Benchmarking Microservice Performance: A Pattern-based Approach

Martin Grambow
TU Berlin & Einstein Center Digital Future
Mobile Cloud Computing Research Group
mg@mcc.tu-berlin.de

Lukas Meusel
TU Berlin & Einstein Center Digital Future
Mobile Cloud Computing Research Group
lms@mcc.tu-berlin.de

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

David Bermbach
TU Berlin & Einstein Center Digital Future
Mobile Cloud Computing Research Group
db@mcc.tu-berlin.de

ABSTRACT

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, possibly repeatedly as the service evolves. As microservice APIs differ, benchmarking tools that assume common interfaces – like ones for databases – do not exist.

In this work, we present a pattern-based approach to reduce the efforts for defining microservice benchmarks, while still allowing to measure qualities of complex interactions. It 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. Possible data-dependencies between operations are resolved at runtime. We implement a prototype of our approach, which we use to demonstrate that it can be applied to open-source microservices with little effort. Our work shows that pattern-based benchmarking of microservices is feasible and opens up opportunities for microservice providers and tooling developers.

CCS CONCEPTS

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

ACM Reference Format:
Martin Grambow, Lukas Meusel, Erik Wittern, and David Bermbach. 2020. Benchmarking Microservice Performance:, A Pattern-based Approach. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3341105.3373875

*Work done while at IBM T. J. Watson Research Center, Yorktown Heights, NY, USA

1 INTRODUCTION

The complexity of today's software artifacts and its requirements are steadily growing. Thus, modern applications often rely on a microservice architecture to ease the development process(es), the deployment, and the operation of a complex software system with many components [20]. 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" [22].

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 [14] or dark launches [9], 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 when there is no production system (e.g., in early development stages), when testing for non-current workloads (e.g., testing the Christmas traffic of the shopping cart service in July), or 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 the new version of a microservice. In contrast to live testing, benchmarking evaluates a microservice in a well-defined and isolated testbed which can also include related services. Benchmarking allows the evaluation of non-functional requirements – such as performance – of microservices in specific environments, for specific workloads, and over time. For example, running the same benchmark in a controlled environment as the microservice evolves over time can point to performance regressions (or improvements). 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 were met.

In other domains such as database benchmarking, there are a variety of tools, e.g., YCSB [1, 6, 8], which make use of the common interfaces of database systems to achieve automated, repeatable, and comparable benchmarks even as part of Continuous Integration and Deployment pipelines [13, 31]. For microservices, however, interfaces and the typical requests they expect vary with every service, making it impossible to find a general interface for microservice benchmarking.

In this paper, rather than relying on a standardized interface, we rely on the Representational State Transfer (REST) architectural style commonly used by microservices for pattern-based benchmarking of microservices. In our approach, users define abstract and reusable interaction patterns which are resolved to the actual workload at runtime automatically. Introducing this level of abstraction eliminates the need to implement or adjust benchmarks for every change in the service interface (e.g., if the parameter set has changed). Our approach thus reduces the effort for developers while still ensuring that benchmarks fulfill important characteristics, namely that they are relevant, repeatable, portable, verifiable, and economical [4]. In this regard, we make the following contributions:

(1) An approach to benchmark REST microservices based on abstract and reusable interaction patterns.
(2) An open source proof-of-concept prototype implementing our pattern-based benchmarking approach.
(3) The evaluation of our approach by benchmarking three open-source REST microservices.

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 [20] 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 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 [27].

Richardson's maturity model [11] 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 [1] or RAML [2]. For the sake of simplicity, we decided to only consider OpenAPI in our work as possible interface contract.[3] OpenAPI files are written in YAML or JSON and describe where to reach an API, available operations, and 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" [4] 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., [4, 15]. 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

[1]https://swagger.io/docs/specification/about/
[2]https://github.com/raml-org/raml-spec/
[3]Translations between formats are possible, using for example https://apimatic.io/transformer.

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 (e.g., an ID) and returns it.
DELETE	Deletes an item based on some filter (e.g., an ID).

Table 1: Extensible list of abstract operations which are combined to abstract interaction pattern.

APIs which recur across APIs. One common example is to list resources of a specific type (e.g., by performing *GET .../users*), to then retrieve information about one specific resource (e.g., by performing *GET .../user/1*), and finally updating that resource (e.g., by performing *PUT .../users/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 the microservice's API (e.g., OpenAPI).

