The Journal of Systems & Software 183 (2022) 111084

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A model-driven approach for continuous performance engineering in microservice-based systems☆,☆☆

Vittorio Cortellessa, Daniele Di Pompeo, Romina Eramo, Michele Tucci *

DISIM, University of L'Aquila, Via Vetoio, L'Aquila, Italy

ARTICLE INFO

Article history:
Received 23 December 2020
Received in revised form 24 May 2021
Accepted 2 September 2021
Available online 21 September 2021

Keywords:
Performance engineering
Model-driven engineering
Microservices
Software refactoring
Software evolution
Continuous deployment

ABSTRACT

Microservices are quite widely impacting on the software industry in recent years. Rapid evolution and continuous deployment represent specific benefits of microservice-based systems, but they may have a significant impact on non-functional properties like performance. Despite the obvious relevance of this property, there is still a lack of systematic approaches that explicitly take into account performance issues in the lifecycle of microservice-based systems.

In such a context of evolution and re-deployment, Model-Driven Engineering techniques can provide major support to various software engineering activities, and in particular they can allow managing the relationships between a running system and its architectural model.

In this paper, we propose a model-driven integrated approach that exploits traceability relationships between the monitored data of a microservice-based running system and its architectural model to derive recommended refactoring actions that lead to performance improvement. The approach has been applied and validated on two microservice-based systems, in the domain of e-commerce and ticket reservation, respectively, whose architectural models have been designed in UML profiled with MARTE.

1. Introduction

Microservices have become a popular style for architecting a software system as a suite of small services, and they are nowadays adopted by many key technological players such as Netflix, Amazon, and Google. Major benefits of a microservice-based architecture are that it ensures loose coupling, and it supports rapid evolution and continuous deployment. In addition, having a large set of independently developed services helps in terms of developer productivity, scalability, and maintainability. Contextually, the rapidly growing complexity of software systems has forced practitioners to use and investigate different development techniques to tackle advances in productivity and quality. To this extent, software engineering needs to relay on automated approaches to keep low the development costs while tackling the rapid changes of software capabilities that may considerably impact non-functional properties like performance.

In order to manage software complexity, ever more companies in the last two decades have embedded Model-Driven Engineering (MDE) (Schmidt, 2006) approaches in their processes, with the perceived benefit of enabling developers to work at a higher level of abstraction and to rely on automation throughout the development process. Nevertheless, MDE solutions need to be further developed to scale up for real-life industrial projects (Brunelière et al., 2018). To this intent, one of the major challenges is to work on achieving a more efficient integration between the design and runtime aspects of systems. For instance, through observation and instrumentation, logs and metrics can be collected and related to the original software design in order to comprehend, extrapolate and analyze the inner behavior of a running software system (Cito et al., 2018).

In this context, non-functional properties (e.g., performance, power consumption or memory footprint) are becoming ever more relevant for the success of a software application, and the early identification of problems induces lower cost solutions (Woodside et al., 2007). On one side, in model-based software performance engineering, a number of approaches have been proposed for detecting and removing performance problems in software models. Some techniques are based on the concept of performance antipattern, which characterizes bad design practices that may jeopardize software performance, along with possible refactoring actions aimed to remove them (Cortellessa, 2013). On the other side, methods and tools have been proposed

☆ This research was supported by the AIDOaRt project (ECSEL-JU program - grant agreement n. 101007350).
☆☆ Editor: W.K. Chan.
* Corresponding author.
E-mail addresses: vittorio.cortellessa@univaq.it (V. Cortellessa), daniele.dipompeo@univaq.it (D. Di Pompeo), romina.eramo@univaq.it (R. Eramo), michele.tucci@univaq.it (M. Tucci).

https://doi.org/10.1016/j.jss.2021.111084

for monitoring system execution and measuring performance of running systems. However, many of them do not envisage a solid integration with architectural design models (Brunelière et al., 2018). Instead, one of the main benefits in adopting model-based performance evaluation is the ability to conduct analysis (e.g., what-if analysis) that would be expensive on a real system, such as to analyze the system behavior when exposed to different workloads, or to analyze the performance sensitivity to system parameter variations. Basing on a solid connection between runtime information and architectural design, developers can suggest architectural changes aimed, for example, at meeting performance requirements before the system actually experiments certain scenarios (e.g., some specific workloads).

In this paper, we propose a model-driven approach to realize a continuous software engineering loop in microservice-based systems. The approach exploits design-runtime interactions to support designers of microservices in performance analysis and system refactoring tasks. In particular, microservices are monitored by means of distributed tracing, i.e., logs are stored in a central location and metrics for all instances of a given service are aggregated to understand the overall state. The observed system behavior at runtime is related to the architectural design to investigate potential performance issues and to design and implement effective system refactoring actions.

In order to realize our approach, we exploit the specific characteristics of these systems. In fact, each microservice is a separate and autonomous entity that can change independently of each other. This allows us to make a change to a single service and deploy it independently of the rest of the system. In contrast, in monolithic applications any refactoring would impact a large amount of the system, requiring additional coordination among components when making changes; also, in order to release changes, the whole monolithic application should be deployed, implying to manage a higher delta between releases and a higher risk of malfunctioning (Newman, 2015).

