names or Bitcoin addresses by augmenting the service description with special keywords.

As stated above, some content of the individual requests may depend on the returned values of previous calls (e.g., identifier values). These values must be injected later in the benchmark execution phase (Step 6) for which we use special markers. Nevertheless, once the required number of pattern executions has been generated, the workload can be persisted and reused across several benchmark runs even if the generated workload is incomplete in that sense.

Step 6 – Benchmark Execution: As already stated above, some values of the workload must be replaced during the execution if there are dependencies between requests. For these values, there are many sources depending on the concrete operation: A *CREATE* operation could, for example, return the ID of the created item or respond with an HTTP 200 status code if the ID was part of the request and the item was created successfully. Depending on the implementation, the subsequent *READ* request must select the ID from the response or the request body of the preceding request; this, however, only if the response had an HTTP 200 status.

[5] `https://json-schema.org/`

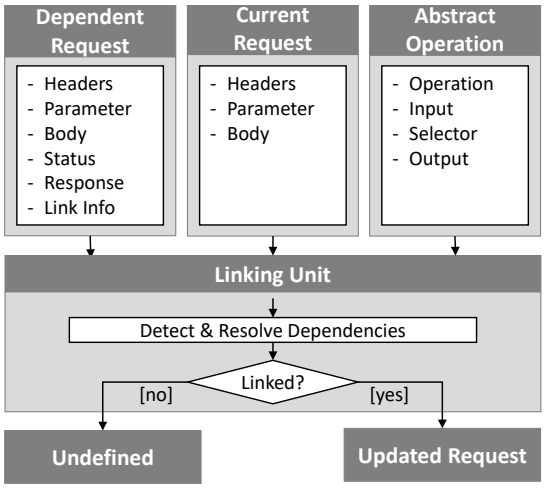

Figure 4: Linking two related requests based on the content. If a link is detected, the successive request is updated and returned.

For such purposes, we have designed the *linking unit* interface illustrated in Figure 4: A linking unit tries to find and resolve dependencies based on the preceding requests (including message body, response, etc.), the current request (including the values to be replaced, e.g., a parameter), and the abstract pattern operation. If a linking unit detects a link, it resolves the dependency, e.g., by replacing the placeholder value in the current request body with some value from the parameters of the previous one, and returns the updated request or *"undefined"* otherwise. This way, it is possible to apply multiple linking units sequentially until a dependency has been resolved; it also allows us to order the application of several units hierarchically (e.g., general or very service-specific units first).

Currently, we have identified four different types of linking units but additional, potentially service-specific, custom units can be added:

- **OpenAPI Link Linking Unit:** OpenAPI 3.0 documents can define links which describe further operations and their content after a query. This linking unit inspects the link definitions of the preceding request and replaces the values of the current request accordingly.
- **Binding Linking Unit:** This unit resolves the links manually defined in the binding definition (see step 3).
- **Parameter Name Linking Unit:** Individual resources can often be accessed by following a path structure in REST interfaces, e.g., /{*username*}. These parameters were initially filled with placeholder values in Step 5 which have to be adjusted now to access actual resources. This linking unit searches for these values in the preceding request based on the parameter name. If exactly one element with this name as key is found in the request (either in parameter values, request body, or response), this value is used in the current request. If there are multiple values to choose from, which one is chosen is determined by the selector (e.g., select a random value).
- **ID Linking Unit:** In some cases, parameter names in the preceding and current requests do not match exactly. For example, a user is created with a field named "*id*" in the request body and individual users can be accessed via the path /{*userID*}. This linking unit resolves dependencies by searching field names for the substring "id" and replacing values in the same fashion as the parameter name linking unit.
- **Custom Linking Unit:** Finally, as dependencies cannot always be detected and resolved with our default linking units, developers can also define custom and service-specific linking units which can be added to the application chain of units by implementing the linking unit interface.