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 services, including the total number of requests and their distribution across patterns.
(B) Once defined, the individual patterns must be mapped to the service URI path and method. Here, an abstract pattern composed of multiple operations (e.g., *listResources*) must be linked to service-specific resources (e.g., a list of *users* or *products*) and its operations (e.g., *GET ../users* and *GET ../products*).
(C) Finally, the abstract requests must be filled with concrete parameter values depending on the interface definition which is, especially for successive requests, hard because parameter values may depend on the outcome of previous requests.

3.2 From Abstract Pattern Definition to Service-Specific Workloads

The key idea of our approach is to define an abstract workload separately from the service 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.

Step	Operation	Input	Selector	Output
1	SCAN	-	-	list
2	READ	list	RANDOM	item
3	UPDATE	item	-	-

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

(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 resource bindings.
(4) **Binding enactment:** Bind pattern configuration to resource paths and service operations.
(5) **Workload generation:** Create service-specific workload.
(6) **Benchmark execution:** Run the workload against the SUT and substitute values at runtime.

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

Following 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, reading can request both, a single resource and multiple resources; we therefore split reading into *READ* (single) and *SCAN* (multiple). Most operations require some filter information about the item to read, to update, or to 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 further basic operations. Using the available basic operations, we can now compose an interaction pattern as a sequence of abstract interactions. Thereby, each interaction is linked to an operation, an abstract resource, and optional filter information. Moreover, it must define where to put output values of an interaction and from where to read input values.

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 picked by a *RANDOM* selector from this list, the corresponding resource is read, and stored into a variable called *item*. Finally, the selected item is updated.

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 [1], a pattern definition can include optional conditions for individual patterns (e.g., waiting times between operations to mimic realistic user behavior). Comparable to YCSB [6], 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 (likely to be) executed in parallel.

Step 3 – Binding Definition: This optional binding definition step can be used to manually bind patterns to service resource(s). If used, this information overrides the default binding from Step 4 described below.

As one usage scenario, microservices sometimes provide multiple resource endpoints (e.g., */users* and */orders*) 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 or overridden (e.g., in case that only the */users* resource endpoint should be benchmarked), this can be achieved through a manual binding.

As another usage scenario, a manual definition can also be applied in cases in which single operations differ from the actual resource path. For example, if a service offers the endpoints */users* and */register*, then the latter one must be manually bound to the abstract *CREATE* operation of the */user* endpoint to make the pattern execute.

In the following, we will refer to the abstract interaction patterns defined in Step 1, the workload definition from Step 2, and the optional binding definition as *pattern configuration*.

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.

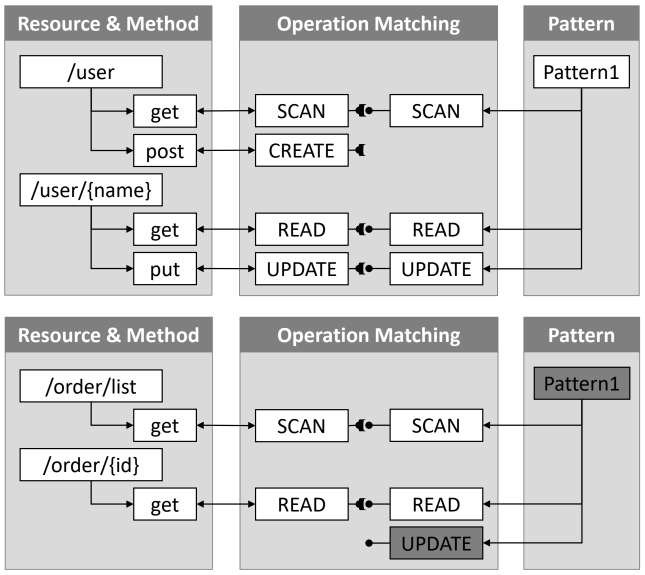

Figure 1: Matching two resource paths to an abstract interaction pattern, only one path (*/user*) is supported.

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*. In the following sub-steps, we describe how we verify that an automated binding can be created[4]:

Step 4.1 – Matching: A pattern can be mapped to a resource path if this path supports all required operations of the pattern. Figure 1 illustrates this for an example with two resource paths and the abstract pattern given in Table 2: While the */user* path supports all abstract operations of the pattern, the */order* path does not support the UPDATE operation. Thus, the pattern is only mapped to the /user resource path. 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.
If all operations in an abstract interaction pattern are supported by a resource path, we bind them (if nothing to the contrary is demanded in the manual binding definition) and create a map from interaction pattern to a list of supported resource paths and their available methods.

