Reducing the maintenance effort for parameterization of representative load tests using annotations

Henning Schulz\(^1\)*,†, André van Hoorn\(^2\) and Alexander Wert\(^1\)

\(^1\)Digital Experience and Data Intelligence, Novatec Consulting GmbH, Karlsruhe, Germany
\(^2\)Institute of Software Technology, Reliable Software Systems, University of Stuttgart, Stuttgart, Germany

SUMMARY

Directly affecting the user experience, performance is a crucial aspect of today’s software applications. Representative load testing allows to effectively test and preserve the performance before delivery by mimicking the actually expected workload. In the literature, various approaches have been proposed for extracting representative load tests from recorded user sessions. However, these approaches require manual parameterization for specifying input data and adjusting static properties such as a request’s domain name. This manual effort accumulates when load tests need to be updated due to changing production workloads and APIs.

In this paper, we address the reduction of the maintenance effort for representative load testing. We introduce input data and properties annotations (IDPAs) that store manual parameterizations and can be evolved automatically. Experts only have to parameterize extracted load tests initially. For dealing with API changes, we develop approaches to evolve IDPAs for the types of changes described in the literature. We evaluated our approach in two experimental studies, by deriving effort estimation models, and in an industrial case study including four different software projects. Our evaluation shows that IDPAs can parameterize generated load tests for restoring the representativeness, especially for applications with workloads dominated by request orders and rates. The maintenance effort can be reduced from a quadratic cumulative effort over time to a linear cumulative effort for a typical mix of API changes. Furthermore, we were able to express all parameterizations required by the industrial projects using the IDPA but also had to integrate extensions using the provided extension mechanisms. © 2019 The Authors. Software Testing, Verification & Reliability

Published by John Wiley & Sons Ltd.

1. INTRODUCTION

Proper parameterization of representative load tests is inevitable for reasonably load testing an application. Load testing is an effective approach to test the performance by simulating synthetic workloads and to preserve performance problems and other load-related issues before delivery [1]. Representative load testing is especially reasonable because a synthetic workload that represents the actual workload in a production system is used. For easing definition of representative workload specifications (a.k.a. workload models), several approaches extracting the specifications from recorded user sessions or requests have been proposed [2–6]. However, such workload models need to be parameterized to be properly executable as a load test in the test environment. For instance, different input data for the user requests fitting to the test environment’s databases are to be used and data dependencies between requests have to be modelled. In particular, a high maintenance effort arises because of changing production workloads and application programming interfaces (APIs) [7].

*Correspondence to: Henning Schulz, Digital Experience & Data Intelligence, Novatec Consulting GmbH, Karlsruhe, Germany.
†E-mail: henning.schulz@novatec-gmbh.de

[The copyright line for this article was changed on 21 February 2020 after original online publication]

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2019 The Authors. Software Testing, Verification & Reliability

Published by John Wiley & Sons Ltd.
Despite the maintenance effort, representative load testing is a promising quality assurance technique, especially in the context of continuous integration and delivery (CI/CD) pipelines [8]. Because representative load testing focuses on the actually expected workload, it prevents testing unlikely occurring workloads and, thus, unnecessary test overhead. Therefore, it suits the principle of short-running CI/CD pipelines. However, manual parameterization and maintenance effort contradict the automation in CI/CD.

Existing approaches model parameterizations including input data as part of the workload models [2–6]. However, this entails high manual effort for evolution over workload and API changes. Therefore, other approaches address the high maintenance effort by validating the representativeness of the workload models repeatedly [7, 9], in order to determine whether a regeneration is worthwhile. Commercial tools are able to identify and resolve many inter-request dependencies automatically [10, 11].

However, these approaches are not feasible in the context of constantly evolving production workloads and APIs and short-running, frequently executed, and automated CI/CD pipelines. As illustrated in Figure 1, workload model extraction as intended by our general vision to representative load testing [12] mainly consists of the three steps of selecting an appropriate workload represented by production user sessions or requests, transforming it to a load test, and executing the load test. The second step typically requires manual intervention for parameterizing the load test [6]. Besides adjusting static properties like the host name or port number that will be different in the load test, input data need to be specified, for example, for using valid user names for the login request. Dynamically selecting one or several new workloads or adopting to a changed API in a new pipeline execution requires extracting new load tests and applying the manual parameterizations again. The mentioned commercial tools support users in this process but cannot evolve the former parameterizations, either. Verifying the representativeness of a workload model repeatedly and updating the test only if required have been a suitable method but become infeasible in CI/CD pipelines.

Therefore, an approach to representative load testing is required that entails little maintenance effort and fits CI/CD pipelines. Manual intervention might still be necessary but has to be reduced and necessarily decoupled from test generation and execution time. When executing a representative load test as part of a CI/CD pipeline, the test has to be generated, parameterized, and executed automatically. Otherwise, the pipeline depends on non-trivial human intervention, which contradicts its principle [8].

In this paper, we address the reduction of the maintenance effort for representative load testing in the context of evolving workloads and APIs. Precisely, we investigate the evolution of parameterizations consisting of input data specifications and static property adjustments over workload and API changes. We need to ensure that our approach will not impair the representativeness and investigate whether it is suitable for restoring the representativeness of generated load tests. Furthermore, even though API changes can require manual refinement, the effort can be significantly reduced by evolving former parameterizations. Finally, we investigate the expressiveness of our approach regarding the requirements of real-world load tests. Summing up, we investigate the following research questions.

RQ1 How much does the parameterization by our approach impair the representativeness of a load test?

Figure 1. Representative load testing as part of a CI/CD pipeline based on Schulz et al. [12].
RQ2 To which degree do evolved parameterizations improve the representativeness of a generated load test?

RQ3 To which degree can we reduce the maintenance effort for the evolution of manual parameterizations of generated load tests over changes in the target application’s API?

RQ4 How expressive is our approach compared to parameterizations of load tests used in industrial projects?

The contributions of this paper are the following. (i) We introduce the input data and properties annotation (IDPA) that externalizes the manual parameterizations from the generated load tests into a separate model. IDPAs can be used for automatically generating new load tests for changed workloads. A demonstration is provided online [13]. (ii) We review existing literature about API changes and define a terminology of API changes with respect to the IDPA. (iii) We present an approach to semi-automatically evolving IDPAs over API changes. (iv) We evaluate our approach in four different studies. A replication package is available online [14]. First, we perform an experimental study with the widely used build artifact management system Sonatype Nexus [15] and the open-source load driver JMeter [16]. Second, we evaluate our approach in combination with the WESSBAS workload model generator [6] and JMeter using the Broadleaf Heat Clinic [17]. For discussing the reduction of the maintenance effort, we derive effort estimation models. Finally, we evaluate the expressiveness of the IDPA by modeling the parameterizations of the load tests of four different industrial software development projects in IDPAs.

Our evaluation motivates that valuable load testing requires proper input data specification. Simply replaying non-parameterized extracted load tests resulted in significantly different performance of the Broadleaf Heat Clinic. Using an IDPA, we were able to specify the input data correctly without a decrease of the representativeness of the generated load test. The study with Sonatype Nexus indicates that the use of IDPAs is especially meaningful for systems with workloads that are dominated by the order and rate of requests, such as session-based web applications like the Heat Clinic. For Sonatype Nexus, we were able to parameterize load tests while mostly preserving the representativeness but introduced a small error, because the system’s behaviour also depended on the input data. Our derived effort estimation models show that without changes in the target application’s API, the maintenance effort is limited to specifying the IDPA initially. For a typical mix of API changes as identified in (ii), a quadratic cumulative maintenance effort over time can be reduced to a linear cumulative effort. Finally, we were able to model the parameterizations required by the industrial projects in IDPAs. However, we encountered several limitations motivating improvements for future work. First, we had to make use of the extension mechanisms of the IDPA to represent all requirements. In addition, some concepts such as defining large JSON inputs turned out to be cumbersome. However, the core concepts of the IDPA were suitably expressive for the industrial requirements.

The remainder of this paper is structured as follows. In Section 2, we provide the background of our work and motivate it in more detail. In Section 3, we review related work. Section 4 presents our approach including the IDPA meta-model, the API change type analysis, and the IDPA evolution approach. Section 5 presents our evaluation including the four studies, a discussion of the research questions, and the threats to the validity. Finally, we conclude this paper with Section 6.

2. BACKGROUND AND MOTIVATING EXAMPLE

In this section, we illustrate the limitations of existing approaches and explain the background of our work. For that, we introduce the Broadleaf Heat Clinic [17], which is a showcase implementation of the Broadleaf Commerce Community Edition [18]. Broadleaf Commerce is an enterprise e-commerce platform built on current open-source technologies. The Heat Clinic is a web shop for hot sauces, providing common web shop functionalities like browsing products, collecting them in a cart, and purchasing, as well as managing accounts including addresses, payments, and wish lists. Overall, it provides an API with 184 endpoints as per April 16, 2018. Even though the Heat Clinic is a sample application, we expect it to be elaborate and representative for real web shop applications, because it intends to show the capabilities of the Broadleaf Commerce.
For executing a representative load test against the Heat Clinic, existing approaches can be utilized to extract a workload model from recorded user sessions or requests [2–6]. Typical workload models consist of an intensity, for example, a number of concurrently simulated users, and a user behaviour. Even though often not explicitly separated, the user behaviour can be furthermore divided into a request model and the input data. The request model describes the order and rate of requests submitted by each individual user simulated by the load test executor. Figure 2(a) illustrates an exemplary Markov-chain-based request model for the Heat Clinic’s endpoints home, login, add to cart, and purchase, as it could be automatically generated by existing approaches [3, 4, 6]. Each of the endpoints is represented by one state in a Markov chain. The transitions between the states model the order of and think times between the requests. A load test executor will then simulate the Markov chain by following the transitions and submitting a request for each state. In each of the requests, valid parameter values – that is, input data – are to be used, which typically have to be added manually to the workload model or to the load test to which the workload model will finally be transformed [6].

For illustration, we provide examples of input data for the exemplary request model in Figure 2(b), using an IDPA. For the sake of simplicity, we only show the home and login endpoint and explain only the most relevant elements. All details of the example will be explained in Section 4.3. The application.yml describes the properties of the endpoints, for example, the path and HTTP request method, and the parameters of each endpoint, for example, username, password, and csrfToken for the login endpoint. As defined in the annotation.yml, the input data for these parameters can come from, for example, a counter for the username or the response of the home endpoint, from which the value is extracted by a regular expression, for the csrfToken. Workload models designed for automated extraction typically do not separate the input data from the request model [2, 4–6].

As mentioned before, the input data typically need to be specified manually. The main reason is that while request models can be directly replayed in a test environment – the endpoints are the same in all environments – this is more challenging for input data. Particularly, the databases in the test environment will differ from the production databases, and thus, the input data have to be different.

---

2Cross-Site Request Forgery [19].
for example, different user names and passwords. Furthermore, data dependencies such as the CSRF
token have to be modelled. Finally, certain static properties have to be overridden, for example, for
changing the domain to the test environment’s domain. Hence, the workload model (or the finally
executed load test) has to be parameterized.

While manually parameterizing a once generated workload model or load test might be feasible,
there can be reasons for updating a workload model repeatedly. On the one hand, the Heat Clinic’s
API can change [20], for example, by adding a details page. On the other hand, the production work-
load changes over time, resulting in different relevant workload scenarios that need to be simulated
in a load test, for example, during a normal day, during a marketing campaign, before Christmas,
and so on. With the existing approaches, the complete workload model is regenerated, and an expert
needs to apply the parameterizations again. Especially when dynamically generating load tests in a
CI/CD pipeline (Figure 1), which aims at automating the build, testing, and delivery of the developed
application [8], the resulting manual effort is infeasible. Therefore, our approach aims at removing
the expert from the pipeline by storing the parameterizations, which do not depend on the extracted
request models, separately. We separate an application model and an annotation model in the IDPA,
allowing to reuse it for newly generated workload models. Additionally, API changes require adapt-
ing the parameterizations itself. For this reason, IDPAs can be semi-automatically adapted to typical
API changes collected in literature.

3. RELATED WORK

In the following, we review the related work of our approach. Existing workload models already
comprise input data and static properties but have different limitations regarding automated evolu-
tion. Approaches to continuous validation of load tests address these limitations but are still limited
for the usage in CI/CD pipelines. Generation of test data is orthogonal to our approach. Further-
more, techniques related to our approach can be found in model-driven engineering and commercial
approaches.