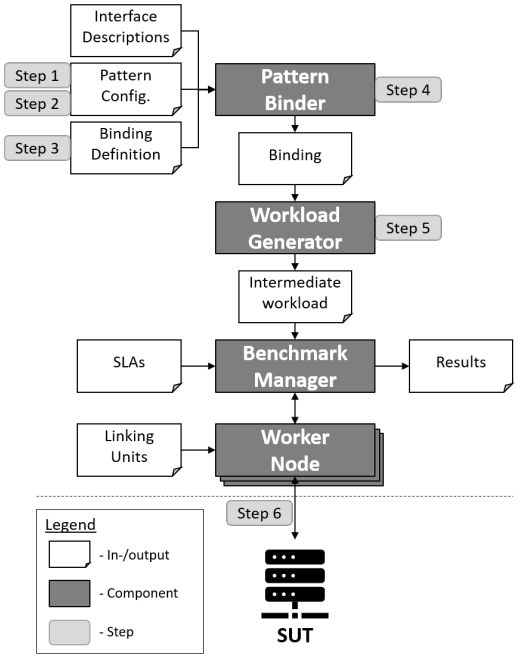

Figure 5: Overview of our system design and setup including input and output documents.

3.3 System Design

Our system design comprises a number of components; these – along with the corresponding steps – are shown in Figure 5. The Pattern Binder creates service-specific interaction sequences based on API description files, a pattern configuration, and optional binding definitions (steps 1 to 3). Once the Pattern Binder has bound every pattern to at least one interaction sequence (Step 4), the binding can be stored and, if necessary, adjusted manually (e.g., if the automatic algorithm identified an unusual but possible interaction sequence). Next, the Workload Generator generates the service-specific but incomplete workload based on this pattern binding (Step 5). As a pattern execution is by definition independent of other pattern executions (otherwise,

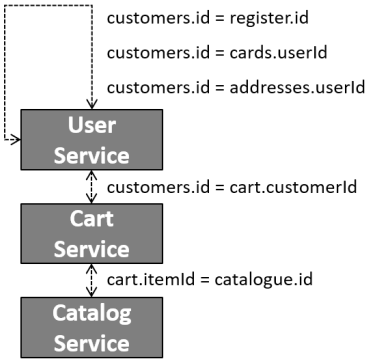

Figure 6: Our evaluation focuses on three microservices of the Sock Shop application and their interdependencies.

they should be merged into one pattern), we can parallelize pattern execution and also distribute this execution across a number of Worker Nodes. Similar to the method proposed in [3], the Benchmark Manager does this by partitioning the workload into worker packages to enable concurrent execution (the number of packages corresponds to the number of concurrent Worker Nodes), then distributes the worker packages to the available Worker Nodes, manages the (concurrent) benchmark execution, and collects the results. Finally, as our approach is intended for use in Continuous Integration and Deployment pipelines [18, 38], the Benchmark Manager compares the observed metrics to predefined requirements and constraints such as service level agreements (SLAs) to ultimately decide on success or failure of the benchmark run.

Within a worker package, requests across patterns can be interleaved as long as requests within a pattern are not reordered. As some requests depend on the outcome of preceding ones (e.g., the UPDATE operation requires the resource ID which was part of the result from a previous CREATE call), the Worker Nodes cannot simply read the generated workload and run the requests against a service endpoint, but must adapt some values at runtime with the outcome from posted requests based on the linking units (Step 6).

4. EVALUATION

In our original paper [19], we evaluated our benchmarking approach through a proof-of-concept prototype and a set of experiments with three different microservices to demonstrate the general applicability for benchmarking individual services. In this paper, we omit these single service experiments and focus on evaluating our extended approach by benchmarking an entire microservice-based application with multiple services using our improved proof-of-concept prototype. As, again, our focus is the applicability of our approach, the actual measurement *results* are irrelevant as long as results *can* be obtained. In this section, we first present our proof-of-concept implementation and describe the microservice application which we benchmarked. Next, we describe how we followed the individual steps of our approach to create and run a pattern-based benchmark workload. Finally, we summarize our evaluation findings.

4.1 Proof-of-concept Implementation

We implemented our approach and system design as an open-source proof-of-concept prototype[6] written in Kotlin. Our prototype implementation can be integrated in an existing Continuous Integration and Deployment pipeline and comprises four components:

1. A Pattern Binder which maps abstract interaction patterns to service-specific operations.

2. A Workload Generator which fills the requests with random values based on the interface description.

3. A Benchmark Manager which orchestrates the benchmark run and aggregates the results.

4. Worker Nodes which execute the workload, resolve dependencies between successive requests using linking units, and measure the runtime of each operation to report the results.

Our prototype uses the open-source library json-schema-faker[7] to generate data for the workload. This library allows us to generate the required JSON elements for the individual HTTP requests based on the schema information in the OpenAPI description file. Moreover, our prototype also supports special faker keywords (defined by the library Faker.js[8]) which can be added to the OpenAPI file. These additional keywords can be used to generate realistic and use-case-specific workload values (e.g., names, product IDs, or dates).