Step 4.2 – Resolving Dependencies: Next, we verify that all interactions can be theoretically carried out based on the service description. In the example from above, a specific

[4]If there are bindings defined in Step 3 which already explicitly map a pattern to specific resource paths and operations or exclude some, this information overrides the following default mapping. In that case, the approach only verifies that it can indeed resolve the manually specified binding.

user with a user id (identified by that id in the resource path) can only be requested if that user id is part of an earlier service request or response. Thus, we iterate over the pattern and try to resolve all dependencies by matching the described input and output of previous abstract operations to the required input of the current one by its name (the name defined in the abstract interaction pattern). If there are unresolved dependencies for a previously mapped pair of abstract pattern and resource path, we remove that path from the corresponding list because the pattern cannot be executed. For example, if the output of the first operation shown in Table 2 would be named "itemlist" and the input for second operation would still be "list", the dependency could not be detected and resolved.

Step 4.3 – Ensuring Executability: Finally, having concluded the mapping and dependency resolution for every interaction pattern, we check if there is at least one resource path left which supports the pattern configuration and all of its abstract interaction patterns. Otherwise, the pattern configuration cannot be executed and we have to cancel the benchmark process as there is no resource path to interact with. Typically, users would then return to Step 3 to define manual bindings or return to Step 1 to modify the pattern definition.

Step 5 – Workload Generation: With the previously created mapping and the interface description, we can finally generate the benchmark workload by building HTTP requests which follow the interaction patterns and the restrictions in the interface description. 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 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 different 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 pick the

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

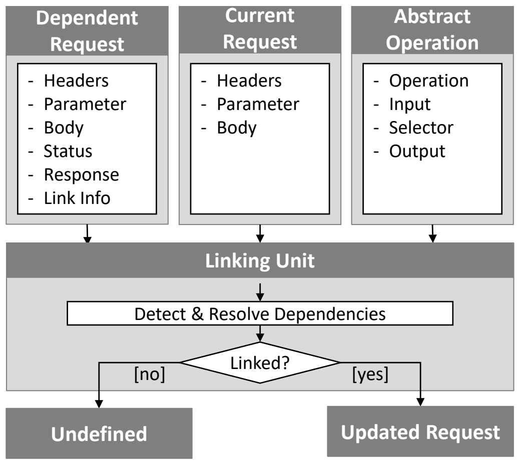

Figure 2: Linking two related requests based on the content. If a link is detected, the successive request is updated and returned.

id from the response or the request body of the preceding request; this, however, only if the response was an HTTP 200 status.

For purposes such as these, we have designed the *linking unit* interface, illustrated in Figure 2: A linking unit tries to find and resolve dependencies based on the preceding request (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 successively until a dependency is resolved and it also offers the possibility to order the application of several units hierarchically (e.g., general or very service-specific units first).

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

- **OpenAPI Link Linking Unit:** OpenAPI 3.0 offers the possibility to 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.
- **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 actually existing 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, one is picked according to the selector (e.g., pick a random value).

- **ID Linking Unit:** In some cases, the 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 defined linking units, there is also the possibility to define custom and service-specific units which can be added to the application chain of units by implementing the linking unit interface.

3.3 System Design

Our system design comprises a number of components; these – along with the corresponding steps – are shown in Figure 3. The Workload Generator creates a service-specific workload based on an API description file, a pattern configuration, and optional binding definitions (steps 1 to 3). Once the Workload Generator has bound the interaction patterns to supported resource endpoints (Step 4), it generates the service-specific but incomplete workload (Step 5). As a pattern execution is by definition independent of other pattern executions (otherwise, they should be merged), we can parallelize pattern execution and also distribute this execution across a number of Worker Nodes. Similar to the method proposed in [1], 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 among 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 [13, 31], 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 method 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

To evaluate our pattern-based approach, we implemented a proof-of-concept prototype and benchmarked three different open-source REST microservices. Our goal is to assess the applicability of our approach to different services, while the results of the benchmark runs are secondary. In this section, we first present our proof-of-concept implementation and describe the microservices which we benchmarked. Next, we describe how we followed the individual steps of our approach to create and run a pattern-based benchmark workload for each service and briefly outline the measured results. Finally, we summarize our evaluation findings.