3.1. Input data and properties in workload models

Existing workload models store input data and, by nature, static properties as defined in an IDPA.
Many of them store the input data as intrinsic elements of the workload model, that is, as part of the
request model [2, 4–6]. Some of them also provide means for extracting the input data from recorded
requests [2, 6]. However, intrinsically storing input data entails mixing automatically generated
elements (request model) and manually defined elements (input data). Hence, evolving the parame-
terizations is challenging. In contrast to that, the IDPA separates manually defined and automatically
generated elements.

A different set of approaches models input data separately from the request model [21–23]. How-
ever, the provided input data models are specific to the respective request models. Furthermore, the
approaches are lacking in means for extracting workload models from recorded user sessions or
requests. Opposed to that, the IDPA is applicable to any workload model and is explicitly designed
for the usage with automatically extracted request models.

Finally, there are domain-specific languages (DSLs) for load or performance testing, which also
model workloads including input data and further parameterizations [24, 25]. Similar workloads are
modelled by load testing tools [16, 26, 27]. However, similar to the first set of the aforementioned
workload models, these approaches do not separate input data and request models and thus hinder
the evolution of parameterizations.

3.2. Continuous validation of generated load tests

All described approaches entail certain challenges leading to manual effort for maintaining load tests
over workload and API changes. Specifically, there are no means for automatically transferring once
defined parameterizations to newly generated workload models. To overcome these limitations, Syer
et al. [9] and Chen et al. [7] propose approaches to continuous validation of generated load tests. By
comparing test and production logs, they detect differences in the workloads to decide whether the
load test has to be regenerated. Hence, manual effort is still entailed each time the workloads differ sufficiently but is reduced to the times a load test regeneration is required. In contrast to that, the maintenance effort of our IDPA approach is independent of the amount of newly generated load tests and can be completely decoupled from test execution time. Therefore, our approach is much more suitable for use in CI/CD pipelines and when multiple load tests are to be generated and evolved.

3.3. Test data generation

An orthogonal technique to our approach is automated generation of test data. Test data generation for functional tests has been a research interest for a long time [28]. Because manually generating test data is cumbersome, automation significantly eases testing. With special regards to load and performance testing, Barros et al. [29] and Farahbod et al. [30] introduced approaches to transferring production data sets to test environments by analyzing the data relationships and obfuscating sensitive data. As opposed to our approach, test data generation approaches generate data in the test environment’s databases but do not focus on specification in a load test. Because we do not focus on test data generation, these approaches are a good complement to ours.

3.4. Model-driven engineering techniques

Model-driven software engineering (MDSE) [31] comprises several techniques that are related to our approach. Similarly to manually adjusting generated load tests, MDSE often requires to adjust generated program code. The recommendation of MDSE is to separate generated and non-generated code [31, 32], similar to our approach of separating IDPAs from workload models. Another analogy to our IDPA approach are model annotations that are added to development models to enrich the transformation to a resulting model [33]. Similarly, we add IDPAs to workload models to enrich the transformation to load tests. Finally, there are approaches to the synchronization of evolving parallel models [34, 35], as we do with workload models, API models, and the IDPA.

3.5. Commercial approaches

As load testing is widely used in practice, there are several commercial tools. Two of the most prominent ones are Micro Focus LoadRunner [10] and Micro Focus Silk Performer [11]. Such tools are powerful in the creation of IDPA-related parameterizations. For instance, correlation of IDs such as the CSRF token of the Heat Clinic can often be performed automatically. With special regards to integration into CI/CD pipelines, continuous load testing arose [36–38]. Similar to us, such approaches aim at integrating load testing in CI/CD pipelines by reducing the test overhead and adding automation. However, representative load testing based on generated workload models is rarely applied in practice, and commercial tools do not natively support it. Hence, they do not support evolution of load tests over workload changes, either.

4. APPROACH

In this section, we describe our approach to the evolution of manual load test parameterizations. We introduce and utilize the input data and properties annotation (IDPA) storing the parameterizations separately from the generated workload models and load tests. Hence, we are able to reuse IDPAs when generating new representative load tests based on up-to-date request logs. As a result, IDPAs allow to be evolved over changing workloads automatically and to be used for automatically generating representative load tests in a CI/CD pipeline. To prepare for changing APIs as well, we analyze typical API change types presented in the literature and develop approaches for handling them with the least possible maintenance effort. For an overview of the technical realization of our approach, we provide a demo online [13].

4.1. Load test parameterization process overview

Figure 3 illustrates the intended use of IDPAs. An IDPA refers to one application to be load-tested – for example, the Heat Clinic – and is developed by an expert – for example, a developer (1).
Furthermore, parts of an IDPA can be automatically extracted from API specifications such as OpenAPI [39] (2). Precisely, the IDPA application model such as the `application.yml` in the example in Figure 2(b) can be extracted. The annotation model – that is, the `annotation.yml` – needs to be configured manually. However, we allow extracting an initial version of the annotation from a workload model in case it contains request parameter values. Also, we provide means for semi-automatically evolving annotations over changes in the application model. For usage for load test generation, IDPAs are to be stored in a repository, for example, alongside the code in the code repository (3).

In order to generate representative load tests, we need to access the latest user sessions or requests. We propose using a measurement repository where the monitored sessions or requests from the application running in production are stored (4). In this way, a CI/CD pipeline can access it when being executed. Similarly, the pipeline can retrieve the IDPA from the code repository (5). To execute the load test, the pipeline first generates it from the user sessions or requests, and by using the IDPA (6). As motivated in Sections 1 and 2, this generation can be performed fully automatically.

The described process allows to generate and execute a representative load test in a CI/CD pipeline without user interaction. Everything that needs to be configured by an expert – that is, the load test parameterization – is performed before committing the code and the IDPA to the repository. The main advantages of this process compared to generating and parameterizing a load test offline is that up-to-date recorded user sessions or requests can be dynamically used and that workload or API changes are covered.

In the following sections, we present the details of the IDPA meta-model and describe the evolution approach.

### 4.2. Concepts behind input data and properties annotations

When designing the IDPA meta-model, we took several concepts as a basis, which are described in the following.

- **Separation of automatically and manually created artifacts.** An annotation needs to refer to the endpoints and parameters of the target application’s API. In practice, API specifications like OpenAPI [39] are often used to describe REST APIs. For this reason, we divide the IDPA into an application model that holds information about the API and can be generated automatically from OpenAPI specifications, and an annotation model holding the manual specifications.

- **Separation of “where” and “what.”** Input data specifications consist of two basic attributes: “what” data are to be used and “where” to place it. In our Heat Clinic example (Section 2), examples for the “what” are the list of user names or the extracted CSRF token value. For the “where”, examples are the user and CSRF token parameters of the login endpoint. For better understandability and to avoid duplicated information, we separate “what” and “where.” As an example, the CSRF token is a parameter that is required for several endpoints of the Heat
Clinic. By specifying the CSRF token data once and referring to it for all of the endpoints, we prevent redundant specification of the data.

- **Extensibility.** There are various different load testing tools allowing to specify input data differently. It is infeasible to integrate all of them in the IDPA. In contrast, this can result in IDPA models that cannot be properly transformed to a load testing tool, for example, if two specifications that solely exist in two different tools are used. Therefore, we focus on a reasonable subset that is sufficient for our evaluations and allow to extend the IDPA easily.

- **Traceability.** For evolving IDPAs over changes in the production workload and the target application’s API, we need to be able to trace the evolution of single elements of an IDPA. For this purpose, each element holds a unique ID that can be traced.

- **Integration with commonly used technologies.** Following model-driven software engineering (MDSE) practices, we implement the IDPA as a modelling language [40]. For increasing the acceptance of our approach, it is important to integrate it into the technology stack of CI/CD pipelines [41]. Therefore, we use the lightweight, commonly used YAML format [42] (.yml extension) for the serialization. YAML is also used for state-of-the-art technologies such as Docker [43] and OpenAPI [39].

![Diagram](image-url)
4.3. The input data and properties annotation

In the following, we describe the IDPA in detail. The meta-model of the IDPA conforms to the JavaScript Object Notation (JSON) schema [44] language, which can also be expressed in YAML. For better readability, we present a more compact representation in Figure 4. The complete meta-model can be found in the supplementary material [14]. The (de)serialization is implemented in Java and available online [45]. Figure 2(b) presents an example conforming to the IDPA JSON schema. As already described, the IDPA is divided into an application model and an annotation model.

As shown in Figure 4(a), an application model consists of one central Application that subsumes all Endpoints of the target application. Currently, we provide HttpEndpoints holding information about the domain, port, path, request method, protocol, and headers. As an example, two HttpEndpoints of the Heat Clinic are login and home (Figure 2b). The Endpoint is an extension point, meaning that new sub types can be added without affecting the remainder of the IDPA. For instance, an AmqpEndpoint could be added holding information about an AMQP server and a queue name [46]. Each of the Endpoints owns a list of Parameters of a respective type. The HttpEndpoint holds HttpParameters that have a type and a name. In the example, the login endpoint has parameters username, password, and csrfToken of type FORM (application/x-www-form-urlencoded). Additionally, all OpenAPI parameter types are provided. Similarly to the Endpoint, new Parameter sub types can be added, for example, an AmqpParameter for the AmqpEndpoint.

An annotation consists of one central ApplicationAnnotation that subsumes several EndpointAnnotations, Inputs, and Overrides. An EndpointAnnotation refers to an Endpoint, for example, login, and subsumes several ParameterAnnotations referring to Parameters of the Endpoint, for example, username, password, and csrfToken, to describe “where” to place input data. For specifying the “what,” a ParameterAnnotation refers to exactly one Input. All annotation types are independent from implementations of Endpoints and Parameters allowing to add new implementations easily. Furthermore, all annotation types hold Overrides, which we will describe later.

As described before, there are various types of input data specifications. Therefore, the Input of the IDPA meta-model is an extension point allowing to add new input types if required. Currently, we provide the following types, illustrated in Figure 4(c). A ListInput holds a list of data to be used. The list can either be directly specified in an IDPA as a DirectListInput or by referring to a CSV file as a CsvInput. In addition, a ListInput may refer to several other associated ListInputs meaning that when used together in one EndpointAnnotation, the data of both ListInputs are always to be retrieved from the same index. A CounterInput allows to specify large amounts of data concisely by using a counter. It holds a format, start value, increment, maximum, and scope. For the login endpoint, we use a CounterInput to model different user names, as shown in Figure 2(b). The scope defines whether the counter is global for all simulated users of the load test or whether each user has an own instance. A JsonInput allows to assemble JSON body values from other Inputs. For that, it holds a data type, which can be a string, number, object, or array. Furthermore, it holds a name. In the case of a string or number, the input is defined by the name, data type, and a reference to another Input from which the value is retrieved. In the case of an object or array, the items reference to several other JsonInputs is to be used instead of the input reference. That is, the referenced JsonInputs are added as nested elements – either as object attributes or as array elements. The JsonInput was first introduced by Angerstein [47]. The last provided input type is the ExtractedInput. This type of input is to be used to extract values from the responses of former requests, for example, for the csrfToken. For this purpose, it holds several ValueExtractions that refer to Endpoints from which the value is to be extracted. The value is either extracted using a regular expression and a match number for locating the value in the response as a RegExExtraction or from a specific JSON path as a JsonPathExtraction. A template for assembling the extracted value and an optional fallback value can also be defined. In our example, the value for the csrfToken is extracted from the response of the home request using a regular expression.
Table I. API change types and mapping to change types in literature

(a) Proposed API change types

| Change type                  | Code   | Can be evolved automatically |
|------------------------------|--------|------------------------------|
| Add endpoint                 | AE     | ✗                            |
| Remove endpoint              | RE     | ✓                            |
| Change endpoint [property]   | CE     | ✓                            |
| Add parameter                | AP     | ✗                            |
| Remove parameter             | RP     | ✓                            |
| Change parameter [property]  | CP     | ✓                            |
| Change input                 | CI     | ✗                            |
| Change response behaviour    | CR     | ✗                            |

(b) Mapping to Li et al. [49]

| Change type                                   | Mapping                  |
|-----------------------------------------------|--------------------------|
| Add or remove parameter                       | AP, RP                   |
| Combine/split methods                         | RE, AE                   |
| Delete method, unsupport request method       | RE                       |
| Rename method                                 | CE[path]                 |
| Rename parameter                              | CP[name]                 |
| Change format of parameter                    | CI, CF[encoding]         |
| Change type/format of return value, expose data| CR                       |
| Parameter, restrict access to API             | CI                       |
| Change default value of parameter             | –                        |