4.2 Sock Shop Microservice Application

We evaluate our approach on a microservice-based Webshop[9] for socks which implements an e-commerce application. In our experiments, we focus on the following three REST services and dependencies (see Figure 6):

User Service[10]: The user service maintains the customer information such as usernames, passwords, credit cards, and addresses. Each user has a unique customer ID which is used to access the respective data set as REST resource. Moreover, there are also resource paths for credit cards and addresses. For example, a new address can be created by sending an HTTP POST to */addresses* and all addresses of a customer can be queried by sending an HTTP GET to */customers/{id}/addresses*.

Cart Service[11]: Each user has a virtual shopping cart which is as REST resource identified by their customer ID. Items from the catalog service can be added, deleted, or modified.

[6] https://github.com/martingrambow/openISBT
[7] https://github.com/json-schema-faker/json-schema-faker/
[8] https://github.com/marak/Faker.js/
[9] https://microservices-demo.github.io/
[10] https://github.com/microservices-demo/user
[11] https://github.com/microservices-demo/carts

Table 4: An overview of the four interaction patterns which we use in our experiments.

Pattern	Step	Operation	Input	Selector	Output
LST	1	SCAN	-	-	list
	2	READ	list	RAND	-
DEL	1	SCAN	-	-	list
	2	READ	list	RAND	item
	3	DELETE	item	-	-
SUBLST	1	SCAN	-	-	list
	3	SCAN	list	RAND	sublist
TWOIN	1	SCAN	-	-	list1
	2	SCAN	-	-	list2
	3	CREATE	list1, list2	RAND, RAND	item

Catalog Service[12]**:** The catalog service provides an interface for retrieving all products available in the shop. A data item comprises an item ID (which is also used for accessing the respective REST resource), a description, tags, and the price of the corresponding product. The service only offers operations for browsing existing products, items can neither be modified nor added or removed.

4.3 Experiment

In line with our proposed process, we run the following experiments to evaluate our pattern-based benchmarking approach:

Step 1 – Pattern Definition: We evaluate the REST microservices discussed above with four self-defined abstract interaction patterns as shown in Table 4: First, a simple list pattern which lists available resources and reads an item from that list (LST). Second, our introductory example pattern from Table 2 which identifies and deletes a resource (DEL). Third, our more complex example pattern from Table 3 which lists sub-resources for a randomly selected root resource (SUBLST). Fourth, another complex pattern which creates a new resource based on two input values (TWOIN).

In all steps where an item needs to be selected from a list, we always use a random selector for reasons of simplicity but there will of course be services for which another selector makes more sense, e.g., picking the latest item. Moreover, we want to emphasize again that these patterns are examples only, as our goal is not to identify a comprehensive pattern catalog.

Step 2 – Workload Definition: In this evaluation, we want to demonstrate the general functionality and applicability of our approach and not to rate the performance of individual microservices. Thus, we run rather "small" workloads and use one benchmark run only. In practice, however, these parameters must be adapted to fulfill usual benchmark best practices, e.g., regarding execution duration [7]. For our experiment, we run 1,000 pattern requests in total, 250 per pattern. We also run an initial preload phase which inserts

[12] `https://github.com/microservices-demo/catalogue`

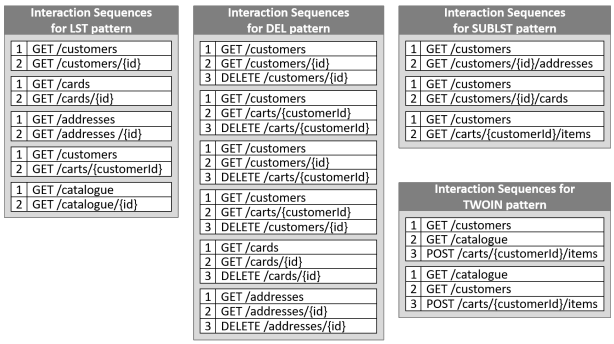

Figure 7: All our example patterns can be matched to at least one interaction sequence each.

1,000 customers including one credit card, one address, and one cart item in advance for each user. The catalog service already offers nine items out of the box.

Step 3 – Binding Definition: For our experiment, we do not have to manually exclude operations (this may be different in other scenarios). We, however, define five manual links for the ID fields of the evaluated microservices as shown in Figure 6.