In a previous paper (Arcelli et al., 2019), we presented a preliminary version of this approach. It has been realized within Eclipse EMF.[1] It integrates a model-driven framework for the definition of a traceability model between logs extracted from a running system and an architectural model, which has been realized by means of JTL (Cicchetti et al., 2010). Basing on feeding an architectural model with runtime monitored performance indices, we have adopted an end-to-end solution for performance improvement (Arcelli et al., 2018). This paper extends our previous work as follows:

- We extended the work with the translation of refactoring actions suggested by performance model analysis into refactoring actions applied to the running system, thus closing a continuous performance engineering loop that was missing in our original work.
- We have enabled the detection of performance antipatterns on the basis of performance indices gathered from a running system instead of employing model-based estimated performance indices. In addition, we introduce here the extraction of an additional performance index, i.e. CPU utilization. This inevitably leads to more realistic evaluations of performance problems.
- In our previous work, we applied the approach to a single case study, E-Shopper, at the modeling level. In this paper, beside completing the application of the whole performance engineering loop to E-Shopper (i.e., at modeling and running system levels), we also apply the approach on an additional case study, Train Ticket, which is larger than the previous one and it comes from the literature.
- In this work we present an evaluation of the approach; in particular, we answer to two research questions, which we did not introduce in the former study, that are: (RQ1) Do the proposed model refactoring actions improve the performance of the running system? (RQ2) To what extent does performance antipattern (PA) removal improve the whole performance?

As in our previous work, the approach is here applied on microservice-based systems modeled by means of UML (OMG, 2015) profiled with MARTE (OMG, 2008), which is the official OMG[2] profile for augmenting the UML with quantitative knowledge. In particular, for behavioral aspects we consider Sequence Diagrams, whereas for static aspects we consider Use Case, Component, and Deployment Diagrams.

The rest of the paper is organized as follows: Section 2 introduces background information about the techniques used in the work; Section 3 describes the proposed model-driven approach for continuous performance engineering; in Section 4 the approach is applied and validated on two realistic microservice-based systems; threats to validity are discussed in Section 5; related work is presented in Section 6, and finally Section 7 concludes the paper.

2. Background

In the following, we describe the background of this work in terms of existing techniques that have been adopted.

2.1. Monitoring infrastructure

One of the defining characteristics of microservice-based systems is that each service must be independently deployable (Chen, 2018). This aspect favors the independent development of services, but it also makes traditional application monitoring insufficient. While suitable for monolithic applications, the gathering of logs and metrics of each service does not provide a complete understanding of the system behavior. This is the main reason for the adoption of distributed tracing as a mean to correlate events generated in individual services with a transaction traversing the entire system.

In this work, we focused on microservices applications deployed on *Docker*,[3] and developed with *Spring Boot*,[4] and *Spring Cloud*[5]. As a consequence, we chose *Spring Cloud Sleuth*[6] to implement a distributed tracing solution. In *Spring Cloud Sleuth* a trace consists of a series of casually related events that are triggered by a request as it moves through a distributed system. These events are called spans and they represent a timed operation occurring in a component. Spans contain references to other spans, which allow a trace to be assembled as a complete workflow. A span contains a set of basic information: the name of the operation, the name of the component providing the operation, the start timestamp and duration (or, alternatively, the finish timestamp), the role of the span in the request and a set of user-defined annotations called tags. Beside basic information, spans generated by *Spring Cloud Sleuth* also contain the IP address and port number of the service, the Java class and method implementing the operation, as well as the unique identifier of the *Spring Cloud* instance.

[1] Eclipse Modeling Framework: https://www.eclipse.org/modeling/emf/.

[2] Object Management Group: https://www.omg.org/.

[3] Docker: https://www.docker.com/.

[4] Spring Boot: https://spring.io/projects/spring-boot.

[5] Spring Cloud: https://spring.io/projects/spring-cloud.

[6] Spring Cloud Sleuth: https://spring.io/projects/spring-cloud-sleuth.

Once the application is instrumented to produce traces, an infrastructure is necessary to collect and store them. In our approach, the traces produced by each service during the execution are gathered by the *Zipkin*[7] distributed tracing system. In turn, *Zipkin* is configured to forward the monitoring data to the distributed database and search engine *Elasticsearch*.[8]

The resources utilization of individual microservices is another important aspect to consider when monitoring system performance. For this work, we selected *perf*,[9] among the wide range of performance analyzing tools. *Perf* is one of the most commonly used performance counter profiling tools on Linux and supports hardware and software performance counters, tracepoints and dynamic probes. We used *perf* to gather accurate CPU utilization for each microservice. We stored such measures in *Elasticsearch* along with the traces collected by *Zipkin*.

2.2. MDE techniques

Model Driven Engineering (MDE) (Schmidt, 2006) leverages intellectual property and business logic from source code into high-level specifications enabling more accurate analyses. In general, an application domain is consistently analyzed and engineered by means of a *metamodel*, i.e., a coherent set of interrelated concepts. A model is said to *conform* to a metamodel, meaning that the former is expressed by the concepts encoded in the latter. Constraints are defined at the meta-level, and the consistency relationships between models are guaranteed by means of (bidirectional) model transformations specified on source and target metamodels. With the introduction of model-driven techniques in the software lifecycle, also the analysis of non-functional properties has become effective by means of dedicated tools for the automated assessment of quality attributes (Cortellessa et al., 2011).

In this work, we used two model-driven frameworks to define model-driven traceability links to relate software architecture and runtime information and to perform performance analysis and model refactoring, respectively. Such frameworks are introduced in the follows.