(c) Mapping to Sohan et al. [50]

| Change type                                   | Mapping                  |
|-----------------------------------------------|--------------------------|
| Move API elements                             | RE, AE, RP, AP           |
| Rename API elements                           | CE[path], CP[name]       |
| Behaviour change                              | CR                       |
| HTTP header change                            | CE[header]               |
| Post/error condition change                   | –                        |

(d) Mapping to Fokaefs et al. [48]

| Change type                                   | Mapping                  |
|-----------------------------------------------|--------------------------|
| Inline type                                   | –                        |
| Adding new types                              | CI, CR                   |
| Changing input or output types                | CI, CR                   |
| Aggressive evolution                          | CI, CR                   |
| Renaming variables                            | CI, CR                   |

(e) Mapping to Wang et al. [20]

| Change type                                   | Frequency | Mapping                  |
|-----------------------------------------------|-----------|--------------------------|
| Change resource URL                           | 11.52%    | CE[domain, base-url]     |
| Change/add/delete response format             | 1.52%     | CR                       |
| Change/add authentication model                | 2.18%     | CI                       |
| Change (API/method/parameter) rate limit      | –         | –                        |
| Change method name                            | 15.65%    | RE                       |
| Change response format                        | 41.52%    | AE                       |
| Change authentication model                   | 0.43%     | CR                       |
| Change domain URL                             | 6.96%     | CE[domain]               |
| Delete method                                 | 2.83%     | CI                       |
| Add method                                    | 0.87%     | CI                       |
| Add error code                                | 4.57%     | RP                       |
| Change parameter name                         | 11.09%    | AP                       |
| Change format or type                         |            |                          |
| Change required type                          |            |                          |
The final concept of the IDPA are the already mentioned Overrides. With an Override, a static property such as the domain name of a generated load test can be overridden. It holds an OverrideKey to identify the property and a string value as replacement. There is one OverrideKey sub type for Endpoints and Parameters, respectively. For direct use, we provide two enumerations of OverrideKeys, shown in Figure 4(b). With an HttpEndpointOverrideKey, the domain, port, protocol, and base URL of an HttpEndpoint can be overridden. For instance, if the Heat Clinic can be accessed via the path / in the production environment, but via /test/stage in the test stage, this can be specified as a base path. The HttpParameterOverrideKey can be used to change the encoding of an HttpParameter. Further OverrideKey enumerations can be added by extending the existing ones. Each OverrideKey can also be used in more high-level scopes, meaning that the Override applies to all annotations in that scope. That is, to change the base path of all endpoints, we can specify the respective override in the ApplicationAnnotation.

4.4. Analyzing API change types

To be able to evolve IDPAs over changes in the target application’s API, we investigate existing literature to identify types of API changes that can occur. We found relevant papers by Fokaefs et al. [48], Li et al. [49], Sohan et al. [50], and Wang et al. [20]. Overall, the authors investigated API changes of 25 different applications. Wang et al. also provide frequencies of occurrence for endpoint-level and parameter-level changes. The API change types we identified and a mapping to the change types defined in literature are provided in Table I.

Based on the literature survey, we identified eight change types that are relevant for IDPA evolution (Table Ia). First, there are additions and removals of endpoints and parameters that require adding or deleting annotations in the IDPA. In the literature, these change types can be found such as Add/Delete Method and Add/Delete Parameter in the list by Wang et al. (Table Ie). Two further change types are property changes of endpoints or parameters which require updating the application model of an IDPA. Rename Method/Parameter in the list by Li et al. (Table Ib) are examples for equivalent change types from the literature. As changed properties, we found domain, path, and header for endpoints and name and encoding for parameters, but changing other properties is also thinkable. The change type Change Input subsumes all changes that can require adaptation of the input data specified in an IDPA. Because this is a very broad category, there are various change types from literature that fall into it. The last change type is Change Response Behaviour that denotes all changes causing different request responses. Changes of this type might impact RegExExtractions of the IDPA and thus require adaption. Similarly to Change Input, Change Response Behaviour subsumes several proposed change types. Some of the proposed change types do not map to our types because there is no impact on the IDPA.

4.5. Evolving IDPAs over changing APIs

Being aware of the possible API change types, we develop a strategy to evolve IDPAs over API changes. As one of our main goals, the maintenance effort for this process has to be minimized. For the integration into CI/CD pipelines, it is especially important that all manual intervention is offline from the pipeline execution and thus from the load test execution. Hence, the evolution takes place after API changes have been introduced and before the IDPA and the changes are committed to the repository (Figure 3).
Our proposed process to adapting an IDPA to API changes is illustrated in Figure 5. The process starts with an application expert, for example, a developer, who introduces a delta of the API (1). Most likely, the delta will be introduced by a code change. If an API specification like OpenAPI is used, it will be updated and automatically transformed to replace the former application model of the IDPA (2a). By basing on unique IDs of the API specification, we can ensure that similar application model elements can be identified. For instance, the login endpoint will have the same ID before and after the update, but some details such as the path or parameters might change. If no API specification is used, the application expert can also modify the application model manually (2b). However, as API specifications are widely used in continuous software engineering [51], this manual effort can be prevented in most cases. Because the application model was replaced, references of the annotation model of the IDPA to the application model might not be valid anymore (3). As an example, if the CSRF token was not used anymore in the new version of the login request, the ParameterAnnotation pointing to it would be broken after the update. By tracing the unique IDs of the model elements, our approach detects the changes of the application model and the invalid references automatically and provides them to the expert (4). The expert can then adapt the annotation model (5) and commit the IDPA and the changed application for load testing to the repository (6).

Table I(a) summarizes the ability to evolve changes automatically in this process. Change Endpoint/Parameter Property is fully covered by the automated adaption of the application model, because these properties are only stored there. Manual adaption is not required. Remove Endpoint/Parameter will most likely cause invalid references of the annotation model that are reported to the expert. As a default adaption, the model elements that hold the invalid references will be removed automatically. However, we recommend to double-check the removals as the annotation elements could be reused, for example, if new related endpoints or parameters are introduced, which can use the same parameterizations. Add Endpoint/Parameter cannot be handled automatically, because these change types require domain knowledge of the expert. At least, our approach provides information about the additions to support the expert. Change Input and Change Response Behaviour have to be handled by the expert.

To summarize, our IDPA evolution approach still entails manual intervention in case of API changes but limits it to adapting the elements that are affected by a change. Minor changes can be handled automatically. Because experts can adapt the IDPA before the commit to a CI/CD pipeline, load testing can be automatically executed without manual intervention as part of the pipeline.

5. EVALUATION

In this section, we evaluate our approach with respect to the research questions introduced in Section 1. In summary, we conduct four different studies. First, we conduct two experimental studies with Sonatype Nexus and the Broadleaf Heat Clinic, which are both realistic web applications. In particular, Sonatype Nexus is a real-world application used by about 10 million developers world-wide [15]. By comparing derived effort estimation models, we discuss the asymptotic maintenance effort required for load test parameterization with and without our approach. Finally, we evaluate the expressiveness of the IDPA in four industrial load testing projects. In the following, we first revisit the research questions and introduce the two metrics used in our evaluation. Then, we provide the methodology and results of the four studies. Finally, we discuss the research questions and threats to validity.

5.1. Research questions

In Section 1, we defined four research questions to be addressed in our evaluation. We explain which study we conduct for each question:

RQ1 – How much does the parameterization by our approach impair the representativeness of a load test?

This question targets the ability of an IDPA to parameterize a load test properly. We consider representative load tests for Sonatype Nexus defined without an IDPA and parameterize...
it with an IDPA. The difference of the results of the two tests will indicate how much an IDPA impairs the representativeness. The results are presented in Section 5.3.

RQ2 – To which degree do evolved parameterizations improve the representativeness of a generated load test?

Having evaluated the general ability of an IDPA to parameterize a load test, we target its intended usage scenario, which is the parameterization of a generated load test. We utilize the WESSBAS [6] approach to generate representative load tests for the Broadleaf Heat Clinic from the measurement data of a reference workload and execute them once as they are and once parameterized with an IDPA. We vary both the reference workload and the Heat Clinic’s API for assessing the improvement of the representativeness though the parameterization with an IDPA in different scenarios. The results are presented in Section 5.4.

RQ3 – To which degree can we reduce the maintenance effort for the evolution of manual parameterizations of generated load tests over changes in the target application’s API?

As we aim at reducing the maintenance effort for parameterizing a generated load test, we discuss the effort reduction in Section 5.5. The outcomes are effort estimation models that can be compared asymptotically. We leave further empirical studies to parameterize our models and hence assess the maintenance effort in precise metrics such as person days for future work.

RQ4 – How expressive is our approach compared to parameterizations of load tests used in industrial projects?

Finally, we evaluate the expressiveness of the IDPA for parameterizing industrial load tests. For this, we consider existing load tests of four different industrial projects and derive IDPAs that can replace the existing parameterizations. In doing so, we evaluate how many parameterizations can be expressed with existing inputs and whether we can add missing inputs via the existing extension points. The results are presented in Section 5.6.

5.2. Metrics

In our evaluation, we target three different measures: RQ1+2 address the representativeness of parameterized load tests compared to different references, RQ3 deals with maintenance effort, and in RQ4, we consider expressiveness. Because we discuss the expressiveness qualitatively, we do not use a specific metric for that. For evaluating the representativeness and the maintenance effort, we define two appropriate metrics $\rho$ and $\theta$.

5.2.1. Representativeness – $\rho$. To assess the representativeness of the generated load tests, we develop a metric $\rho$ comparing the measurements of the load test execution with a reference measurement. $\rho$ has the following characteristics. (i) It has an optimum at 0. (ii) It has one scalar value per load test execution. (iii) It is based on the request rates per endpoint. (iv) Because wrong input data often lead to HTTP error response codes (e.g. 400 or 500), they are considered in the metric. (v) It considers the percental deviation of the request rates to all endpoints equally, (vi) except for very small request rates which have a lower impact on the metric value, because high percental deviations are likely in this case. A further important aspect of a representativeness metric is the time. While the request rates and response codes might be met well in summary after the whole test, there can be greater differences during some periods. Accounting for that, we will consider the metric for a small time unit such as 1 min and calculate the cumulative sum as an overall measure for the whole test.

To calculate the metric value, we represent the measured request rates $x_{i,c}$ to endpoint $E_j$ with response code $c \in \{200, 302, 400, 500\}$ as a matrix $X^{(i)} := (x_{i,c})$. For instance, a part of the matrix resulting from the reference load test of the first iteration of the first experiment series is the following.

\[
\begin{align*}
E_{\text{home}} & \quad 200 & \quad 302 & \quad 400 & \quad 500 \\
E_{\text{add}} & \quad 81.3 & \quad 0 & \quad 0 & \quad 0 \\
E_{\text{logout}} & \quad 49.4 & \quad 1.5 & \quad 0 & \quad 0 \\
E_{\text{add}} & \quad 0 & \quad 70.2 & \quad 0 & \quad 0
\end{align*}
\] (1)

© 2019 The Authors. Software Testing, Verification & Reliability Published by John Wiley & Sons Ltd. Softw. Test. Verif. Reliab. 2020;30:e1712. DOI: 10.1002/stvr
In the following, let $X_{\text{ref}}$ be the reference measurement. Let $X_{\text{gen}}$ furthermore be the measurement of a generated load test execution. The Frobenius distance $\|X_{\text{ref}} - X_{\text{gen}}\|_F$ constitutes a metric conforming characteristics (i) to (iv). To meet (v) as well, we normalize the measurements with a matrix

$$\mathcal{N} := \text{diag}(X_{\text{ref}} \cdot \mathbb{1})^{-1}, \quad \mathbb{1} \in \mathbb{R}^4$$

To meet (vi), we introduce a weight function for the request rates. We want to have low weights on low request rates and high weights on high request rates with an asymptotic value of 1. Therefore, we use the logistic function

$$w(x) = \frac{1}{1 + e^{-kx} \left( \frac{1}{w(0)} - 1 \right)}$$

In order to determine the steepness $k$, we define values of $w$. We define $w(0) := 0.01$ and, because we only want to have low weights on the lowest request rates, $w(10) := 0.9$. This results in $k = 0.6792$. For $x = (x_1, \ldots, x_n)^T$, we define $w(x) := (w(x_1), \ldots, w(x_n))^T$ and yield a weight matrix:

$$\mathcal{W} := \text{diag}(w(X_{\text{ref}} \cdot \mathbb{1})), \quad \mathbb{1} \in \mathbb{R}^4$$

Finally, we define the representativeness metric $\rho$ for the measurement $X_{\text{gen}}$ of a generated load test as

$$\rho(X_{\text{gen}}) := \|\mathcal{N} \cdot \mathcal{W} \cdot (X_{\text{ref}} - X_{\text{gen}})\|_F$$

To yield a baseline for the metric, we execute one load test $p$ times for each tested system and retrieve measurements $X_j$, $j = 1, \ldots, p$. Then, we calculate $\rho(X_j)$, $j \geq 2$ per minute using $X_1$ as reference, respectively. We calculate the mean $\mu_\rho$ and standard deviation $\sigma_\rho$ from the resulting measures and use $\mu_\rho \pm 3\sigma_\rho$ as baseline. A baseline for the cumulative $\rho$ after $t$ minutes is defined by $t\mu_\rho \pm 3\sqrt{t^2\sigma^2_\rho}$.