The customer ID, which is the key for accessing the REST resource, is used in four different contexts under different names: Within the user service, the field *id* is used for the */customers* and */register* paths. Moreover, the same value is referenced as *userId* in the */cards* and */addresses* paths. Besides these links, we define two links which connect the user service to the cart service via the customer ID and the cart service to the catalog service via the item ID.

Step 4 – Binding Enactment: Here, we bind our example patterns to the target microservices and their resources. Figure 7 outlines the resulting binding for each pattern, including our manual definitions from Step 3.

For the *LST* pattern, our binding algorithm identifies five possible interaction sequences. Three of them are limited to the user service (e.g., query a list of customers and get the details of a random customer afterwards), one interacts with the catalog service (get all items and select a random one), and one sequence combines the user and cart service (query a list of customers and get the virtual cart of one random customer from the resulting list).

The *DEL* pattern can be found in six interactions. The first four interactions start with listing registered customers. Next, one randomly selected customer ID can be used to either get the details for this customer or to get the customer's shopping cart. Third, the customer ID serves as input to delete either the customer or the cart. The remaining two sequences start with listing all credit cards or addresses. In the first binding iteration, the search is restricted to the respective endpoint and the */customer* endpoint as no other link has been defined (we discussed this additional condition in Step 4 of Section 3). Thus, the *id* fields are not mixed up and the *READ* operation continues to use either the */cards* or */addresses* endpoint. The */customer* endpoint is disre-

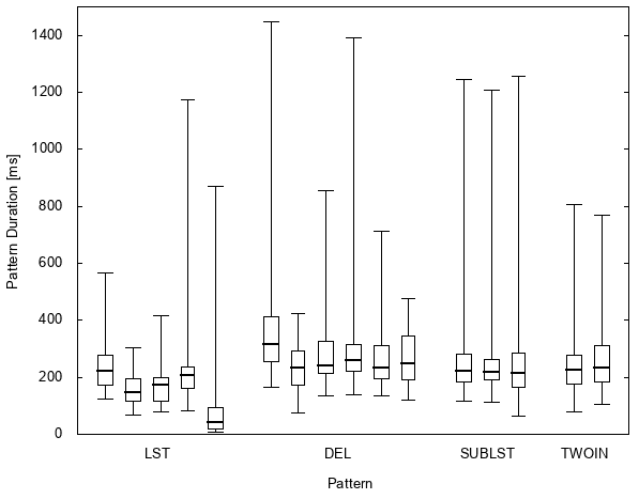

Figure 8: Example results for total sequence duration with n=250 measurements per pattern. The box plots use the same order as the interaction sequences in Figure 7.

garded because the *customer.id* is not linked to the *cards.id* (but to *cards.userId*). The same holds for the second binding iteration and the final *DELETE* operation.

Our binding for the *SUBLST* pattern identifies three interaction sequences, all starting with listing customers. Next, the randomly selected customer ID can be used to list the user's addresses, credit cards, or shopping cart items.

The *TWOIN* pattern combines two different inputs to create a new resource. In our evaluated application, this pattern can be used to add items from the catalog to a user's shopping cart. For the first two operations, it does not matter whether customers or items are listed first as neither operation has a dependency on the respective other one. Thus, our binding algorithm identifies two interaction sequences, one starting with listing customers, the other starting with listing catalog items. Finally, the *CREATE* operation requires one input as parameter and another input inside the request body. Here, other operations (such as registering a new customer) are not matched because they either expect only one input value or the manually defined service links do not match (e.g., the *card.id* is not linked to the *cart.customerId*).

Step 5 – Workload Generation: To generate the workload for our experiments with our prototype, we adjust the OpenAPI files slightly for the following reasons: First, we convert[13] the service descriptions to the current OpenAPI version 3.0 for implementation reasons. Second, we align the description with the actual implementation, as the OpenAPI file is missing a few properties offered by the corresponding implementation. Finally, we add faker.js keywords to generate realistic random values, e.g., for names, numbers, and dates.

[13]`https://mermade.org.uk/openapi-converter`

Step 6 – Benchmark Execution: We run our benchmark on AWS EC2[14] instances, all in the same availability zone. For our experiment, one instance runs the Sock Shop microservice application including the three evaluated microservices and two instances each run a Worker Node with five threads each to demonstrate the parallelization and scalability features of our prototype. Finally, a fourth instance hosts the Pattern Binder, Workload Generator, and Benchmark Manager.