2.2.1. JTL

JTL (Janus Transformation Language) (Cicchetti et al., 2010) is an Eclipse EMF-based tool realized to maintain consistency between software artifacts.[10] Its constraint-based and relational model transformation engine is specifically tailored to support bidirectionality, change propagation and traceability. Within the framework, designers can specify model transformations as bidirectional relationships between elements of two domains (i.e., metamodels). The bidirectional engine provides the possibility to apply the transformation rules in both ways, from right to left domains and vice versa. The JTL transformation mechanism provides a relational semantics relying on Answer Set Programming (ASP) (Gelfond and Lifschitz, 1988) and uses the DLV constraint solver (Leone et al., 2006) to find consistent solutions.

In this work, we used the JTL traceability mechanism to store relevant details about the linkage between right and left model elements at execution-time (Eramo et al., 2018). In MDE, such linkages are often based on the concept of traceability relationships, which may help designers to understand associations and dependencies that exist among heterogeneous models (Paige et al., 2011; Winkler and von Pilgrim, 2010). In particular, a *traceability link* is a relationship between one or more source model elements and one or more target model elements, whereas a *trace model* is a structured set of traceability links, e.g., between source and target models. Within JTL, traceability links are extrapolated during the transformation execution and made explicit by the framework. In fact, traceability models are maintained as models conforms to the dedicated traceability metamodel, as defined in its Ecore format within EMF. Traceability models can be stored, viewed and manipulated (if needed) by the designer.

2.2.2. PADRE

PADRE (Performance Antipatterns Detection and model REfactoring) (Arcelli et al., 2018) is a unified framework that tries to improve the performance quality of UML models through a performance antipatterns detection and a model-based refactoring engines, which exploit Epsilon (Kolovos et al., 2010) to implement detection rules and refactoring actions.[11] Furthermore, PADRE employs UML models augmented by MARTE stereotypes in order to link performance data to UML elements. MARTE, which stands for Modeling and Analysis of Real-time and Embedded systems, is the official OMG profile that extends the UML with quantitative knowledge. The MARTE profile is structured in packages and sub-package each one with a specific aim. In our approach, we use stereotypes contained in Generic Quantitative Analysis Modeling (GQAM) package. Through the GQAM stereotypes we are able to bring the runtime data back to the UML element, and on the basis of this runtime knowledge, PADRE can detect and eventually remove performance antipatterns. A performance antipattern (Cortellessa et al., 2014a) is a description of well-known bad design practices that might lead to performance degradation.

Moreover, PADRE is equipped with a performance analyzer, which exploits Queueing Networks and an MVA approximation algorithm to obtain performance indices.

In particular, PADRE can detect eight performance antipatterns, and it provides several refactoring actions. A refactoring action can be either a specific action, i.e., designed to remove specific antipatterns, or a general one, i.e., aimed at improving the system performance quality without targeting specific aspects. PADRE provides three different detection and refactoring sessions, namely user-driven, batch and multiple sessions. The multiple sessions allows to apply more than one refactoring action in a row, the batch session performs refactoring actions until every performance antipattern has been removed, and the users-driven allows the performance expert to select a specific performance antipattern occurrence to be removed. In the presented paper, we employ the user-driver session having the performance expert part of our refactoring loop.

2.3. Performance antipattern

In this section we introduce the performance antipattern (PA) concept. A performance antipattern describes a bad design practice that might lead to performance degradation in a system. This concept is mutated from the design antipatterns one, which describes well-known bad practices that might cause system quality degradation (e.g., low cohesion).

Smith and Williams had textually described a set of performance antipatterns (PA) in Smith and Williams (2002) that they have identified on the basis of existing experiences. Then, this textual descriptions have been translated in first-order logics representation (Cortellessa et al., 2014b), thus enabling the automated detection of PAs. The first-order logics representation of a PA is a combination of multiple literals, where each one maps on a specific system view. Furthermore, every literal is compared

[7] Zipkin: https://zipkin.io/.
[8] Elasticsearch: https://www.elastic.co/products/elasticsearch.
[9] perf: https://perf.wiki.kernel.org/.
[10] JTL: https://github.com/MDEGroup/jtl-eclipse.
[11] PADRE: https://github.com/SEALABQualityGroup/padre.

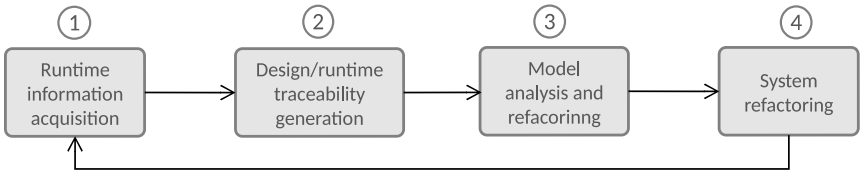


Fig. 1. A high-level workflow of our approach.

to a threshold that represents a safety limit that the system shall not overstep.

In this paper we consider the Blob and the Pipe and Filter performance antipatterns, because they have a larger potential to occur in microservice-based systems. In the following we recap the definition of these PAs.

Blob. It occurs when a single component (also known as God Class) performs the most part of the work of a software system, and its manifestation results in excessive message traffic that may degrade performance. Expression (1) describes the Blob performance antipattern in first-order logics (Cortellessa et al., 2014b).