5.2.2. Maintenance effort – $\theta$. To measure the maintenance effort required for creating and evolving IDPAs or direct load test parameterizations, we introduce a metric $\theta$. It summarizes the effort for applying IDPA changes $c = (t_c, o_c, k_c)$ consisting of the changed element’s type $t_c$, the change operation $o_c \in \{A, C, R\}$ denoting add, change, and remove operations, and the number of changes $k_c$ of that type and operation. The types of possibly changed elements are EndpointAnnotation, ParameterAnnotation, and subtypes of Input. We denote the effort introduced by one change as $\gamma(t_c, o_c)$. Hence, $\gamma(t_c, o_c)$ depends on the complexity of applying a certain operation to a certain IDPA element type. Given $C$ holds all changes introduced to an IDPA, the resulting formula for the overall effort metric can be expressed as follows:

$$\theta(C) = \sum_{c \in C} k_c \cdot \gamma(t_c, o_c)$$

Calculating concrete values for $\theta$ – for example, measured in person hours – requires extensive empirical studies for determining the precise values of $\gamma(t_c, o_c)$. This is out of the scope of this paper and we leave it for future work. Instead, we still can use the metric for qualitative comparisons. In doing so, we consider the $\gamma(t_c, o_c)$ as abstract but constant variables, based on which we derive formulas describing the asymptotic effort. The results of empirical studies can then be used to parameterize the formulas and compute concrete values.
5.3. Experimental study with Sonatype Nexus

In our first study, we address RQ1: How much does the parameterization by our approach impair the representativeness of a load test? For that, we execute an experiment series with Sonatype Nexus [15], which is a widely used open-source build artifact management system. We generate a representative load test based on real-world access logs and parameterize it using an IDPA. Then, we can compare the results of the original test to the parameterized test results. In the following, we first describe the methodology of the experiments and then present the experiment results.

5.3.1. Methodology.

5.3.1.1. System under test. The system under test in our experiments is Sonatype Nexus. Its intent is the management of build artifacts such as Maven Project Object Models (POMs) or Java Archives (JARs). Its most famous public instance is Maven Central, which is used in worldwide software development projects. The program code of Nexus is open-source and available on GitHub. In our experiments, we utilize Nexus version 2.11, which is available on Docker Hub. The Nexus API we based on comprises 135 endpoints for uploading, browsing, searching, and retrieving build artifacts. In addition, there are endpoints for user management.

5.3.1.2. Prerequisites. Before executing the actual experiments, we needed to meet several prerequisites. First, we needed a representative load test for Nexus. For that, we recorded the access logs of the publicly available Nexus instance of an IT company within 3 months. Our goal was to replay the same load which happened on this Nexus instance. For increasing the workload and thus increasing the reliability of the experiment results, we increased it as follows. First, we split the logs into sessions based on the client IP address and interaction pauses of at least 30 min. Then, we randomly sampled and concatenated sessions for 100 parallel threads until each thread lasted at least 30 min. When concatenating two sessions, we added a wait time of 5 min. The resulting threads can be replayed by a load test representing a varying and representative load that is high enough for reliable comparison. For repeated execution of different workloads, we generated 20 sets of 100 threads each. In the following, we refer to these threads as original.

As a second prerequisite, we derived an IDPA application model from the official Nexus API description and the access logs. Furthermore, we extracted all requested artifacts and other parameter values from the logs and stored it into an IDPA annotation as DirectListInputs and CsvInputs. In addition, we used a JsonInput for the rarely occurring POST requests. Hence, we aimed at parameterizing the load tests similarly as the original requests. For that, we annotated the previously generated threads, that is, replaced each request with the corresponding endpoint of the application model and parameterized it with the annotation of the IDPA. We name the resulting threads annotated. All artifacts that were successfully requested in the access logs were stored into a separate file for populating the Nexus for the experiments. As a consequence, we could execute the original access logs without modification against our test instance of Nexus and result in the same responses as originally.

The last prerequisite was the definition of load tests that replay the generated sessions. For that, we used the open-source load driver JMeter [16]. We utilize 100 basic loops, which each have their own thread as generated above and send the defined requests with the defined wait times. For the parameterized threads, we transformed the IDPA to JMeter test plan elements so that the parameterized requests of the threads will use the such specified input data. All threads, JMeter load tests, and the IDPA can be found in the replication package [14].

---

§https://search.maven.org/
¶https://github.com/sonatype/nexus-public
||https://hub.docker.com/r/sonatype/nexus
**https://repository.novatec-gmbh.de/
††https://oss.sonatype.org/nexus-restlet1x-plugin/default/docs/rest.html
5.3.1.3. Experiment process. We executed the experiments as illustrated in Figure 6. We executed 20 iterations, each using one of the 20 sets of original threads (1) and the corresponding annotated threads (2). We merged them with the defined JMeter tests (3) and executed the two tests against a dedicated Nexus instance sequentially for 30 min each (4). Finally, we collected the JMeter results for comparison (5) and continued with the next iteration. Furthermore, for determining a baseline of the \( \rho \) metric, we executed the first iteration 20 more times in both the original and annotated variant. Before each load test execution, we restarted and redeployed Nexus and populated it using the artifacts extracted from the access logs.

5.3.1.4. Experiment setup. For executing the experiments, we utilized two machines, which are connected with a common 1 Gbit switch. Both have an Intel® Xeon® CPU E5620 with 2.40 GHz clock frequency, four cores, and eight threads. The first machine has 8 GiB memory and hosted (i) the Sonatype Nexus in a Docker container and (ii) a lightweight Spring Boot service offering a REST API for restarting the Nexus. The second machine has 32 GiB memory and hosted the JMeter load driver and a shell script process that ran the experiment series automatically.

5.3.2. Results. In the following, we review the results of the experiment series with Nexus. The raw data and the precise analysis of all iterations are available online as part of the replication package [14]. First, we consider the first iteration individually. Figure 7(a) shows the requests per minute divided by endpoint and response code. It can be seen that the number of requests varies over time. The most frequent response code is 404 (Not Found), which can be attributed to clients that are checking whether a certain artifact is present. On a first sight, there is no visual difference between the request rates of the original and the annotated results.

For analyzing potential differences further, we calculate the \( \rho \) metric. As a first step, we calculate a baseline for the \( \rho \) metric using the experiment series we executed for this purpose. For both the original and annotated tests, we consider the first iteration as reference and calculate the \( \rho \) metric for each of the remaining 20 executions per minute. For the resulting 20 · 30 values of \( \rho \), we calculate the mean and the standard deviation, which describe the mean error and variance of the results of one and the same test. These are \( \mu_{\rho, \text{orig}} = 0.0018 \) and \( \sigma_{\rho, \text{orig}} = 0.0038 \) for the original baseline, and \( \mu_{\rho, \text{ann}} = 0.0490 \) and \( \mu_{\rho, \text{ann}} = 0.0431 \) for the annotated baseline. We then use \( \mu_{\rho, \text{orig}} \pm 3\sigma_{\rho, \text{orig}} \) as baseline.

Figures 7(b) and 7(c) show the cumulative \( \rho \) metric calculated for the annotated results with the original results as reference per minute compared to the two baselines. For the original baseline,
it can clearly be seen that $\rho$ is greater than the baseline. Hence, the representativeness is impaired, which cannot be explained with the normal variation of the original test results. However, for the annotated baseline, it is only slightly greater than $\mu_{\rho,\text{ann}}$ and lies inside the range of three standard deviations. Hence, most of the inaccuracy can be explained with the normal variations of the annotated test results, which is apparently larger than for the original test. We explain this finding by the difference in the original and annotated JMeter tests. While the original test only replays the recorded requests, the annotated test also loads CSV files, which serve as feeders for the parameterized requests. Before each request, a new line of each CSV file is loaded. Hence, small delays are introduced, which can slightly change the inter-request timings. This effect will especially impair the $\rho$ metric when there are requests of many different endpoints, such as between minutes 5 and 7 in the first iteration, because some of the requests can be counted in a different minute. $\rho$ is relatively high during this time range, as it can be seen in the sharp increase of the cumulative plot. This hypothesis is supported by the fact that most of the highest values of our baseline calculation were at minute 7.

The remaining iterations 2 to 20 produce similar results compared to the first iteration, as illustrated in Figure 8(a). We show the cumulative $\rho$ metric at the end of the test per iteration compared to the annotated baseline. Except for iteration 9, $\rho$ is slightly greater than $\mu_{\rho,\text{ann}}$ but still inside the baseline. Iteration 9 is even less than the baseline. We can conclude that most of the inaccuracy of the annotated test results can be explained with the normal variation. However, because of the high amount of iterations with $\rho$ values greater than the baseline mean, we also conclude that the representativeness is slightly but systematically impaired. As already mentioned before, the most frequent response code is 404, which is returned if a requested artifact is not present. Hence, if the annotated tests request a different amount of artifacts that are not present, the number of 404 responses will differ from the original tests and hence $\rho$ will increase. Therefore, we correlate the difference in the number of 404 responses between the original and the annotated test – $\Delta$ in the following – with $\rho$, which is shown in Figure 8. Except for one outlier – which is iteration 8 – a clear trend can be seen. This is supported by a fitted linear model $\sum \rho = \beta_1 + \beta_2 \Delta$, which results in $\beta_1 = 1.7730$, $\beta_2 = 0.0032$, and a deviance of 0.8594. We explain this finding by an imperfect IDPA annotation, which parameterized the requests slightly different than it was originally. In fact, we used the overall distribution of requested artifacts in the IDPA annotation, but the original test of each iteration only uses a subset of all artifacts.

To conclude, the representativeness of the annotated tests compared to the original tests – measured by $\rho$ – is slightly impaired. This can be mainly explained with a higher variance of the results of the annotated tests, which is due to the CSV files that need to be loaded. Furthermore, a smaller influencing factor are the slightly different input data. However, the $\rho$ values of all iterations are clearly within a baseline of $\mu_{\rho,\text{ann}} \pm 3\sigma_{\rho,\text{ann}}$.

5.4. Experimental study with Broadleaf Heat Clinic

In a second experimental study, we address RQ2: *To which degree do evolved parameterizations improve the representativeness of a generated load test?* We utilize the WESSBAS [6] approach to generate load tests based on recorded requests and parameterize the generated tests using an IDPA. The study is composed of two experiment series. The first series only considers changes in
the workload to investigate whether an IDPA can be used to preserve the representativeness of a generated load test repeatedly. The second series additionally includes API changes of the system under test. In the following, we describe the experiment methodology and present the results.

5.4.1. Methodology.

5.4.1.1. System under test. As system under test, we use the Broadleaf Heat Clinic we already introduced in Section 2. For our evaluation, we needed to have different versions with different APIs. In the commit history of the Git repository [17], we identified one commit (010f8a2 at August 3, 2017) that introduced API changes. In the following, we refer to the version before this commit as v1. As the second version v2, we denote the current version as per April 16, 2018. For further versions v3 to v20, we adapted version v2 by adding a randomly chosen number between 1 and 5 of the most common change types identified in Section 4.4 in a random order with the frequencies of occurrence provided by Wang et al. [20]. We focused on the change types that can be (semi-) automatically evolved by our approach and that have a frequency of at least 1%. These change types are Add/Remove Endpoint, Change Endpoint Path, Add/Remove Parameter, and Change Parameter Name. Furthermore, we presume that the change operations can be generalized to all kinds of Change Endpoint Property and Change Parameter Property, as they have in common that only the application model has to be adjusted while the annotation remains unchanged. In order not to break functionality, we duplicated randomly chosen elements as additions and removed only duplicated elements. The list of applied changes can be found as part of the replication package [14].