Our experiment benchmarks three different REST microservices of an example application. As a result, we get pattern execution measurements which include the total pattern duration and the duration of individual requests. These measurements can, e.g., be used to generate box plots for each pattern and interaction sequence. Figure 8 shows the latency at pattern level for our evaluated patterns using box plots[15]. Again, since the actual results are irrelevant and only show the applicability of our approach, we do not discuss the measured results.

4.4 Summary

As described above, our prototype can benchmark typical microservice-based applications with minimal adaptation effort. After initializing all application services, our proof-of-concept implementation benchmarks the evaluated and linked microservices (almost) out of the box and only a few minor changes in the description file are needed. Moreover, the abstract workload definition – which, in our evaluation example, is a JSON file with less than 100 lines – can be reused across a variety of services in different versions. The five manual links between the microservices which we had to define are about 30 lines. Unless major breaking interface changes are made, these manual links can be reused across benchmark runs against different versions of the microservice application. Overall, we managed to design and setup the benchmark for the entire microservice applications in less than an hour in total which significantly reduces the effort necessary to benchmark a microservice: there is simply no need anymore to manually implement a benchmark (tool) from scratch which can also be quite challenging [4].

5. DISCUSSION

As the evaluation shows, our approach can be used to benchmark REST microservices based on their service description but, nevertheless, there are some points to consider when applying our pattern-based approach and which we want to discuss in more detail here. Moreover, we also present current limitations and propose possible solutions for them.

Our design generates a synthetic and traced-based workload based on a pattern and workload definition. Defining a proper workload comes with its own challenges (which, however, is not specific to this approach). If a workload definition creates more resources than deleting existing ones, the list of resources grows with every iteration and may produce unrealistic workloads. E.g, the workload definition from our evaluation may fit the carts service because the number of

[14]`https://aws.amazon.com/ec2/`

[15]Boxes represent 25%, 50%, and 75% quartiles; whiskers show min and max duration for each sequence

items is usually constant (about the same number of additions and deletions) but, on the other hand, it may not fit the customer service well, where the number of users usually increases during the service lifetime because there are more new users registering than existing ones leaving. Thus, the actual patterns and a realistic distribution of these patterns has to be considered when defining the workload (e.g., by inspecting the log files to identify common interactions and their frequency [22]). This also implies that an existing workload based on a common pattern catalog yet to be defined cannot be applied blindly to other microservices.

Next, our approach relies on the semantics of REST-based interfaces and assumes HTTP-based microservices. Microservices using other communication protocols can be used as well but essentially require manual bindings for every operation. Since the basic CRUD semantics exist independent of the protocol used, it will be interesting to see whether there are common ways in which these are exposed in non-REST-based microservices and whether these could be leveraged by our approach. E.g., the abstract operations could be mapped via the function name instead of the HTTP verb. Nonetheless, our approach can already be used for a large variety of microservices for which there are no benchmarking alternatives yet.

While not every pattern and workload definition can be blindly applied to every REST microservice, our approach allows developers to run a benchmark against REST services which share the same characteristics in general, which is helpful in several situations: First, every new service version can be compared to older versions as long as the individual changes do not alter the basic characteristics of the microservice. This is particularly important for use in Continuous Integration pipelines [18, 38, 10]. Second, if the API changes (e.g., a new parameter is introduced or a schema is adjusted), nothing but the interface description file must be replaced (ideally, this description is generated from the microservices' source code with every build) and the benchmark adapts automatically, there is no need to adjust workload files or source code. Third, our approach can be used to evaluate different services which share the same purpose (e.g., user management). This is particularly useful when replacing a microservice with a new one as both can be benchmarked and compared extensively prior to switching in production.

In contrast to our initial prototype which was limited to single microservices and did not consider cross-service dependencies and sub-resources, our extended approach addresses this issue. There are, however, still some limitations: First, the dependencies between microservices must be defined manually in advance. This requires domain knowledge about the target application and can be hard for applications with many services. Here, our prototype could be enhanced with an automated analysis based on text similarities which detects the service dependencies automatically to support application developers, e.g., that identifies that `userID` and `user/{id}` refer to the same property. Furthermore, our binding algorithm identifies all possible interaction sequences for a pattern configuration (and an optional binding definition). This should be handled with caution as the binding might also identify unrealistic sequences which should be disabled before running the benchmark. E.g., two of the *DEL* interaction sequences which we found in our evaluation might be unrealistic (in another scenario). Here, the third operation deletes a shopping cart after getting the details for a selected customer or deletes a customer after querying its cart. Thus, we recommend verifying the identified bindings manually before running benchmark experiments, especially for applications which involve multiple microservices. Nevertheless, in the worst case, such unrealistic matchings simply add additional load on the SUT – it is only important to disregards their respective results.