The first inequality of Expression (1) refers to the number of the exposed interfaces of a component, where a too high number of interfaces is considered as a precondition to identify the component as a God Class. The second inequality, instead, checks whether the God Class is effectively involved in the system. The third inequality refers to hardware utilization of hardware where the component runs. Only if all three inequalities hold then the component is identified as a Blob performance antipattern.

$$\begin{aligned} &\exists c_x, c_y \in \mathbb{C}, S \in \mathbb{S} \mid \\ &F_{numClientConnects}(c_x) \geq Th_{maxConnects} \quad \wedge \\ &F_{numMsgs}(c_x, c_y, S) \geq Th_{maxMsgs} \quad \wedge \\ &F_{maxHwUtil}(P_{xy}, all) \geq Th_{maxHwUtil} \end{aligned} \tag{1}$$

Pipe and filter. It occurs when the slowest filter in a "pipe" causes the system to have unacceptable throughput. This situation is formalized in Expression (2) (Cortellessa et al., 2014b).

$$\begin{aligned} &\exists OpI \in \mathbb{O}, S \in \mathbb{S}, i \in \mathbb{N} \mid \\ &F_{resDemand}(Op) \geq Th_{resDemand} \wedge F_{probExec}(S, OpI) = 1 \quad \wedge \\ &F_{maxHwUtil}(P_c, all) \geq Th_{maxHwUtil} \end{aligned} \tag{2}$$

Differently to the Blob performance antipattern, Pipe and Filter identifies an operation to be the cause of performance degradation. First of all, because the operation requires a too high amount of resources to be executed (i.e., a heavyweight resource demand), as described in the first inequality of Expression (2). Beside this, the operation has to be certainly executed in order to be the cause of performance degradation, and this is checked in the second equality. Finally, either the hardware utilization must exceed a safety threshold (i.e., third inequality) for the Pipe and Filter to manifest itself. Here the first two literals refer to design characteristics of the system, while the last two one to performance properties.

3. Our approach

The idea underlying our approach exploits the correspondences between the architectural design and the runtime aspects of a software system, with the aim of improving its performance.

Fig. 1 depicts four main steps of the continuous performance engineering loop we considered, as follows:

1. *Runtime data acquisition*: Microservice-based systems are monitored by means of distributed tracing; thus, logs are stored in a central location and metrics for all instances of a given service are aggregated to understand the overall state. Then, the collected data are represented in a model-based format compliant with EMF.
2. *Design-runtime traceability generation*: In this phase, the system behavior at runtime is matched with the architectural design by means of traceability models that are automatically generated on the base of correspondences that are formally pre-defined through a metamodel.
3. *Model analysis and refactoring*: The analysis of the above created traceability models aims to connect system performance issues with the affected design components that are identified as possible causes. The results of such analysis lead to the definition of model refactoring actions, that are applied on the system model in order to identify the most promising ones.
4. *System refactoring*: The emerging refactoring actions are implemented and applied to the running system, whose runtime data is acquired and used in the next iteration of this workflow.

In the rest of this section, we describe each step in detail.

3.1. Runtime data acquisition

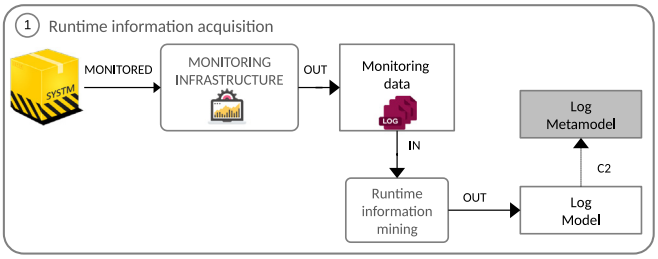


Fig. 2. Runtime data acquisition.

As depicted in Fig. 2, runtime data (i.e., logs/traces) are obtained through a monitoring infrastructure over a running system. The specific infrastructure adopted in this work has been detailed in Section 2.1.

The collected runtime data is then integrated in an EMF-based environment and translated in EMF artifacts. In this step, raw logs, as the one shown in Fig. 4, are automatically transformed in Log Models conforming to a specific Log Metamodel reported in Fig. 3.

The Log Metamodel defines the characteristics of a Log element, which is the root of a log model. A Log stores all the Trace information about requests being sent to EndPoints. Service elements, that may represent the microservices of an application, are associated to both Spans and EndPoints. Services also include a *utilization* attribute setting the percentage of CPU usage during the observation time. A Trace is identified by a unique ID and includes a set of Spans representing execution events. A Span is defined by the following attributes: timestamp describing when

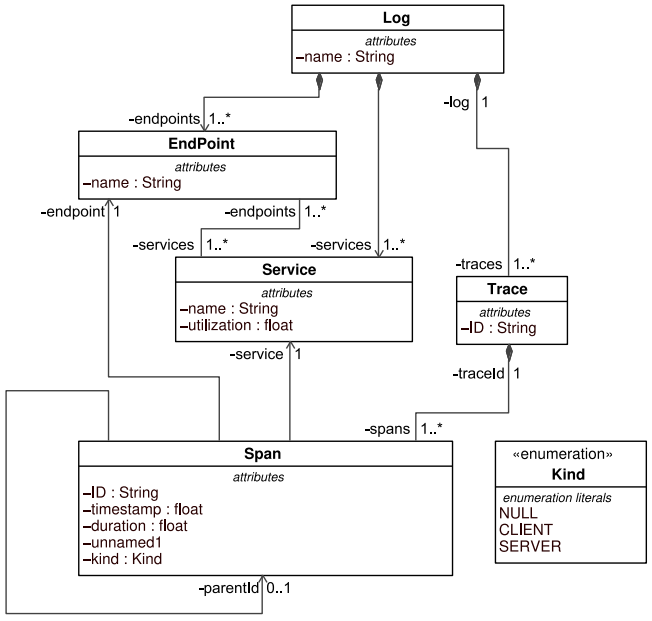


Fig. 3. Log Metamodel.

the event occurs, *duration* describing the time to complete the call, and kind that may be one of SERVER, CLIENT or UNDEFINED. Moreover, when a Span is triggered by another one, the parentId reference connects the triggered Span to the triggering one, called the parent Span. A Span also refers to an EndPoint, which is the URL used to perform a request.