5.4.1.2. Prerequisites. To run the experiments, we needed two main prerequisites. First, we had to define an IDPA that is used for generating load tests. Second, we needed a reference load test that mimics the production workload. To ensure that the input data specifications of the IDPA and the reference load test are equal, we used the JMeter [16] load testing tool for both the reference load test and the generated representative load tests. Furthermore, we defined the IDPA first and specified the inputs in JMeter similarly. For adapting the IDPA to the duplicated endpoints and parameters of the Heat Clinic versions, we duplicated the respective IDPA elements as well. In order to vary the simulated production workload, we designed the reference load test to hold a Markov chain as workload model. Each state of the Markov chain holds the transition probability from one endpoint of the Heat Clinic to another. In order to gain different user behaviour, we varied the transition probabilities. To make sure there are only state transitions that are possible by using the user interface of the Heat Clinic, we defined the allowed transitions in a template that was used as basis for changing the Markov chain.

5.4.1.3. Experiment process. We executed two experiment series with 20 iterations each. In the following, we describe a single iteration of each series.

With the first experiment series, we evaluate whether the IDPA can be automatically applied to a generated load test for preserving the representativeness of the test. Therefore, we used version v2 for all iterations and varied the simulated production workload. One iteration of this series is illustrated in Figure 9. First, we generated a random Markov chain representing the production workload and replaced the original Markov chain of the reference load test (1). In this way, we had a different simulated production workload in each iteration. In the next step, the reference load test was executed against the Heat Clinic (2) and measurement data were collected by the open-source APM tool inspectIT [52] (3). Then, we used the WESSBAS approach [6] to extract a representative workload model from the measured request logs (4). Next, the workload model was transformed to a JMeter load test once considering the IDPA (5) and once without any additional modifications (6). Finally, both generated load tests were executed subsequently (7) and measurement data were collected (8). For evaluation, we compared the results of the three executed load tests. In order to have a clean environment, the Heat Clinic was restarted before each load test execution and populated with 200 user accounts, which were then used by the load tests.

With the second experiment series, we evaluate the effect of API changes. Therefore, we executed the same experiment series as before but increased the version of the Heat Clinic at the beginning.
5.4.1.4. Experiment setup. For executing the experiments, we utilized the same machines as for the Nexus experiments. The first machine hosted (i) the Heat Clinic and (ii) a lightweight Spring Boot service offering a REST API for restarting the Heat Clinic. The second machine hosted the following services: (i) our IDPA evolution approach; (ii) JMeter for executing load tests; (iii) the WESSBAS approach as workload model extractor; (iv) inspectIT for collecting measurement data; (v) an InfluxDB [53] time series database for storing the measurement data; (vi) and a Java process that ran the experiment series automatically.

5.4.2. Results. In this section, we provide the results of the experimental study with the Heat Clinic. The raw data and the analysis results of all iterations are available online as part of the replication package [14].

5.4.3. Varying the workload only. We start the analysis of the results with the request rates. Figure 10(a) shows the request rates per endpoint, HTTP response code, and minute for the first iteration. It can be seen that, except for small variations, the load tests generated with IDPA have similarly looking bars as the reference load tests while the load tests without additional adjustments have a significant amount of erroneous requests on top. These requests target the error page of the Heat Clinic with path /error. Furthermore, it turns out that the overall request rates of the generated tests are slightly smaller than the ones of the reference tests, except for the additional error requests.

Investigating the differences in the load test executions in more detail, we calculate the cumulative $\rho$ metric and a baseline for $\rho$ shown in Figures 10(b) and 10(c) for the selected iterations 1 and 16. For the baseline, we use the results of the reference test we executed 20 times for this purpose with the first iteration as reference test and the remaining 19 iterations as generated tests. That results in $\mu_\rho = 0.7753$ and $\sigma_\rho = 0.1744$ per minute. It can clearly be seen that the metric is higher when not parameterizing the generated load test – $\rho_N$ in the following – compared to using the IDPA – $\rho_W$ in the following. While $\rho_W$ is very close to $\mu_\rho$ in iteration 1, $\rho_N$ is significantly larger than the baseline area. However, in iteration 16, $\rho_N$ is close to $\mu_\rho$, too, even though it is slightly larger while $\rho_W$ is constantly less than $\mu_\rho$. 

Figure 9. One experiment iteration.

Figure 10. Different measures of selected experiment iterations with varied workload per minute. The plots are labelled as reference tests (R) and generated tests with (W/solid line) and with no (N/dashed line) IDPA.

of each iteration, starting at v1. Furthermore, we used the IDPA evolution mechanisms to adapt it to the APIs of the respective versions and adjusted the reference load test. All IDPAs and reference tests can be found in the replication package [14].

DOI: 10.1002/stvr

Softw. Test. Verif. Reliab. 2020;30:e1712.

© 2019 The Authors. Software Testing, Verification & Reliability Published by John Wiley & Sons Ltd.
is greater than the baseline $/SYN \rho_N$ in all iterations except for 16 and 14. In contrast to that, $\rho_W$ is within the baseline area in all iterations and even less than $\mu_\rho$ in 17 iterations. The average cumulative values at test end are 11.8374 for $\rho_N$ and 34.5529 for $\rho_W$. Furthermore, there is no significant trend when using the IDPA. Let $v$ be the iteration number. Fitting a linear model $\sum \rho_W = \beta_1 + \beta_2 v$ results in $\beta_1 = 12.0722$, $\beta_2 = -0.0224$ and a deviance of 7.8675, not indicating any significant upward trend. We conclude that the generated tests without parameterization significantly impair the representativeness. In contrast to that, the representativeness of the tests generated with IDPA is clearly within normal variations of the reference test results.

Finally, we analyze the consequences of the less representative load tests. For this purpose, we investigate the response times of the home endpoint of the Heat Clinic during the respective load tests, shown in Figure 11(b). Obviously, the box plots of the reference test and the generated test with IDPA look relatively similar while the response times of the generated tests without adjustments appear to be smaller. To verify that the difference is significant and to measure the effect size, we apply a two-sided $t$-test and Cohen’s $d$. Even though our samples are not normally distributed, they have large sample sizes between 670 and 1536. Thus, we can apply the $t$-test due to the central limit theorem. As significance level, we use 0.05. Our null hypothesis for iteration $j$ is $H_{j,0}: \mu_{\text{ResponseTime}_{\text{ref},j}} = \mu_{\text{ResponseTime}_{\text{gen},j}}$.

For the IDPA tests, $H_{j,0}$ is rejected and $H_{j,A}$ is accepted for nine iterations while $H_{j,0}$ cannot be rejected for the remaining 11 iterations. Cohen’s $d$ is smaller than 0.2 and thus negligible for all 20 iterations with a maximum absolute value of 0.1997. For the tests without adjustments, $H_{j,0}$ is rejected for all of the iterations. Cohen’s $d$ is small for eight iterations with values between 0.3183 and 0.4888 and medium for 12 iterations with values between 0.5346 and 0.7377.

5.4.4. Varying the workload and the API. The request rates of the second experiment series look very similar to the ones shown in Figure 10(a). The $\rho$ metric appears similar to the one before as well, as shown in Figure 12. As before, $\rho_W$ is close to $\mu_\rho$ in all iterations and clearly within $\mu_\rho \pm 3\sigma_\rho$. Also, there are iterations where the difference between $\rho_W$ and $\rho_N$ is very large – such as iteration 1 (Figure 12a) – and others, where it is less – such as iteration 15 (Figure 12b). A visible difference in Figure 12(c) is that $\rho_N$ decreases and stays small after iteration 10. We explain this finding by a change in the API. A potential candidate is the duplicate of the logout endpoint. The difference of $\rho_N$ and $\mu_\rho$ appears to be slightly smaller than the $\rho_W$ in all iterations. As before, $\rho_W$ is very close to $\mu_\rho$ in all iterations and clearly within $\mu_\rho \pm 3\sigma_\rho$. Also, there are iterations where the difference between $\rho_W$ and $\rho_N$ is very large – such as iteration 1 (Figure 12a) – and others, where it is less – such as iteration 15 (Figure 12b). Again, we fit a linear model $\sum \rho_W = \beta_1 + \beta_2 v$ into the measured cumulative values at test end. It results in $\beta_1 = 11.9536$, $\beta_2 = 0.2699$, and a deviance of 47.0205. Hence, the fitted model indicates a small upward trend, but with a high fitting error compared to the first experiment series.

5.5. Maintenance effort estimation models

In the following, we discuss the difference of the maintenance effort required for evolving an IDPA compared to not using an IDPA, that is, repeatedly parameterizing a load test directly. This is described in RQ3: To which degree can we reduce the maintenance effort for the evolution of manual parameterizations of generated load tests over changes in the target application’s API?
As described in Section 5.2, determining the concrete maintenance effort requires empirical studies related to [54], which is left for future work. Therefore, we utilize the metric $\theta$ for deriving formulas that can be compared asymptotically. We first introduce our methodology on deriving the formulas and then present the results of the discussion.

5.5.1. Methodology. The goal of our discussion is to compare the asymptotic effort required for evolving IDPAs compared to parameterizing a load test repeatedly. In doing so, we assume that the amount and distribution of changes introduced to the target system’s API is constant over time. Even in the case that this assumption does not hold, it affects both parameterization approaches equally and thus allows for a fair comparison. As a second assumption, we consider the effort for changing an IDPA element and the respective load test element equal. For instance, we assume the effort for mapping an Input to a parameter using a ParameterAnnotation is equal to setting the input value of a JMeter request parameter. This assumption is valid, as we are only comparing the asymptotic behaviour.

For deriving the formulas, we consider multiple iterations. In each iteration, a new load test is generated and parameterized. Between two iterations, certain API changes can be introduced. In a first step, we derive formulas depending on the average amount of introduced changes per iteration. These formulas will be different for IDPA evolution and direct load test parameterization. In a second step, we then determine the amount of different IDPA or load test element changes based on the frequencies presented by Wang et al. [20] and the API composition of the Sonatype Nexus, which we used in our previous experiment. As a result, we yield more concrete formulas only depending on few variables that can be compared.

5.5.2. Results. For the following discussion, we introduce several terms. For convenience, we use “change” as a synonym for a change to an IDPA or a load test parameterization. In contrast to that, an “API change” denotes a change to the tested system’s API. With $\theta_0$, we denote the effort for creating an initial load test. This effort will be the same for both parameterization approaches, because it requires adding the same parameterizations. With $C$, we denote a set of changes introduced in one iteration while $\bar{C}$ denotes all possible changes, that is, all combinations of possible values of the changed element’s type $t_c$ and the change operation $o_c$. We furthermore define $n := \sum_{c \in C} k_c$ as the amount of changes per iteration. $\alpha(t_c, o_c)$ is defined as the average number of changes per type and operation to be applied because of one API change. In the first step, we consider $\alpha(t_c, o_c)$ as an abstract variable, but we will parameterize it in the second step based on the average distribution of API changes.

Using these definitions, we can express the effort required for evolving an IDPA from iteration $i - 1$ to $i$ in dependence of $n$ and $\alpha(t_c, o_c)$. Furthermore, we can abstract from the precise efforts $\gamma(t_c, o_c)$ by defining the maximum value $\Gamma := \max(\gamma(t_c, o_c))$:

$$\theta_{\text{IDPA}}(C, i) = \sum_{c \in \bar{C}} k_c \cdot \gamma(t_c, o_c) \approx n \cdot \left( \sum_{c \in C} \alpha(t_c, o_c) \cdot \gamma(t_c, o_c) \right) \leq n \cdot \Gamma \cdot \sum_{c \in \bar{C}} \alpha(t_c, o_c) = \text{const.}$$
Hence, the upper estimate of $\theta_{\text{IDPA}}$ is constant and especially does not depend on the iteration $i$.

The main difference when parameterizing a load test directly is in the set of changes to be applied in each iteration. In contrast to using an IDPA, all parameterizations have to be redefined from scratch. However, there are no remove or change operations. Instead, the corresponding parameterizations can be left out or defined correctly according to the new version. As a consequence, when reusing the $\gamma(t_e, o_c)$ of the IDPA, the $\theta$ formula is different. First, we need to apply all changes from the previous iteration again plus the changes with an add operation. However, we need to subtract all remove operations but with the effort of adding the respective parameterization type. Then, we can again estimate the formula with the $\alpha(t_e, o_c)$ and the minimum effort $\gamma := \min(\gamma(t_e, o_c))$.