Overall, we believe that our approach and its prototypical implementation are useful for a large percentage of microservice deployments as they significantly reduce the implementation effort for microservice developers. Some restrictions such as the REST requirement apply but could also serve as an incentive to switch to REST in some cases where other communication solutions are used for legacy reasons.

6. RELATED WORK

Benchmarking is a well-established method in the IT domain to quantify and verify quality of service of hardware or software systems [7]. There are many benchmarks for different kinds of SUT, especially for database and storage systems, e.g., [32, 13, 3, 30, 25], but also for virtual machines, e.g., [8, 36], web APIs [5, 6], or cloud-based queuing systems, e.g., [26]. To the best of our knowledge, however, there is currently no approach (or even a tool) for benchmarking microservices. We believe that this is largely due to the fact that microservices do not come with the common interface typical to other system domains such as POSIX for virtual machines or SQL and CRUD interfaces for data management. Without such a common interface, it becomes quite hard to implement a benchmark that complies with standard benchmark requirements – especially portability [21, 15, 7, 4, 37]. Recent work in the microservice benchmarking domain presents general benchmark requirements for microservices [1], focuses on the automation of performance tests of microservices [11], or implements a benchmark suite with six different microservice applications [17].

Nevertheless, there are some approaches and tools which could ease microservice benchmarking beyond building a complete benchmark from scratch: Load generators such as Artillery IO[16] or LoadUI[17] can run a defined and service-specific workload against a microservice. By manually defining scenarios which represent typical interactions, a service-specific workload can be created with parameters settings which include the amount of request or the distribution of scenarios. While it is possible to import service description files and external data items as "workload", this is always specific to a particular microservice and its respective version, i.e., there is no portability. Kao et al. [23] also generate requests based on (regularly updated) service descriptions, but focus on web services and manually define each test specification. With our approach, on the other side, arbitrary REST microservices can be benchmarked as long as the ser-

[16] `https://artillery.io/`

[17] `https://www.soapui.org/professional/loadui-pro.html`

vice supports the respective interaction patterns. Pattern definitions can be reused for benchmarking other microservices. Atlidakis et al. [2] analyze OpenAPI specifications and automatically generate functional tests to detect bugs and security vulnerabilities. Our approach, on the other side, focuses on non-functional requirements. Nevertheless, their algorithm to resolve dependencies by analyzing actual service responses can be used to identify dependencies between services automatically, which can ease our binding definition step.

Zheng et al. [39] also use interaction patterns comprised of basic operations (create, get, delete) for benchmarking but do so for object storage services. Their approach relies on the standard interface defined by CDMI and, hence, does not have to deal with interface heterogeneity. Beyond these, there are several systems which could be (mis)used as a load generator. Benchmarking systems such as YCSB [9] or ND-Bench [31] can be used to create synthetic workloads against a CRUD endpoint. While these tools are very powerful load generators – particularly when considering the broad range of configuration options – they completely disregard the mapping from the generic CRUD to a specific microservice. Although creating such a mapping will be possible for a large percentage of microservices, actually programming the mapping still remains a manual effort that needs to be repeated for every microservice and version that shall be benchmarked. Furthermore, we believe that benchmarking interaction with microservices should preferably be based on sequences of operations instead of isolated operations to get more realistic results (systems such as YCSB+T [12] or BenchFoundry [3] are probably a better fit).

Besides workload generation and invocation of REST endpoints, our approach generates synthetic data for the workload. For data generation, we rely on JSON schema and the faker.js library discussed above. Approaches such as [33] are more powerful options for data generation and also support parallel generation. Such parallelization could improve our prototype in which generating the workload trace prior to distributing it onto the Worker Nodes can be rather slow. Nevertheless, we do not see parallelization as a critical feature since the generated workload can be persisted and reused instead of being generated from scratch for every benchmark run.

Aside from benchmarking, alternatives such as canary releases and blue-green testing coupled with monitoring [35] can be used if the option exists to expose the new microservice (version) to a share of the production traffic.