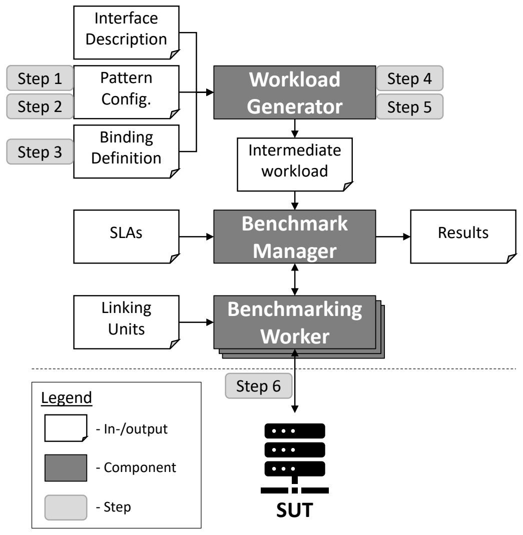

Figure 3: Overview of our system design and setup including input and output documents.

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 or Deployment pipeline and comprises three components:

(1) A graphical web frontend to interact with the Workload Generator and Benchmark Manager.
(2) A server backend that includes the Workload Generator and the Benchmark Manager.
(3) A Worker Node which runs the workload, resolves dependencies between successive requests using linking units, and measures 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. The 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 Services

As we want to show the general applicability of our proposed approach in realistic scenarios, we decided to benchmark the following three different open-source REST microservices:

[6]https://github.com/martingrambow/openISBT
[7]https://github.com/json-schema-faker/json-schema-faker/
[8]https://github.com/marak/Faker.js/

Pattern	Step	Operation	Input	Selector	Output
CRE	1	CREATE	-	-	item
	2	GET	item	-	-
LST	1	SCAN	-	-	list
	2	READ	list	RANDOM	item
UPD	1	SCAN	-	-	list
	2	READ	list	RANDOM	item
	3	UPDATE	item	-	-
DEL	1	SCAN	-	-	list
	2	READ	list	RANDOM	item
	3	DELETE	item	-	-

Table 3: List of evaluated patterns which were ran against the select microservices.

Petstore: The petstore[9] is a microservice maintaining pets and users. It is a simple service which is often used in tutorials to show how the interface description language OpenAPI works and covers almost all aspects of it.

Teams: The Flask-RESTplus Example API[10] is a popular (more than 800 stars on GitHub) REST microservice which organizes users into teams.

Sock Shop: This microservice-based Webshop[11] simulates all parts of an e-commerce application including orders, payments, and users. Nevertheless, for our evaluation we only used the resource paths */customers* and */register* of the user service[12].

4.3 Experiment

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

Step 1 – Pattern Definition: We evaluated the outlined REST microservices with four self-defined abstract interaction patterns as shown in Table 3: First, a creation pattern (CRE) which creates a resource and verifies that it can be accessed. Second, a list pattern (LST) which lists available resources and reads an item from that list. Third, an update pattern (UPD) which identifies and then updates a resource and fourth, a deletion pattern (DEL) which picks and deletes one.

In steps which require to pick an item from a list, we always used a random selector for simplification, but there might be services for which another selector makes more sense, e.g., picking the oldest item. Moreover, we want to emphasize again that these patterns are examples only, as our goal was not to identify a comprehensive pattern catalog.

Step 2 – Workload Definition: In this evaluation, we aim to show the general functionality and applicability of our approach and not to rate the performance of the microservices in detail. Thus, we decided to run rather small workloads and to use one benchmark run only. In practice, however, these parameters must be adapted to fulfill the benchmark requirements, e.g., regarding runtime. For our experiment we ran 1,000 pattern requests in total and distributed them among the individual patterns equally. Furthermore, we ran an initial preload phase which inserts 1,000 data items in advance for each microservice.

[9]https://github.com/OpenAPITools/openapi-petstore
[10]https://github.com/frol/flask-restplus-server-example
[11]https://microservices-demo.github.io/
[12]https://github.com/microservices-demo/user

Service Resource	Pattern CRE	LST	UPD	DEL
Petstore				
/pet	✓	✓	✓	✓
/user	✓	-	-	-
/store/inventory	-	-	-	-
/store/order	✓	-	-	-
Teams				
/teams	✓	✓	✓	✓
/users	✓	✓	✓	-
Sock Shop				
/customers	✓	✓	-	✓
/cards	(-)	(-)	-	(-)
/addresses	(-)	(-)	-	(-)

Table 4: Matching of evaluated service resources and patterns. Resource paths are either supported [✓], not supported [-], or not yet supported [(-)].