In the formula, we use $\bar{T}$ as the set of all possibly changed types $t_e$. The effort for direct load test parameterization (dltp) in iteration $i$ is then

$$\theta_{\text{dltp}}(C, i) = \theta_{\text{dltp}}(C, i - 1) + \sum_{c \in C, o_c = A} k_c \cdot \gamma(t_e, A) - \sum_{c \in C, o_c = R} k_c \cdot \gamma(t_e, A)$$

$$\approx \theta_{\text{dltp}}(C, i - 1) + n \cdot \sum_{t_e \in T} (\alpha(t_e, A) - \alpha(t_e, R)) \cdot \gamma(t_e, A)$$

$$\geq \theta_{\text{dltp}}(C, i - 1) + n \cdot \gamma \cdot \sum_{t_e \in T} (\alpha(t_e, A) - \alpha(t_e, R))$$

$$= \theta_0 + i \cdot n \cdot \gamma \left( \sum_{t_e \in T} \alpha(t_e, A) - \sum_{t_e \in T} \alpha(t_e, R) \right)$$

It can be seen that in contrast to using an IDPA, the effort depends on the iteration $i$. Furthermore, the final factor of the sums of $\alpha$ influence the effort metric.

For sustainably comparing $\theta_{\text{IDPA}}$ and $\theta_{\text{dltp}}$, we parameterize the formulas with concrete values of $\alpha(t_e, o_c)$. For that, we base on the API change frequencies determined by Wang et al. [20]. As illustrated in Figure 13, we can translate the API change frequencies to IDPA change frequencies. For that, we first translate them to API change frequencies based on our API change types introduced in Section 4.4. Then, we determine which IDPA changes would be required and translate the frequencies. Hence, each number attached to each change operation and type denotes the number of respective changes to be applied for one API change with the distribution defined on the left. These numbers are equal to $\alpha(t_e, o_c)$. Because additions or removals of EndpointAnnotations can also require additions or removals of ParameterAnnotations and Inputs, we base on the average number of parameters per endpoint based on Sonatype Nexus. It has 135 endpoints with 149 parameters overall and hence 1.1037 parameters per endpoint on average.

![Figure 13. Changes to an IDPA resulting from API changes. The numbers denote the number of changes of the respective type required for one API change according to the distribution determined by Wang et al. [20].](image-url)

The boxes on the right hold the sum of the number of changes next to it.
REDUCING THE EFFORT FOR PARAMETERIZATION OF REPRESENTATIVE LOAD TESTS

Figure 14. Upper estimate of the cumulative $\theta_{\text{IDPA}}$ (solid line) and lower estimate of the cumulative $\theta_{\text{dltp}}$ (dashed line) measured in multiples of $n \cdot \gamma$ over the number of iterations.

Setting the $\alpha(t_c, o_c)$ to the determined values results in the following formulas of $\theta_{\text{IDPA}}$. In addition, we calculate the cumulative sum, which represents the overall effort spent for parameterizing load tests until an iteration $p$. As the average number of changes per API change is 2.2252, it is:

$$\theta_{\text{IDPA}}(C, i) \leq n \cdot \Gamma \cdot 2.2252$$

$$\sum_{i=0}^{p} \theta_{\text{IDPA}}(C) \leq \theta_0 + p \cdot n \cdot \Gamma \cdot 2.2252$$

We do the same for $\theta_{\text{dltp}}$ resulting in the following formulas. The average number of additions per API change is 1.5536, while the average number of removals is 0.5933. Hence, the difference is 0.9603 resulting in the following:

$$\theta_{\text{dltp}}(C, i) \geq \theta_0 + i \cdot n \cdot \gamma \cdot 0.9603$$

$$\sum_{i=0}^{p} \theta_{\text{dltp}}(C, i) \geq (p + 1)\theta_0 + \frac{1}{2} p(p + 1) \cdot n \cdot \gamma \cdot 0.9603$$

It can be seen that the cumulative effort is linear when using an IDPA while it is quadratic when parameterizing load tests directly. As illustrated in Figure 14, the precise relation of the two approaches depends on the initial effort $\theta_0$ and the relation of the maximum effort per change $\Gamma$ and minimum effort $\gamma$. The figure shows the upper estimate of the cumulative effort with IDPA and the lower estimate of the cumulative effort with direct parameterization. The values we chose are arbitrary but show the three different scenarios we identified. The first scenario is that in the beginning, there is no significant difference between the two approaches, which however becomes larger with more iterations (Figure 14a). In the second scenario, the effort for direct parameterization is significantly higher from the first iteration after $\theta_0$ on (Figure 14b). Finally, the effort of direct parameterization could be lower than with an IDPA in the beginning, but finally become larger (Figure 14c). Regardless of the scenario, the parameterization using an IDPA will always have less effort in the long term because of the linear asymptotic behaviour compared to the quadratic behaviour of the direct load test parameterization.

5.6. Industrial case study

In this industrial case study, we address RQ4: How expressive is our approach compared to parameterizations of load tests used in industrial projects? For this purpose, we analyze the load tests used in four different industrial software development projects. In the following, we provide the methodology, present the results, and provide the lessons we learned during conducting the case study.

5.6.1. Methodology.

5.6.1.1. Design and planning. We selected the case study subjects from the authors’ existing contacts. The subjects are four different industrial projects from the construction and automotive sector.
In each of the projects, a certain web-based software application is developed, which comprise both user-faced and backend applications. For the sake of confidentiality, we refer to the projects as A, B, C, and D in the following.

Each of the projects develops and executes load tests for the developed application including IDPA-related parameterization concepts, which constituted our source of information. We considered all load tests of all projects in our study to reach the most possible diversity of parameterization concepts. We were given full access to the load tests for analysis purposes.

5.6.1.2. Data collection. The precise artifacts we were given access to are all load tests including accompanying artifacts that are needed to execute the tests. Table II provides a summary of the number of targeted endpoints, the number of load tests per project, and the evaluation results. All load tests are implemented in Scala using the Gatling tool [26]. Furthermore, the load tests of each project utilize a common code base, for example, implementing requests to specific endpoints of the application, and configuration files such as CSV or JSON files specifying input data. All load tests have been implemented manually by the respective development team, based on known or intended usage of the application. Hence, they are not representative as presumed in this paper. However, the set of parameterizations used in a load test is independent from the fact whether the test is generated or manually implemented. For conducting the case study, we had access to the load tests including the common code bases and all configuration files of each project.

5.6.1.3. Data analysis. For each of the projects, we analyzed the provided load tests. We identified all specifications that were used to parameterize the tests. These are all specifications defining input data for parameters or modifying the execution of a request from its default. For instance, if a specific header is explicitly defined for a request, we considered it being such a specification. In addition, we identified all specifications that would be necessary to transform a request from the production system to the test system. That is, assuming a request was extracted from production request logs, we determined the changes that would need to be made to bring the request into the actual form. Having all parameterizations identified, we tried to express them using the IDPA. In case this was not possible, we introduced new concepts using the provided extension points.

We performed the steps described earlier for each project individually and sequentially, starting from project A and ending with project D. In each step, we introduced extensions of the IDPA if necessary and reused them in the next project if possible. In this way, we could determine whether the introduced extensions can be generally used and thus could be added to the core IDPA or whether each project requires individual parameterization concepts.

5.6.2. Results. An overview of the results of our industrial case study is provided in Table II. We show the number of load tests, utilized Overrides, utilized Core inputs contained in the core IDPA meta-model described in Section 4.3 and utilized custom Inputs, which we introduced using the extension points of the IDPA. Also, we provide the percentage of custom Inputs over all utilized Inputs.
5.6.2.1. Project A. The first project we investigated has two different load tests that target 7 endpoints and were implemented using the same code base. We identified two overrides which would need to be used for transforming production requests to requests to the test system. First, the domain name needs to be changed with the HttpEndpoint.domain override. Second, each stage in this project has a specific base path, such as /test/stage or /dev/stage‡‡. Hence, we need to override this base path using the HttpEndpoint.base-path override.

The input data concepts used in the load tests correspond to the IDPA entities DirectInput, CsvInput, and ExtractedInput. For the ExtractedInputs, JsonPathExtractions are used to extract the input data from returned JSON bodies. We were able to represent all parameterizations solely using the provided IDPA concepts. Hence, the percentage of custom inputs is 0%.

5.6.2.2. Project B. Project B has one load test, which targets 17 endpoints and consists of several Scala code files. As overrides, we only identified HttpEndpoint.domain for adjusting the domain name of the requests. The utilized core inputs were DirectInput, ExtractedInput, and JsonInput. The ExtractedInputs were both used with RegExExtractions and JsonPathExtractions. The JsonInputs were used to define JSON values which consist of both static values and dynamically retrieved values, for example, from ExtractedInputs.

However, the load test of this project utilized several parameterization concepts which we could not represent using the core inputs. Therefore, we introduced custom inputs using the Input extension point. First, the load test used randomly generated Universally Unique Identifiers (UUIDs) as an input. We introduced a RandomStringInput, which takes a template as parameter. The template defines the format of the generated strings and is based on regular expression syntax. For the UUIDs, we used the template [0-9A-D]{8}-[0-9A-D]{4}-[0-9A-D]{4}-[0-9A-D]{4}-[0-9A-D]{12}. The second extension we introduced is the RandomNumberInput which randomly selects a number between a lower and an upper limit. The limits can be either defined as constants or by retrieving a value from another input. In this case, the lower limit was set to 0 while the upper limit was extracted from a JSON response. Next, the load test utilized an application-specific authentication mechanism. This mechanism generates a specific header to be used in all requests. For that, we introduced the AuthInput which generates these headers. A FilteredInput was used for selecting a certain percentage of multiple values from another Input. In this case an ExtractedInput utilizing a JsonPathExtraction. Finally, for generating dates, which were used as input of some parameters, we introduced the DatetimeInput. This Input produces the current date and time in a defined pattern (e.g. yyyy-MM-dd) and with a defined offset (e.g. 6M for 6 months). The offset is optional. Overall, seven of overall 41 utilized Inputs were of the newly introduced custom types, which is 17.07%.

5.6.2.3. Project C. Project C also has one load test defined in a single source file, accompanied by configuration files. It targets a single endpoint. As before, the HttpEndpoint.domain override would need to be used for transferring production requests to the test system. The used core inputs are DirectInput, CsvInput, and JsonInput.

For this project, we also had to make use of custom extensions. First, we could reuse the DatetimeInput introduced for Project B. This time, the used pattern was the time stamp in seconds. Furthermore, we had to add an ability to use input data defined as environment variables. For that, we introduced the EnvironmentInput, which reads the value of an environment variable and also allows to define a certain template into which the value should be inserted. Overall, the percentage of custom inputs is 0.3%.

Even though we were able to represent all parameterizations as an IDPA, we encountered limitations of the JsonInput and the CsvInput. The JSON body of one request of the load test is defined by a JSON file, which has 1250 lines. We represented this JSON by a JsonInput but resulted in a much longer description with 3094 lines. Furthermore, due to the recursive structure of the JsonInput, we had to define 470 nested JsonInputs. Because of this and because

‡‡These base paths do not reflect the actual paths and are only for illustration purposes.
the JsonInput does not exactly show the JSON structure, readability is degraded. This finding motivates implementing a new input, which allows specifying long JSON strings more concisely. The limitation of the CsvInput is because in this load test, a CSV file with multiple columns is used. However, the CsvInput only allows to use one column. Hence, we needed to add one CsvInput per column and associate the inputs, which resulted in a less concise definition and redundant information, such as the CSV file’s name.

5.6.2.4. Project D. The last project we investigated contains three different load tests. In addition, there are 12 small tests which are used for synthetic monitoring, that is, for executing them against the production system regularly to check whether the system behaves as intended. Because the technology of these tests is similar to the load tests, we considered both test types for our evaluation. Overall, the tests target 34 different endpoints.

The overrides we utilized are HttpEndpoint.domain and HttpEndpoint.header for adding a set of defined headers to the requests. The used core inputs are DirectInput, CsvInput, ExtractedInput with a RegExExtraction, and JsonInput. From the already introduced custom inputs, we made use of the RandomStringInput, RandomNumberInput, DatetimeInput, and EnvironmentInput. However, we also had to introduce another custom input for combining the values of several other inputs. The CombinedInput consists of a list of Inputs and a template which defines how to combine the Inputs. In this case, we utilized the CombinedInputs for defining a combination of randomly chosen numbers, which we generated using the RandomNumberInput, to yield a random but valid date string. Overall, the percentage of custom inputs is 9.3 %.

Another kind of parameterization we identified in this project are explicitly defined HTTP cookies. For one request, the load tests define a cookie, which is determined dynamically and is to be set before sending the request. With the existing concepts of the IDPA, we were not able to represent this concept, because it can neither be mapped to Overrides nor to Inputs. Overrides are not suitable, because they cannot deal with dynamically defined values. Inputs are not appropriate, because they can only define the whole value of a parameter – in this case, the cookie header – and cannot add a value to the existing list of cookies. Therefore, we did not take the cookies into account in the IDPA.