Fig. 5 depicts a sample of a Log Model that represents the original logs shown in Fig. 4, where the information that is negligible for our purposes has not been included. For instance, the topmost Span (id *16bb4e7b689f807a*) represents the first span in Fig. 4 with a *27ms* duration, of *SERVER* kind, and with the *November 20th 2018 10:52:48.107* timestamp for the call to the `http://categories/category` EndPoint belonging to the *gateway* Service. Such a model is automatically generated from the original raw log by means of a Java transformation able to serialize the textual representation of the logs into xmi-encoded models conforming to the Log Metamodel.

3.2. Design-runtime traceability generation

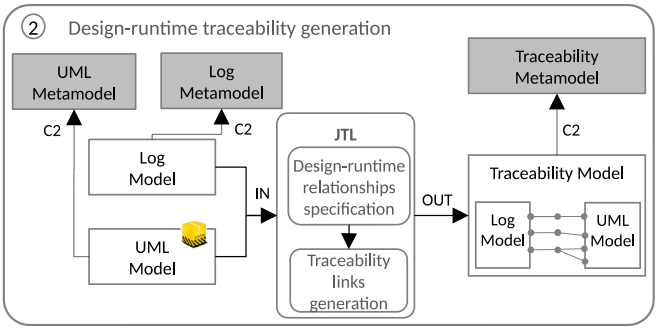


Fig. 6. Design-runtime traceability generation.

In this phase, as depicted in Fig. 6, the correspondences between the system behavior at runtime and the architectural design are defined and generated. In this work, we generate


```
traceId: 149c4cef3ac7f19f duration: 27,000 shared: true localEndpoint.serviceName: gateway localEndpoint.ipv4: 172.28.0.12 localEndpoint.port: 4000 timestamp_millis: November 20th 2018, 10:52:48.107 kind: SERVER name: http:/categories/category id: 16bb4e7b689f807a parentId: 149c4cef3ac7f19f timestamp: 1,542,707,568,107,000 tags.spring.instance_id: 002ffdb287d6:gateway:4000 _id: cE-JMGcBBzL8qQlHYHn4 _type: span _index: zipkin:span-2018-11-20 _score: -

traceId: 149c4cef3ac7f19f duration: 17,000 shared: true localEndpoint.serviceName: categories-server localEndpoint.ipv4: 172.28.0.18 localEndpoint.port: 5555 timestamp_millis: November 20th 2018, 10:52:48.115 kind: SERVER name: http:/categories/category id: 4ad2da86e8767b82 parentId: 16bb4e7b689f807a timestamp: 1,542,707,568,115,000 tags.mvc.controller.class: CategoriesController tags.mvc.controller.method: getCategory tags.spring.instance_id: 5b58aea6835e:categories-server:5555 _id: bk-JMGcBBzL8qQlHYHn2 _type: span _index: zipkin:span-2018-11-20 _score: -

traceId: 1b013fa75420bf54 duration: 6,000 shared: true localEndpoint.serviceName: gateway localEndpoint.ipv4: 172.28.0.12 localEndpoint.port: 4000 timestamp_millis: November 20th 2018, 10:52:48.976 kind: SERVER name: http:/categories/category id: 18c3f41109c8c6fd parentId: 1b013fa75420bf54 timestamp: 1,542,707,568,976,000 tags.spring.instance_id: 002ffdb287d6:gateway:4000 _id: oO-JMGcBBzL8qQlHZHkK _type: span _index: zipkin:span-2018-11-20 _score: -

traceId: cb9b9ab5d908bf18 duration: 6,000 shared: true localEndpoint.serviceName: gateway localEndpoint.ipv4: 172.28.0.12 localEndpoint.port: 4000 timestamp_millis: November 20th 2018, 10:52:50.500 kind: SERVER name: http:/categories/category id: 6850d3b3195db041 parentId: cb9b9ab5d908bf18 timestamp: 1,542,707,570,500,000 tags.spring.instance_id: 002ffdb287d6:gateway:4000 _id: 1k-JMGcBBzL8qQlHZ3n1 _type: span _index: zipkin:span-2018-11-20 _score: -

traceId: cb9b9ab5d908bf18 duration: 4,000 shared: true localEndpoint.serviceName: categories-server localEndpoint.ipv4: 172.28.0.18 localEndpoint.port: 5555 timestamp_millis: November 20th 2018, 10:52:50.501 kind: SERVER name: http:/categories/category id: 848f713a3fc3a108 parentId: 6850d3b3195db041 timestamp: 1,542,707,570,501,000 tags.mvc.controller.class: CategoriesController tags.mvc.controller.method: getCategory tags.spring.instance_id: 5b58aea6835e:categories-server:5555 _id: 1E-JMGcBBzL8qQlHZ3ny _type: span _index: zipkin:span-2018-11-20 _score: -
```


Fig. 4. A fragment of the raw log.