Step 3 – Binding Definition: The */customers* endpoint of the Sock-Shop's user microservice does not offer a method to create new customers. Instead, new customers can be added via the */register* path. For this reason, we inserted an additional manual binding definition and overrode the automatic binding behavior to support the CRE pattern for this endpoint. Besides this custom binding for the Sock Shop service, we did not define any further manual bindings.

Step 4 – Binding Enactment: Here, we bind our defined patterns to the evaluated microservices and its resources. Table 4 outlines the resulting binding for each service with every interaction pattern, including our manual binding definition from Step 3.

As the abstract *SCAN* operation is only supported by the */pet* resource path in the Petstore service but part of the *UPD*, *LST*, and *DEL* pattern, all other resource paths cannot be benchmarked with the given workload definition. Thus, the */pet* path is the only one for the Petstore microservice.

Although the Teams service manages users and teams, it does not provide the ability to delete users. Therefore, only one resource path (*/teams*) is tested for this service as well.

The Sock Shop service does not offer an update operation at all but all other operations to support the CRE, LST, and DEL pattern for all resource paths. Thus, we decided to adjust the workload definition slightly for this service and only execute these three supported patterns. Moreover, the resource endpoints */cards* and */addresses* require an existing user ID which need to be provided in requests. Currently, our implementation does not have a linking unit for "out of pattern data" (which would be statically defined

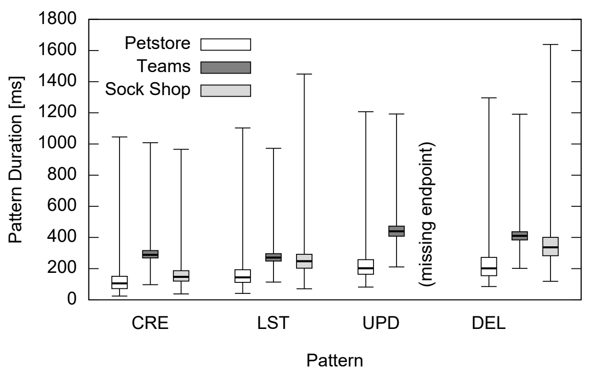

Figure 4: Duration of pattern execution with n=1,000 measurement per service and pattern.

prior to the benchmark run) so that we only benchmarked the */customers* endpoint.

Step 5 – Workload Generation: To generate the workload for our experiments with our prototype, we had to adjust the OpenAPI files slightly for the following reasons. First, our prototype currently only supports JSON data so that we removed some parts, e.g., XML-related definitions. Furthermore, we had to add some definitions to the description if the evaluated service requires an API key or OAuth authorization; e.g., we added an enum with only one element (the API key) to the corresponding scheme or defined three different tags for pets in the Petstore microservice. Finally, we also had to convert[13] the service descriptions of the Teams and Sock Shop microservice to the current OpenAPI version 3.0 for implementation reasons.

Step 6 – Benchmark Execution: We ran our benchmarks on AWS EC2[14] instances, all in the same availability zone. For each experiment, one SUT instance was running the examined microservice and two instances were running a Worker Node with five threads each to ensure that the benchmarking client is not the bottleneck and that the SUT experiences a certain load. Since the measurement results themselves are irrelevant, we particularly did this to showcase the scalability of our approach based on the Worker Nodes. Moreover, a fourth instance hosted the Benchmark Manager.

Our experiment benchmarked three different REST microservices with almost identical pattern configurations (as explained above, we excluded the UPD pattern for the Sock Shop service). All three services were running in Docker containers and were set up in advance. Even though the services are too different to compare and the usage of Docker might introduce an additional bias [12], Figure 4 gives an overview of the benchmark results as determined by our prototype (latency at pattern level). While actual results are irrelevant, this shows the applicability of our approach and hints that the performance of different microservices varies.