5.6.3. Lessons learned. During the execution of the case study, we learned several lessons, which we summarize in the following.

5.6.3.1. Most input data specifications are common. As is can be seen in Table II, the clear majority of input data specifications used in the projects are list-based, extracted, and JSON input data. Precisely, 97.85 % of all input data specifications fall into these types. This fact indicates that the data flow of most applications can be described with these means. As a consequence, load testing tools or input data specification approaches such as the IDPA should at least implement these kinds of input data specifications.

5.6.3.2. Individual input data specifications are required. Even though the input data specifications that do not fall into the aforementioned common categories are only 2.15 %, there still exist cases in which they are required to parameterize a load test properly. Particularly, there is a need for adding application-specific input data specifications, such as the AuthInput in Project B. Without this input, the authentication header required in each request could not be generated. Hence, custom input data specifications should be considered by load testing or input data specification approaches. In the original load test implemented in Gatling, the comprehensive capabilities of the underlying Scala language were used. Other load testing tools such as JMeter allow adding plugins implementing custom functionality. In our IDPA, we addressed this challenge by extending the Input, to implement custom inputs such as the AuthInput.

5.6.3.3. Input data can be large. A challenge we faced when defining the IDPA was the size of some input data. An example is the large JsonInput based on a JSON file with 1250 lines in
Project C. Hence, the solution used to define this JSON value needs to be scalable. Other examples of large input data are CSV files, such as one in Project C with more than 25,000 lines. Furthermore, CSV files can have multiple columns which each define a set of input data. Therefore, it is crucial to provide input specifications such as the CsvInput, which allow referring to external files instead of inlining its content. In our approach, we encountered limited scalability of the JsonInput and CsvInput, which we will address in future work.

5.6.3.4. **Common input data specifications are often sufficient.** Another finding is that people tend to use comprehensive specification mechanisms if they are available, even if more standard mechanisms would be sufficient. For instance, Project D utilizes the CombinedInput and several RandomStringInputs for generating a random date. The same result could be achieved either by using a single RandomStringInput with an appropriate template or even by using a CsvInput with a CSV file prefilled with random dates. The same applies for the DatetimeInput used in this project, which is utilized for generating a random but unique number. However, the specifications used by the project are most convenient, because neither complex templates have to be defined nor long lists of input values have to be generated. Furthermore, there are also cases where the DatetimeInput, RandomNumberInput, and RandomStringInput are required, because either the current date has to be defined or a random value based on extracted values has to be generated. Both is the case in Project B.

Therefore, most input data can be defined with the common types of specification but with potentially less convenience. At the same time, there can be specifications that cannot be mapped to the common types. As a consequence, input data models such the IDPA do not need to implement all variants of imaginable input data specifications but also need to provide means for several non-standard specifications for being usable in industry.

5.7. Discussion

In this section, we discuss the results of our evaluation with respect to the research questions.

5.7.1. **RQ1 – Impairing the representativeness.** In our first experimental study with Sonatype Nexus, we were able to parameterize representative load tests, which replayed recorded requests from a productive Nexus instance, while mostly preserving the representativeness. The $\rho$ metric we calculated was within a baseline representing the normal variations of such parameterized load tests. However, we also encountered two factors that slightly impaired the representativeness. First, the variation of the parameterized tests was higher than the original tests, which is because of CSV files that need to be loaded before each request. Second and more important, there was a small but systematic variation from the baseline mean, because we were not able to achieve the same ratio of artifacts successfully requested to artifacts that could not be found (response code 404). This can be attributed to the fact that the parameter values of the requests influence whether an artifact will be found and thus how the internal program flow behaves. Opposed to Nexus, in our second experimental study with the Broadleaf Heat Clinic, no such effects could be observed, as the input data do not significantly influence the program flow, as long as they are properly defined.

We conclude that for applications whose workload and internal behaviour are dominated by the order and rate of submitted requests, the IDPA can be used for reliably parameterizing a load test without impairing the representativeness. In contrast to that, load tests for applications with significantly different behaviour for different parameter values can only be as representative as the input data defined in the IDPA. Hence, the benefit added by a parameterization through an IDPA is less, because the dominant part of the workload model has to be defined manually. In the case of our experiments with Nexus, this effect could be observed, even though we still were able to define input data resulting in a representativeness within the baseline.

5.7.2. **RQ2 – Improving the representativeness.** In the second experimental study, we used the IDPA for its intended purpose, namely, the parameterization of generated load tests. Compared to unparameterized load tests, the IDPA could significantly improve the representativeness. This was reflected in the $\rho$ metric. While the metric values for the parameterized tests can be well explained
with the normal variations of the reference tests, the values for the unparameterized tests were much higher than the baseline in general. Also, we could not detect any trend at least when only the work-load is changing. With a changing API, there were more variations, and we could also detect a small trend of decreasing representativeness with the IDPA, even though the linear correlation was not statistically significant. In the end, all $p$ values were clearly within the baseline for the parameterized tests in all iterations, while they were outside the baseline for the unparameterized tests in most iterations.

An important finding is that reduced representativeness can also impair internal behaviour of the tested application, such as the response times. In our experiments, the response times of the unparameterized tests were significantly different from the ones of the reference tests with small to medium effect sizes. In contrast to this, the tests parameterized with the IDPA only have significantly different response times in less than half of the iterations with a negligible effect size. We can conclude that improper or missing parameterization of generated load tests can lead to a different behaviour of the tested application and thus to corrupt load test results. The IDPA turned out to be one possibility to add the required parameterizations to the generated tests repeatedly.

5.7.3. RQ3 – Reducing the maintenance effort. For addressing RQ3, we derived estimation models for the maintenance effort and compared them asymptotically. Even though we could not determine the precise effort in units such as person days, our models show that using an IDPA reduces a quadratic cumulative effort over time to a linear cumulative effort. Irrespective of the precise values of the parameters of the models, this will lead to a significantly reduced effort when using an IDPA in the long term. More importantly, the IDPA does not only reduce the maintenance effort but moves the effort away from test generation time. Hence, new load tests can be generated and executed fully automatically, as required by our overall load testing vision [12]. In future work, we suggest parameterizing our derived models with concrete values, which could be determined by empirical studies.

5.7.4. RQ4 – Expressing industrial parameterizations. In our case study with four different industrial projects, we were able to represent the adjustments of static properties and input data of all investigated load tests in IDPAs. However, we had to make use of the extension points of the IDPA to introduce new Inputs. Most of the newly introduced Inputs can be shared between the investigated projects. In particular, while we had to add five new Inputs for Project B, we only needed to add one new one for Project C and D, respectively. Hence, we presume that we already covered a majority of all relevant input types. In summary, our core input types introduced in Section 4.3 covered more than 97.85% of all inputs of all investigated projects.

The newly introduced input types motivate adding them to the core IDPA meta-model. However, this is not suitable for all of them. In particular, the AuthInput is specific to Project B. Hence, it should not be added to the meta-model, as it cannot be reused in other projects. In fact, the AuthInput is a good example where extension points are required. The remaining input types are independent from the project or tested application. However, as we designed the IDPA to be tool independent, we need to assess whether these inputs can be implemented with all load testing tools. As a first indication, the JMeter functions RandomString, Random, and timeShift [55] provide similar functionality as the RandomStringInput, RandomNumberInput, and DatetimeInput. As a potential implementation of the EnvironmentInput, most load testing tools allow passing arguments from outside. Hence, these input types are likely to be generalizable to other projects, as there also exist solutions in load testing tools and as they are used by multiple of our investigated projects. Also the CombinedInput is promising, as it is only a combination of other inputs and hence, easily implementable. For the FilteredInput, it is currently unclear whether it is useful for other projects, as it is only required once in Project B. We leave this investigation and the extension of the meta-model for future work.

The case study also revealed three limitations of the core IDPA. First, we encountered that long JSON files cannot be defined concisely using the existing Inputs. The JsonInput enables such definitions but is cumbersome and decreases readability. In the future, we are going to improve the JsonInput for better use with long JSONs. We suggest to implement a new version of

© 2019 The Authors. Software Testing, Verification & Reliability Softw. Test. Verif. Reliab. 2020;30:e1712. Published by John Wiley & Sons Ltd. DOI: 10.1002/stvr
the JsonInput that simply holds the JSON object tree formatted in YAML, potentially referring to other inputs. Even though this is less flexible than the current implementation, it scales better with large JSON values. Second, CsvInputs turned out to define redundant information in case of multiple rows in the CSV file. The CsvInput is designed in this way, because the ParameterAnnotations need to directly link to one individual column. However, we are going to investigate in more compact descriptions of CSV files as well. An improved CsvInput should define the information about the CSV file only once and hold a list of nested Inputs which each represent one column. Last, Project D requires setting a defined HTTP cookie. We were not able to map this to an IDPA concept. However, we presume that explicitly setting cookies is not required for representative load testing, which extracts the request model from recorded requests and represents the behaviour of real users. A real user would interact with the system without setting cookies explicitly. In contrast to that, the load tests of Project C were manually defined and do not precisely represent a real user’s behaviour.

Overall, we conclude that the IDPA is suitably expressive for defining parameterizations of extracted representative load tests of industrial applications. However, the current version lacks in several input types which are required for multiple applications. The Input extension point allows to add these input types. However, we need to investigate which input types need to be added to the core IDPA for future use. Furthermore, we encountered the usefulness of the extension mechanism, because Project B required an input type that is specific to the project and should not be added to the core IDPA, namely, the AuthInput.

5.8. Threats to validity

We see the following threats to the validity of our work, grouped by conclusion, internal, construct, and external validity.

5.8.1. Conclusion validity. In our evaluation, we derived several conclusions that can bear threats to the validity of our study. In the experimental study with the Broadleaf Heat Clinic, we compare response time means using a two-sided t-test. However, the boxplots of the response times appear asymmetric and thus indicate a distribution other than normal distribution. However, as all sample sizes have a size of at least 670, we can apply the t-test according to the central limit theorem.

In the study with Sonatype Nexus, we base our analysis on the results of the JMeter load tests rather than server-side monitoring tools. Hence, there is a latency between the first machine hosting the load test and the second machine hosting Nexus, which could potentially impair the results. However, we presume this latency is relatively small, as the machines are connected with a common 1 Gbit switch. Furthermore, as the low original baseline indicates, there is low variation in the results except for the CSV file loading. Thus, we conclude that there is no significant impairment.

5.8.2. Internal validity. We defined a metric ρ that bases on a weight function w. Potentially, this weight function could slightly impair our results. However, w only reduces the impact of very small request rates, because they are likely to strongly deviate in percentage. Thus, w smoothenes the metric without changing the value for larger, reliable request rates.

Our second metric – θ – assumes that all changes with a certain operation to a certain type of element require the same effort. In practice, there will be more variation. However, for an asymptotic comparison, the assumption is sufficient. When parameterizing our derived effort estimation models with concrete values, future works might consider the variation of the effort, for example, as a probability distribution per change operation and type.

Another threat is that we evaluated the IDPA’s expressiveness with non-representative load tests in the industrial case study. Hence, the findings might not be relevant for generated representative load tests. However, we presume the parameterization concepts used in load tests is independent of the representativeness, as they mostly depend on the tested application.

5.8.3. Construct validity. Our approach and experimental studies assume that test data with the same quality and quantity as the production data are available. In fact, in the experimental studies,
we use the same database for the reference and generated load tests. In practice, this might not be the case. However, generating and managing representative test data are not our research focus. For this purpose, we refer to existing approaches [29, 30].

Another potential threat is the usage of Markov-chain-based workload models. In practice, request sequences are more often used. However, as the IDPA is completely independent from the workload model, we can assume that there are no side effects due to Markov chains. Furthermore, Markov chains are treated to be suitable for load testing [56].

Finally, we artificially introduced API changes to the Heat Clinic for assessing the influence of such changes on the representativeness of load tests parameterized by an IDPA. Hence, our findings could originate from changes that are uncommon in practice. Therefore, we based on the literature [20] to implement changes according to a common distribution. In addition, we considered API changes that were already contained in the commit history between version v1 and v2, which did not result in fundamentally different measurements.

5.8.4. External validity. We concluded from our industrial case study that the IDPA is suitably expressive for defining load test parameterizations. Also, we presumed we covered most of the required input types with the introduced extensions, because most of the input types could be shared between the projects. However, this finding might stem from a similarity of the investigated projects. We faced this issue by investigating four different industrial projects. However, identifying the set of input types that should be added to the core IDPA meta-model is left for future work.