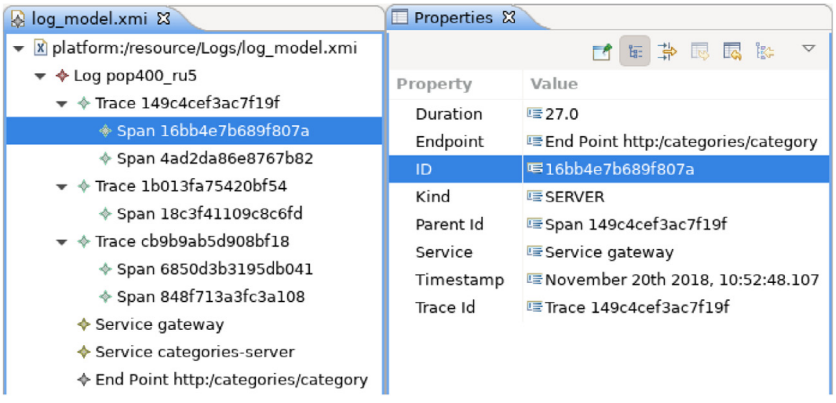


Fig. 5. A Log Model sample in Eclipse.

traceability links between UML and Log Models by means of JTL (introduced in Section 2.2.1). In particular, JTL supports the specification of *design-runtime relationships* in a declarative way at metamodel level, as bidirectional model transformations (i.e., between design and log metamodels). The JTL traceability engine is able to execute such bidirectional model transformations and automatically generate the corresponding traceability links between elements of the UML design model and the log model ones.

Traceability links are collected in an explicit way in Traceability Models conforming to a dedicated metamodel, namely the JTL *Traceability Metamodel*. As depicted in Fig. 7, it basically defines the notion of TraceModel, which is the root element of a traceability model. It relates a model belonging to a "left" domain to a model belonging to a "right" domain. In particular, a set of trace links between left and right elements, as long as the rules that enforced their mapping, are collected. A TraceLink relates one or more elements belonging to the left domain (leftLinkEnd) and the corresponding (one or more) elements belonging to the right domain (rightLinkEnd). Such links connect elements of TraceLinkEnd type that have a name and a type. Each TraceLinkEnd refers to an object of EObject type (org.eclipse.emf.ecore.EObject) that represents a specific object in the left or right domain.

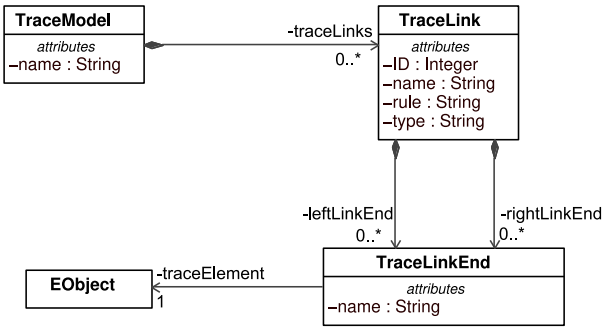


Fig. 7. JTL Traceability Metamodel.

In Listing 1 we show an excerpt of JTL-defined correspondences between the design and runtime concepts. While runtime concepts are represented by means of elements belonging to the Log metamodel (as explained in Section 3.1), software design concepts are represented by elements belonging to the UML metamodel. In particular, for behavioral aspects we target elements of Sequence Diagrams, whereas for static aspects we target Use Case, Component, and Deployment Diagrams.

The specification is defined by means of relations between elements of the two involved metamodels. UML Use Cases are related to monitoring Traces, since they represent executions of the system. UML Messages in Sequence Diagrams are related to monitored Spans, as both represent operations occurring in a scenario. In order to map an Operation invoked by a Message to a specific API EndPoint, we relate an EndPoint of a Span to a Signature of a Message. Finally, since microservices are modeled as UML components, we relate them to Services of Spans. The above described mappings can be specified in a declarative manner as correspondences in JTL, as described in the following.

In Line 1, variables log and uml are declared to match models conforming to the Log and UML metamodels, respectively. The specified relations are described as follows:

- The top relation Trace2UseCase (Lines 3–13) maps a container element of Trace type in the Log domain, and a container element of UseCase type in the UML domain. The where clause invokes the execution of the Span2Message relation;
- The Span2Message relation (Lines 15–23) maps a Span and a Message type elements involved in a use case interaction. The where clause invokes the execution of the EndPoint2Signature relation;
- The EndPoint2Signature relation (Lines 25–33) maps an EndPoint of a Span and an Operation type element that represents the signature of a message;
- The top relation Service2Component (Lines 35–43) maps a Service type container element to a Component type one.

```
1  transformation Log2UML (log:Log, uml:UML) {
2    ...
3    top relation Trace2UseCase {
4      checkonly domain log t : Log::Trace {
5        spans = s : Log::Span { }
6      };
7      checkonly domain uml uc : UML::UseCase {
8        ownedBehavior = ob : UML::Interaction {
9          message = m : UML::Message { }
10       }
11     };
12     where { Span2Message(s, m); }
13   }
14
15   relation Span2Message {
16     checkonly domain log s : Log::Span {
17       endpoint = ep : Log::EndPoint { }
18     };
19     checkonly domain uml m : UML::Message {
20       signature = s : UML::Operation { }
21     };
22     where { EndPoint2Signature(ep, s); }
23   }
24
25   relation EndPoint2Signature {
26     n : String;
27     checkonly domain log ep : Log::EndPoint {
28       name = n
29     };
30     checkonly domain uml s : UML::Operation {
31       name = n
32     };
33   }
34
35   top relation Service2Component {
36     n : String;
37     checkonly domain log s : Log::Service {
38       name = n
39     };
40     checkonly domain uml c : UML::Component {
41       name = n
42     };
43   }
44   ...
45 }
```

Listing 1: Log2UML correspondences specification

The described mapping assumes that the design of the system is consistent with its implementation (e.g., in terms of naming convention used). Moreover, the above described correspondences are specified according to the adopted notations. However, the approach can be extended to different modeling languages or monitoring technologies. In fact, JTL allows the specification of heterogeneous relations with different level of complexity, e.g., elements that do not trivially match by names, or relations between elements with one-to-many multiplicity (Cicchetti et al., 2010).