[13]https://mermade.org.uk/openapi-converter
[14]aws.amazon.com/ec2/

4.4 Summary

As described above, our prototype could benchmark three popular microservice examples with minimal adaptation effort. After initializing all services, our proof-of-concept implementation benchmarked the microservices (almost) out of the box and only a few changes in the description file were necessary (e.g., we had to replace the service URL or introduce an enum for the API key). Moreover, we had to define an abstract workload which is, in our evaluation example, a JSON file with less than 100 lines (which however would be reusable across a variety of services in different versions) and we had to define one manual binding for the Sock-Shop service which comprises about 20 lines. All in all, we could design and setup the benchmark for all evaluated microservices 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 [2].

Furthermore, our evaluation only considered the performance (pattern duration) of the examined microservice. In practice, however, other aspects such as consistency or scalability can also be evaluated.

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 teams service because the number of teams is usually constant (about the same number of additions and deletions) but, on the other hand, it may not fit the user 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 [16]). 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 [13, 31]. 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 microservice' 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.

Beyond considering the approach itself, there are also some limitations of our prototype: Some operations require values from other resource endpoints or even other services. Our current prototype can partially solve this issue by defining custom binding definitions but this is not sufficient to cover all use cases. For example, the evaluated Sock Shop service offers the */cards* endpoint but all offered methods require an existing userID which is not retrieved or created as part of the respective pattern. Thus, a new user must first be created or at least an existing user must be retrieved from the */customers* endpoint and this is currently not covered in our prototype. To solve this issue, we plan to make use of OpenAPI's link definitions[15] or to implement a resource pool which shares these resource identifiers across Worker Nodes. Based on link definitions, we could issue additional preparing operations before the start of a pattern execution. Moreover, our current version also do not support sub-resources or considers pagination. Currently, only the first results of a *SCAN* request are evaluated, but users can apply suitable filters to limit pagination issues. For sub-resources, we plan to extend the pattern definition to also include operations on sub-resources and interactions with them. Finally, our prototype is currently limited to benchmarking single services; patterns that invoke multiple microservices can easily be used with our approach but have not been added to our prototype yet.

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 [4]. There is a large number of benchmarks for different kinds of SUT, especially for database and storage systems, e.g., [1, 8, 18, 23, 25], but also for virtual machines, e.g., [5, 29], web APIs [3], or cloud-based queueing systems, e.g., [19]. 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 [2, 4, 10, 15, 30].

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. With our approach, on the other side, arbitrary REST microservices can be benchmarked as long as the service supports the respective interaction patterns.

Zheng et al. [32] 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 [6] or NDBench [24] 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. As such, systems like YCSB+T [7] or BenchFoundry [1] 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 [26] 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.

[15]https://swagger.io/docs/specification/links/

[16]https://artillery.io/

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

Aside from benchmarking, alternatives such as canary releases and blue green testing coupled with monitoring [28] 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., [17, 21]. 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 an Continuous Integration pipeline [13, 31].

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 three open-source microservices. With this, we could show that pattern-based benchmarking of microservices is indeed feasible which opens up opportunities for microservice providers and tooling developers.

REFERENCES

[1] David Bermbach, Jörn Kuhlenkamp, Akon Dey, Arunmoezhi Ramachandran, Alan Fekete, and Stefan Tai. 2017. BenchFoundry: A Benchmarking Framework for Cloud Storage Services. In *Proc. of the International Conference on Service Oriented Computing (ICSOC 2017)*. Springer.

[2] David Bermbach, Jörn Kuhlenkamp, Akon Dey, Sherif Sakr, and Raghunath Nambiar. 2014. Towards an Extensible Middleware for Database Benchmarking. In *Proc. of the TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2014)*. Springer.

[3] David Bermbach and Erik Wittern. 2016. Benchmarking Web API Quality. In *Proc. of the International Conference on Web Engineering (ICWE 2016)*. Springer.

[4] David Bermbach, Erik Wittern, and Stefan Tai. 2017. *Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective.* Springer.

[5] Amir Hossein Borhani, Philipp Leitner, Bu-Sung Lee, Xiaorong Li, and Terence Hung. 2014. WPress: An Application-Driven Performance Benchmark for Cloud-Based Virtual Machines. In *Proc. of the International Enterprise Distributed Object Computing Conference (EDOC 2014)*. IEEE.

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of the Symposium on Cloud Computing (SOCC 2010)*. ACM.

[7] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. 2014. YCSB+ T: Benchmarking web-scale transactional databases. In *Proc. of the International Conference on Data Engineering Workshops (ICDE 2014)*. IEEE.

[8] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7, 4 (2013).