Finally, an alternative to both benchmarking and live testing – at least for clients of a microservice – can be to rely on SLAs while monitoring violations, e.g., [24, 28]. This approach, however, only shifts the responsibility for ensuring microservice performance to another organizational entity and does not actually solve the challenge of detecting performance changes of microservices early on, ideally as part of a Continuous Integration pipeline [18, 38, 10].

7. CONCLUSION

Benchmarking microservices serves to understand and check their non-functional properties for relevant workloads and over time. Performing benchmarks, however, can be costly: each microservice requires the design and implementation of a benchmark from scratch, possibly repeatedly as the service evolves. As microservice APIs differ widely, benchmarking tools, which typically assume common interfaces of the system under test, do not exist yet.

In this work, we proposed a pattern-based approach to reduce the efforts for defining microservice benchmarks, while still being able to measure qualities of complex interactions. Our approach assumes that microservices expose a REST API, described in a machine-understandable way, and allows developers to model interaction patterns from abstract operations that can be mapped to that API. Required parameter values are provided at runtime and possible data-dependencies between operations are resolved. We implemented our approach in a prototype, which we used to demonstrate the low effort applicability of our pattern-based benchmarking approach to an open-source microservice-based application with multiple services and dependencies. With this, we could show that pattern-based benchmarking of microservices is indeed feasible which opens up opportunities for microservice providers and tooling developers.

8. ACKNOWLEDGMENTS

We would like to thank Lukas Meusel who contributed to the implementation of a first proof-of-concept prototype for our original paper [19]. The contributions of Erik Wittern were mostly done while at IBM T. J. Watson Research Center, Yorktown Heights, NY, USA.

9. REFERENCES

[1] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi. Benchmark Requirements for Microservices Architecture Research. In *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*. IEEE, 2017.

[2] V. Atlidakis, P. Godefroid, and M. Polishchuk. REST-ler: automatic intelligent REST API Fuzzing. 2018.

[3] D. Bermbach, J. Kuhlenkamp, A. Dey, A. Ramachandran, A. Fekete, and S. Tai. BenchFoundry: A Benchmarking Framework for Cloud Storage Services. In *Proc. of the International Conference on Service Oriented Computing (ICSOC 2017)*. Springer, 2017.

[4] D. Bermbach, J. Kuhlenkamp, A. Dey, S. Sakr, and R. Nambiar. Towards an Extensible Middleware for Database Benchmarking. In *Proc. of the TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2014)*. Springer, 2014.

[5] D. Bermbach and E. Wittern. Benchmarking web api quality. In *Proc. of the International Conference on Web Engineering (ICWE 2016)*. Springer, 2016.

[6] D. Bermbach and E. Wittern. Benchmarking web api quality – revisited. *Journal of Web Engineering*, 2020.

[7] D. Bermbach, E. Wittern, and S. Tai. *Cloud Service Benchmarking: Measuring Quality of Cloud Services*

from a Client Perspective. Springer, 2017.

[8] A. H. Borhani, P. Leitner, B.-S. Lee, X. Li, and T. Hung. Wpress: An application-driven performance benchmark for cloud-based virtual machines. In *Proc. of the International Enterprise Distributed Object Computing Conference (EDOC 2014)*. IEEE, 2014.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of the Symposium on Cloud Computing (SOCC 2010)*. ACM, 2010.

[10] D. Daly, W. Brown, H. Ingo, J. O'Leary, and D. Bradford. The Use of Change Point Detection to Identify Software Performance Regressions in a Continuous Integration System. In *Proc. of the ACM/SPEC International Conference on Performance Engineering (ICPE 2020)*, 2020.

[11] A. De Camargo, I. Salvadori, R. d. S. Mello, and F. Siqueira. An Architecture to Automate Performance Tests on Microservices. In *Proc. of the 18th International Conference on Information Integration and Web-based Applications and Services*. ACM, 2016.

[12] A. Dey, A. Fekete, R. Nambiar, and U. Röhm. Ycsb+ t: Benchmarking web-scale transactional databases. In *Proc. of the International Conference on Data Engineering Workshops (ICDE 2014)*. IEEE, 2014.

[13] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4), 2013.

[14] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and deployment at facebook. *IEEE Internet Computing*, 17(4), 2013.

[15] E. Folkerts, A. Alexandrov, K. Sachs, A. Iosup, V. Markl, and C. Tosun. Benchmarking in the cloud: What it should, can, and cannot be. In *Proc. of the TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2012)*. Springer, 2013.