Furthermore, we only investigated web applications having REST APIs. We cannot generalize our results to applications that do not meet this assumption. However, even though the IDPA is not limited to a special type of applications, such a generalization is not a goal of this paper. For future work, additional studies complementing ours are required, especially in other domains and contexts, for example, other than web-based applications.

6. CONCLUSION AND FUTURE WORK

In the context of CI/CD pipelines, representative load testing is a promising technique. However, existing approaches entail unfeasible manual effort for load test maintenance. We introduced the input data and properties annotation (IDPA) for storing manual parameterizations of representative load tests separately. When generating load tests with existing approaches, the IDPA can be merged automatically, enabling frequent updates of the workload without manual intervention. Furthermore, we presented an approach to semi-automatically evolving IDPAs over common API changes presented in literature.

With two experimental studies with Sonatype Nexus and the Broadleaf Heat Clinic in combination with WESSBAS and JMeter, we showed that our approach can parameterize generated load tests without impairing the representativeness. In fact, it can be used to restore the representativeness compared to an unparameterized test as generated by a workload model extractor. From the study with Sonatype Nexus, we concluded that IDPAs are especially meaningful for the use with workloads dominated by the order and rate of requests. Furthermore, the effort estimation models we derived show that the IDPA reduces the maintenance effort in the case of a typical mix of API changes from a quadratic behaviour to a linear behaviour. In the case of no API changes, the effort is even limited to initially creating an IDPA. Finally, we were able to express the parameterizations of the load tests of four projects in an industrial case study using IDPAs. We conclude that the IDPA’s expressiveness is sufficient for the use in industrial contexts. However, we also encountered limitations, which need to be tackled in future works. Therefore, our approach complements existing approaches to representative load testing and resolves the main issue hindering the application of representative load testing in CI/CD pipelines.

For future work, we are going to address the encountered limitations of the current implementation of the IDPA. This comprises the cumbersome specifications of large JSON inputs and CSV files with multiple columns, as well as the introduced extensions that are candidates to be added to the core meta-model. Second, we suggest conducting an empirical study for enriching our derived effort estimation models with precise measures of the effort in, for example, person days, related to
Benestad et al. [54]. Furthermore, we detected a small but systematic error in the request rates generated by WESSBAS. For practical use with confidence, this error has to be resolved. As a starting point, we identified the utilized approach for measuring think times. Finally, it would be beneficial to integrate IDPAs with the powerful commercial load testing tools practitioners are used to [10, 11] as well as with test data generation approaches [29, 30] to yield a production-ready self-contained system. The combination of test data generation approaches and specification in IDPAs can also be used for addressing the challenge of automatically populating the tested application with production-like test data [1].

ACKNOWLEDGEMENT

This work has been supported by the German Federal Ministry of Education and Research (grant no. 01IS17010, ContinuITy). The authors would like to thank our anonymous industrial partners, who participated in our case study. Further thanks go to our following colleagues from the ContinuITy project, who contributed through numerous discussions: Christoph Heger, Stefan Siegl, Philipp Merkle, Dušan Okanović, Tobias Angerstein, Alper Hidioglu, Manuel Palenga, Vincenzo Ferme, and Alberto Avritzer.

REFERENCES

1. Jiang ZM, Hassan AE. A survey on load testing of large-scale software systems. IEEE Transactions on Software Engineering 2015; 41(11):1091–1118.
2. Krishnamurthy D, Rolia JA, Majumdar S. A synthetic workload generation technique for stress testing session-based systems. IEEE Transactions on Software Engineering 2006; 32(11):868–882.
3. Menascé DA. Load testing of web sites. IEEE Internet Computing 2002; 6(4):70–74.
4. Ruffo G, Schifanella R, Sereno M, Politi R. WALTy: A user behavior tailored tool for evaluating web application performance. In Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (NCA 2004), 2004; 77–86.
5. Shams M, Krishnamurthy D, Far BH. A model-based approach for testing the performance of web applications. In Proceedings of the 3rd International Workshop on Software Quality Assurance (SOQUA 2006), 2006; 54–61.
6. Vögele C, van Hoorn A, Schulz E, Hasselbring W, Krcmar H. WESSBAS: Extraction of probabilistic workload specifications for load testing and performance prediction – a model-driven approach for session-based application systems. Software and System Modeling 2018; 17(2):443–477.
7. Chen T-H, Syer MD, Shang W, Jiang ZM, Hassan AE, Nasser MN, Flora P. Analytics-driven load testing: An industrial experience report on load testing of large-scale systems. In Proceedings of the 39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP 2017), 243.
8. Humble J, Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation 1st (ed.) Pearson Education: Boston, USA, 2010.
9. Syer MD, Jiang ZM, Nagappan M, Hassan AE, Nasser MN, Flora P. Continuous validation of load test suites. In Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014), 259.
10. Micro Focus. LoadRunner: Application Load Testing Software. 2019. https://www.microfocus.com/en-us/products/loadrunner-load-testing/overview, Accessed: 2019-08-15.
11. Micro Focus. Silk Performer, 2019. https://www.microfocus.com/products/silk-portfolio/silk-performer/, Accessed: 2019-08-15.
12. Schulz H, Angerstein T, van Hoorn A. Towards automating representative load testing in continuous software engineering. In Companion Proceedings of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE 2018), 2018; 123–126.
13. Schulz H. Continuity-project/idpa-demo: Idpa demo v1.0.0., 2019. https://doi.org/10.5281/zenodo.2647977.
14. Schulz H, van Hoorn A, Wert A. Replication package: Reducing the maintenance effort for parameterization of representative load tests using annotations, 2019. https://doi.org/10.5281/zenodo.3255389.
15. Sonatype Inc. Nexus repository manager – Software component management, 2019. https://www.sonatype.com/nexus-repository-sonatype, Accessed: 2019-08-15.
16. Apache Software Foundation. Apache JMeter 2019. http://jmeter.apache.org/, Accessed: 2019-08-15.
17. The Broadleaf Heat Clinic Spring Boot Application 2019. https://github.com/BroadleafCommerce/DemoSite, Accessed: 2019-08-15.
18. OWASP. Cross-Site Request Forgery (CSRF), 2018. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF), Accessed: 2019-08-15.
20. Wang S, Keivanloo I, Zou Y. How do developers react to RESTful API evolution? In Proceedings of the 12th International Conference on Service-Oriented Computing (ICSOC 2014) volume 8831 Lecture Notes in Computer Science, 2014; 245–259.

21. Draheme D, Grundy J, Hosking J, Lutteroth C, Weber G. Realistic load testing of web applications. Proceedings of the 10th Conference on Software Maintenance and Reengineering (CSMR 2006), 2006; 11–70.

22. Lutteroth C, Weber G. Modeling a realistic workload for performance testing. Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (ECOC 2008), 2008; 149–158.

23. Zhou J, Zhou B, Li S. LTF: A model-based load testing framework for web applications. Proceedings of the 14th International Conference on Quality Software (QSIC 2014), 2014; 154–163.

24. Bernardino M, Zorzo AF, Rodrigues EdM. Canopus: A domain-specific language for modeling performance testing. Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST 2016), IEEE Computer Society, 2016; 157–167.

25. Ferme V, Pautasso C. A declarative approach for performance tests execution in continuous software development environments. In Proceedings of 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2018); ACM, 2018.

26. Gatling Corp. Gatling Open-source load and performance testing – for DevOps and CI/CD, 2019. https://gatling.io/Accessed: 2019-08-15.

27. BlazeMeter. Taurus: A new star in the test automation tools constellation, 2015. https://www.blazemeter.com/blog/taurus-new-star-test-automation-tools-constellation Accessed: 2019-08-15.

28. Ince DC. The automatic generation of test data. The Computer Journal 1987; 30(1):63–69. https://doi.org/10.1093/ocomjnl/30.1.63.

29. Barros MD, Shiau J, Shang C, Sidewohl K, Shi H, Forsmann J. Web services wind tunnel: On performance testing large-scale stateful web services. In Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007), 2007; 612–617.

30. Farahbod R, Dadashi A. Data generation for performance evaluation, 2014. https://patents.google.com/patent/US9613074B2/en Accessed: 2018-11-22.

31. Völter M, Stahl T, Bettin J, Haase A, Helsen S. Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons: Chichester, England, 2013.

32. Steinberg D, Budinsky F, Merks E, Paternostro M. EMF: Eclipse Modeling Framework, 2nd (ed.) Pearson Education: Boston, USA, 2008.

33. Fritzsche M, Gilani W. Non-intrusive model annotation, 2008. https://patents.google.com/patent/US8145468B2/en Accessed: 2018-11-22.

34. Kramer ME, Burger E, Langhammer M. View-centric engineering with synchronized heterogeneous models. In Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO 2014), 2013; 1–6.

35. Milovanovic V, Milicev D. An interactive tool for UML class model evolution in database applications. Software and System Modeling 2015; 14(3):1273–1295.

36. How to include load testing in your continuous integration environment, 2017. https://www.blazemeter.com/blog/how-include-load-testing-your-continuous-integration-environment Accessed: 2019-08-15.

37. SOASTA. Continuous load testing | automated load testing tool, 2018. https://www.soasta.com/solutions/continuous-load-testing Accessed: 2018-04-13.

38. Tricentis GmbH. Continuous load testing, 2018. https://www.tricentis.com/software-testing-tools/continuous-load-testing Accessed: 2018-11-22.

39. OpenAPI Initiative. OpenAPI initiative, 2019. https://www.openapis.org/ Accessed: 2019-08-15.

40. Brambilla M, Cabot J, Wimmer M. Model-Driven Software Engineering in Practice Second (ed.) Morgan & Claypool; San Rafael, USA, 2017.

41. Bezemer C-P, Eisemann S, Ferme V, Grohmann J, Heinrich R, Jamshidi P, Shang W, van Hoorn A, Villavicencio M, Walter J, Willnecker F. How is performance addressed in DevOps? Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE 2019); ACM, 2019; 45–50.

42. The official yaml web site, 2019. http://yaml.org/Accessed: 2019-08-15.

43. Docker Inc. Enterprise container platform, 2019. https://www.docker.com/ Accessed: 2019-08-15.

44. Wright A. JSON schema: A media type for describing JSON documents, 2018. http://json-schema.org/latest/json-schema-core.html Accessed: 2019-08-15.

45. ContinuITy-Project. ContinuITy/continuity.idpa at master, 2019. https://github.com/ContinuITy-Project/ContinuITy/tree/master/continuity.idpa Accessed: 2019-08-15.

46. OASIS Standard. OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0., 2012.

47. Angerstein T. Modularization of representative load tests for microservice applications. Master’s thesis, University of Stuttgart, Germany, 2018.

48. Fokaefs M, Mikhail R, Tsantalis N, Stroulia E, Lau A. An empirical study on web service evolution. Proceedings of the 18th International Conference on Web Services (ICWS 2011), 2011; 49–56.

49. Li J, Xiong Y, Liu X, Zhang L. How does web service API evolution affect clients? Proceedings of the 20th International Conference on Web Services (ICWS 2013), 2013; 300–307.

50. Sohan SM, Anslow C, Maurer F. A case study of web API evolution. Proceedings of the 11th Congress on Services (SERVICES 2015), 2015; 245–252.

51. Newman S. Building Microservices — Designing Fine-Grained Systems, 1st (ed.): O’Reilly, Sebastopol, USA, 2015.

© 2019 The Authors. Software Testing, Verification & Reliability Softw. Test. Verif. Reliab. 2020;30:e1712. Published by John Wiley & Sons Ltd. DOI: 10.1002/stvr
52. NovaTec Consulting GmbH. *inspectIT - the open source Application Performance Management (APM) solution*, 2018. https://inspectit-performance.atlassian.net/wiki/spaces/HOME/overview Accessed: 2019-08-15.

53. InfluxData, Inc. *InfluxDB: Purpose-built open source time series database*, 2019. https://www.influxdata.com/ Accessed: 2019-08-15.

54. Benestad HC, Anda B, Arisholm E. Understanding cost drivers of software evolution: A quantitative and qualitative investigation of change effort in two evolving software systems, 2010; 166–203.

55. *Apache JMeter - User’s Manual: Functions and Variables*, 2019. http://jmeter.apache.org/usermanual/functions.html Accessed: 2019-08-15.

56. Li Z, Tian J. Testing the suitability of Markov chains as web usage models. In *Proceedings of the 27th International Computer Software and Applications Conference (COMPSAC 2003): Design and Assessment of Trustworthy Software-Based Systems*, 2003; 356–361.