The application of the *Log2UML* transformation on a pair of Log and UML models, as shown in the left and right part of Fig. 8, generates the corresponding Traceability model in the middle part of the figure. In particular, the arrows in the figure cross trace links that connect the source and target model elements they refer to.

For instance, the *Trace2UseCase_149c4cef3ac7f19f* traceability link relates the *Get HomePage* use case in the right end and the corresponding *149c4cef3ac7f19f* log trace in the left end. Hence, for each message in the use case, we are able to know when the corresponding operation has started and its response time. As a consequence, the traceability model can be used to map complex

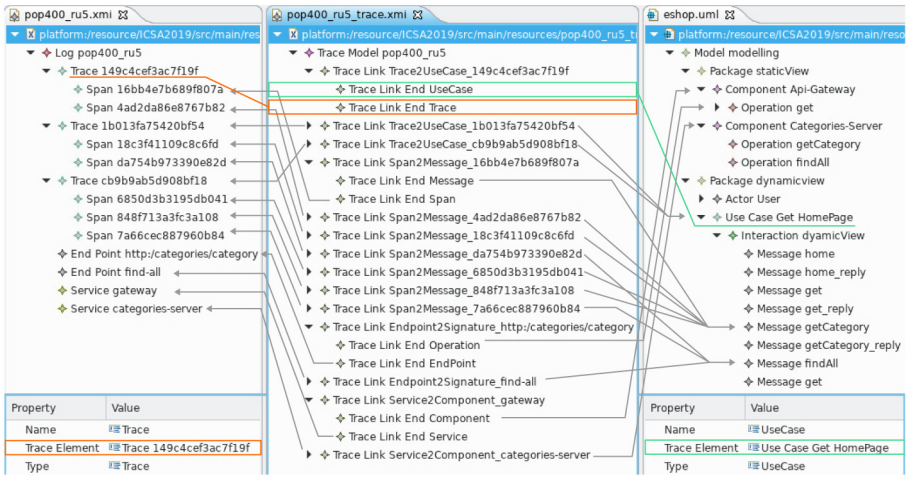


Fig. 8. An example of Traceability model between Log and UML models.

performance measures, such as the average response time of a specific scenario or the average service time of an operation. This process will be described in detail in the next section.

3.3. Model analysis and refactoring

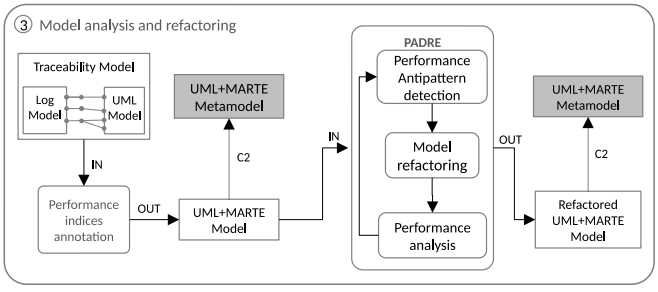


Fig. 9. Model analysis and refactoring.

This step is aimed at analyzing the design model and removing possible performance flaws. In particular, we use runtime data (i.e., traces) to augment the design model, and then we execute a performance analysis driven by antipatterns (Cortellessa et al., 2011) on the augmented model.

The *Performance indices annotator* in Fig. 9 exploits the Traceability Model to report runtime data back to the model, and for this goal it exploits the MARTE stereotypes, in that they allow: (i) to set input data (i.e., performance parameters) of performance analysis (e.g., operations service demand), and (ii) to fill the output data (i.e., performance indices) of a performance analysis back to the model (e.g., utilization).

We adopt the *MARTE:GAQM* package stereotypes, among MARTE ones, as follows:

- *Input* Data:
  - *GaWorkloadEvent:generator*: it expresses the generational value of a workload. For example, we annotate here the exponential arrival rate λ for an open class of jobs.
  - *GaWorkloadEvent:pattern*: it denotes if the job class is open or closed;
  - *GaAcqStep:servCount*: it expresses the service demand of a UML Operation. Hence, we annotate the UML Message that trigger that Operation in a UML Sequence Diagram to express its demand in that scenario;
- *Output* Data:
  - *GaScenario:respT*: it expresses a response time. We use it on a UML Use Case to report the response time of that scenario under a specific workload.
  - *GaScenario:throughput*: it expresses a throughput. We use it on a UML Use Case similarly to a response time;
  - *GaExecHost:utilization*: it expresses an utilization. We use it for a UML Node within a Deployment Diagram.

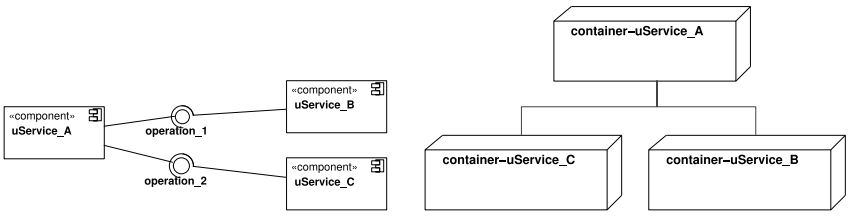
In order to collect operation service demands, we stimulate the system with a lightweight workload. Indeed, in this parameterization step we aim to avoid generating queues in the system, so that the *Service Demand* $D = V * S$ definition holds (Lazowska et al., 1984). In particular, D is the Service Demand, V is the number of visits, and S is the service time. Following the definition, the service demand of an operation, in our case, is given by its response time when the lightweight workload is executed, because we guarantee (by observation) that waiting time in queue is never originated by that workload.