[9] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. 2013. Development and deployment at facebook. *IEEE Internet Computing* 17, 4 (2013).

[10] Enno Folkerts, Alexander Alexandrov, Kai Sachs, Alexandru Iosup, Volker Markl, and Cafer Tosun. 2013. 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.

[11] Martin Fowler. 2010. Richardson Maturity Model. Retrieved February 18, 2019 from https://martinfowler.com/articles/richardsonMaturityModel.html

[12] Martin Grambow, Jonathan Hasenburg, Tobias Pfandzelter, and David Bermbach. 2019. Is it Safe to Dockerize my Database Benchmark?. In *Proc. of the ACM Symposium on Applied Computing, Posters Track (SAC 2019)*. ACM.

[13] Martin Grambow, Fabian Lehmann, and David Bermbach. 2019. 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.

[14] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Pearson Education.

[15] Karl Huppler. 2009. The Art of Building a Good Benchmark. In *Proc. of the TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2009)*. Springer.

[16] Ana Ivanchikj, Ilija Gjorgjiev, and Cesare Pautasso. 2018. RESTalk Miner: Mining RESTful Conversations, Pattern Discovery and Matching. In *Proc. of International Conference on Service-Oriented Computing - Workshops (ICSOC 2018)*. Springer.

[17] Alexander Keller and Heiko Ludwig. 2003. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management* 11 (2003).

[18] Markus Klems, David Bermbach, and Rene Weinert. 2012. 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.

[19] Markus Klems, Michael Menzel, and Robin Fischer. 2010. Consistency Benchmarking: Evaluating the Consistency Behavior of Middleware Services in the Cloud. In *Service-Oriented Computing*, Paul Maglio, Mathias Weske, Jian Yang, and Marcelo Fantinato (Eds.). Lecture Notes in Computer Science, Vol. 6470. Springer.

[20] James Lewis and Martin Fowler. 2014. Microservices. Retrieved February 15, 2019 from https://martinfowler.com/articles/microservices.html

[21] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P King, and Richard Franck. 2003. Web service level agreement (WSLA) language specification. *Ibm corporation* (2003).

[22] Malcolm D McIlroy, EN Pinson, and BA Tague. 1978. UNIX Time-Sharing System: Foreword. *Bell System Technical Journal* 57, 6 (1978).

[23] Steffen Müller, David Bermbach, Stefan Tai, and Frank Pallas. 2014. Benchmarking the Performance Impact of Transport Layer Security in Cloud Database Systems. In *Proc. of the International Conference on Cloud Engineering (IC2E 2014)*. IEEE.

[24] Ioannis Papapanagiotou and Vinay Chella. 2018. NDBench: Benchmarking Microservices at Scale. *arXiv e-prints* (2018).

[25] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. 2011. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *Proc. of the Symposium on Cloud Computing (SOCC 2011)*. ACM.

[26] Tilmann Rabl, Michael Frank, Hatem Mousselly Sergieh, and Harald Kosch. 2010. A data generator for cloud-scale benchmarking. In *Proc. of the TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2010)*. Springer.

[27] Alex Rodriguez. 2008. Restful web services: The basics. *IBM developerWorks* 33 (2008).

[28] Gerald Schermann, Dominik Schöni, Philipp Leitner, and Harald C Gall. 2016. Bifrost: supporting continuous deployment with automated enactment of multi-phase live testing strategies. In *Proc. of the International Middleware Conference (Middleware 2016)*. ACM.

[29] Joel Scheuner, Philipp Leitner, Jürgen Cito, and Harald Gall. 2014. Cloud WorkBench – Infrastructure-as-Code Based Cloud Benchmarking. In *Proc. of the International Conference on Cloud Computing Technology and Science (CloudCom 2014)*. IEEE.

[30] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. 2015. How to Build a Benchmark. In *Proc. of the ACM/SPEC International Conference on Performance Engineering (ICPE 2015)*. ACM.

[31] Jan Waller, Nils C Ehmke, and Wilhelm Hasselbring. 2015. Including performance benchmarks into continuous integration to enable DevOps. *ACM SIGSOFT Software Engineering Notes* 40, 2 (2015).

[32] Qing Zheng, Haopeng Chen, Yaguang Wang, Jiangang Duan, and Zhiteng Huang. 2012. Cosbench: A benchmark tool for cloud object storage services. In *Proc. of the International Conference on Cloud Computing (CLOUD 2012)*. IEEE.