[16] M. Fowler. Richardson maturity model, 2010.

[17] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, and others. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proc. of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM, 2019.

[18] M. Grambow, F. Lehmann, and D. Bermbach. Continuous benchmarking: Using system benchmarking in build pipelines. In *Proc. of the Workshop on Service Quality and Quantitative Evaluation in new Emerging Technologies (SQUEET 2019)*. IEEE, 2019.

[19] M. Grambow, L. Meusel, E. Wittern, and D. Bermbach. Benchmarking Microservice Performance: A Pattern-based Approach. In *Proc. of the 35th ACM Symposium on Applied Computing (SAC 2020)*. ACM, 2020.

[20] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[21] K. Huppler. The art of building a good benchmark. In *Proc. of the TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2009)*. Springer, 2009.

[22] A. Ivanchikj, I. Gjorgjiev, and C. Pautasso. Restalk miner: Mining restful conversations, pattern discovery and matching. In *Proc. of International Conference on Service-Oriented Computing - Workshops (ICSOC 2018)*. Springer, 2018.

[23] C. H. Kao, C. C. Lin, and J.-N. Chen. Performance Testing Framework for REST-based Web Applications. In *13th International Conference on Quality Software*. IEEE, 2013.

[24] A. Keller and H. Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11, 2003.

[25] M. Klems, D. Bermbach, and R. Weinert. A runtime quality measurement framework for cloud database service systems. In *Proc. of the International Conference on the Quality of Information and Communications Technology (QUATIC 2012)*. IEEE, Sept 2012.

[26] M. Klems, M. Menzel, and R. Fischer. Consistency benchmarking: Evaluating the consistency behavior of middleware services in the cloud. In P. Maglio, M. Weske, J. Yang, and M. Fantinato, editors, *Service-Oriented Computing*, volume 6470 of *Lecture Notes in Computer Science*. Springer, 2010.

[27] J. Lewis and M. Fowler. Microservices, 2014.

[28] H. Ludwig, A. Keller, A. Dan, R. P. King, and R. Franck. Web service level agreement (wsla) language specification. *Ibm corporation*, 2003.

[29] M. D. McIlroy, E. Pinson, and B. Tague. Unix time-sharing system: Foreword. *Bell System Technical Journal*, 57(6), 1978.

[30] S. Müller, D. Bermbach, S. Tai, and F. Pallas. Benchmarking the performance impact of transport layer security in cloud database systems. In *Proc. of the International Conference on Cloud Engineering (IC2E 2014)*. IEEE, 2014.

[31] I. Papapanagiotou and V. Chella. Ndbench: Benchmarking microservices at scale. *arXiv e-prints*, 2018.

[32] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. Ycsb++: Benchmarking and performance debugging advanced features in scalable table stores. In *Proc. of the Symposium on Cloud Computing (SOCC 2011)*. ACM, 2011.

[33] T. Rabl, M. Frank, H. M. Sergieh, and H. Kosch. A data generator for cloud-scale benchmarking. In *Proc. of the TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2010)*. Springer, 2010.

[34] A. Rodriguez. Restful web services: The basics. *IBM developerWorks*, 33, 2008.

[35] G. Schermann, D. Schöni, P. Leitner, and H. C. Gall. Bifrost: supporting continuous deployment with

automated enactment of multi-phase live testing strategies. In *Proc. of the International Middleware Conference (Middleware 2016)*. ACM, 2016.

[36] J. Scheuner, P. Leitner, J. Cito, and H. Gall. Cloud workbench – infrastructure-as-code based cloud benchmarking. In *Proc. of the International Conference on Cloud Computing Technology and Science (CloudCom 2014)*. IEEE, January 2014.

[37] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao. How to build a benchmark. In *Proc. of the ACM/SPEC International Conference on Performance Engineering (ICPE 2015)*. ACM, 2015.

[38] J. Waller, N. C. Ehmke, and W. Hasselbring. Including performance benchmarks into continuous integration to enable devops. *ACM SIGSOFT Software Engineering Notes*, 40(2), 2015.

[39] Q. Zheng, H. Chen, Y. Wang, J. Duan, and Z. Huang. Cosbench: A benchmark tool for cloud object storage services. In *Proc. of the International Conference on Cloud Computing (CLOUD 2012)*. IEEE, 2012.