Once *Service Demands* have been collected and filled back to the input *GaAcqStep:servCount* tags, we stimulate the system with a selected (possibly heavy and realistic) workload to discover performance flaws.

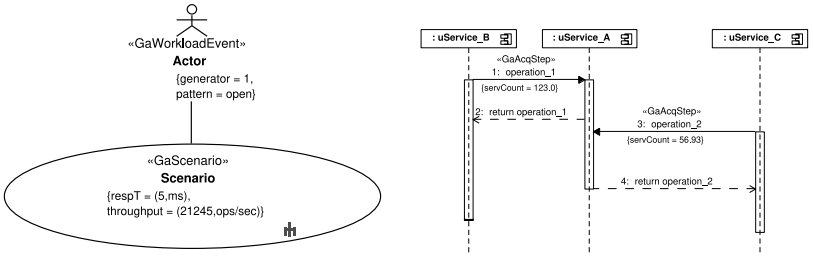
Upon the system execution is completed under the selected workload, the *Performance Indices Annotator* fills the performance indices tags back to the model. Then, a performance-driven refactoring loop can start. PADRE (Arcelli et al., 2018) has been adopted for this goal, as it is an approach that detects performance antipatterns and provides a list of possible refactoring actions that shall mitigate the performance degradation.

The PADRE refactoring loop is made of three main steps: (i) *Performance Antipattern Detection*; (ii) *Model Refactoring*; (iii) *Performance Analysis*. First, the *Performance Antipattern Detection* is executed in order to detect performance antipatterns occurrences. It is worth noticing that PADRE employs multi-views models to discover performance antipatterns. For this reason we use a multi-view UML design model, as depicted in Fig. 10. In case performance antipatterns arise in the model, a *Model Refactoring* step is performed in order to remove them. In this step, PADRE provides a list of possible refactoring actions for each performance antipattern.[12] While executing one refactoring action at a time, the refactored design model is given as

[12] The complete PADRE refactoring action portfolio is described in Arcelli et al. (2018)

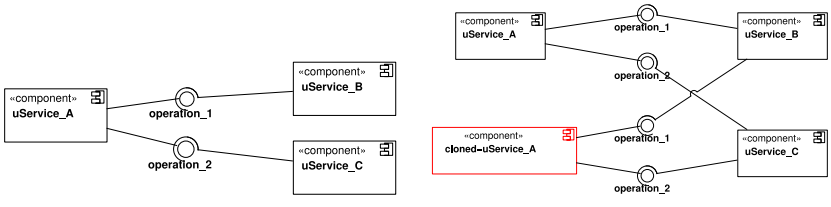


(a) Static view (Component Diagram) (b) Deployment view (Deployment Diagram)

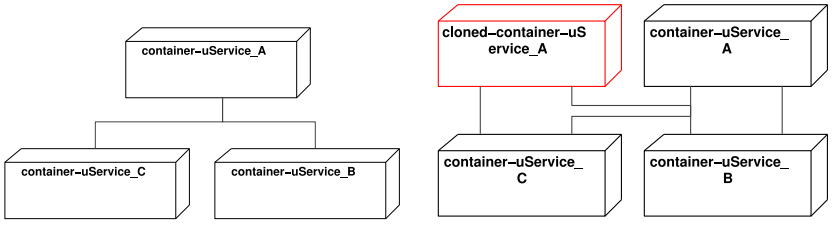


(c) Static view (Use Case Diagram) (d) Dynamic view (Sequence Diagram)

Fig. 10. The *example* UML Software Model.



(a) Initial (b) Refactored



(c) Initial (d) Refactored

Fig. 11. The *clone* refactoring action example on component *uService_A* through a UML Software Model.

input to the PADRE *Performance analysis* step, in which a model transformation is executed to transform the UML-MARTE design model into a closed Queueing Model (Cortellessa et al., 2011).[13] Thereafter, the Queueing Model is solved through the *Mean-Value Analysis (MVA)* algorithm (Reiser and Lavenberg, 1981), which allows to rapidly carry out performance indices. The latter ones are exploited to recognize whether the refactoring action is promising or not.

In this paper, we have restricted the PADRE refactoring actions portfolio to the actions that we found more appropriate for a microservice context, namely:

- Clone refactoring:
  The clone refactoring action is aimed at introducing a replica of a microservice. In our modeling assumptions, we consider a microservice as a *UML Component*, and a docker container as a *UML Node*. The action at a glance is shown in Fig. 11. Fig. 11(a) and 11(c) depict the initial design model, while Fig. 11(b), and 11(d) show the refactored design model. The clone refactoring action in a nutshell: (i) creates a new UML Component (i.e., *cloned-uService_A*), and (ii) creates a new UML Node (i.e., *cloned-container-uService_A*) on which the replica is deployed. In this particular case, the dynamic view is not depicted because the refactoring action does not affect it.
- Move operation refactoring:
  The move operation refactoring action is aimed at moving a "critical" operation (e.g., due to its Service Demand) to a new microservice. Figs. 12(a), 12(c) and 12(e) depict the initial

[13] This kind of transformation is out of scope of this paper, thus we do not provide here more details.

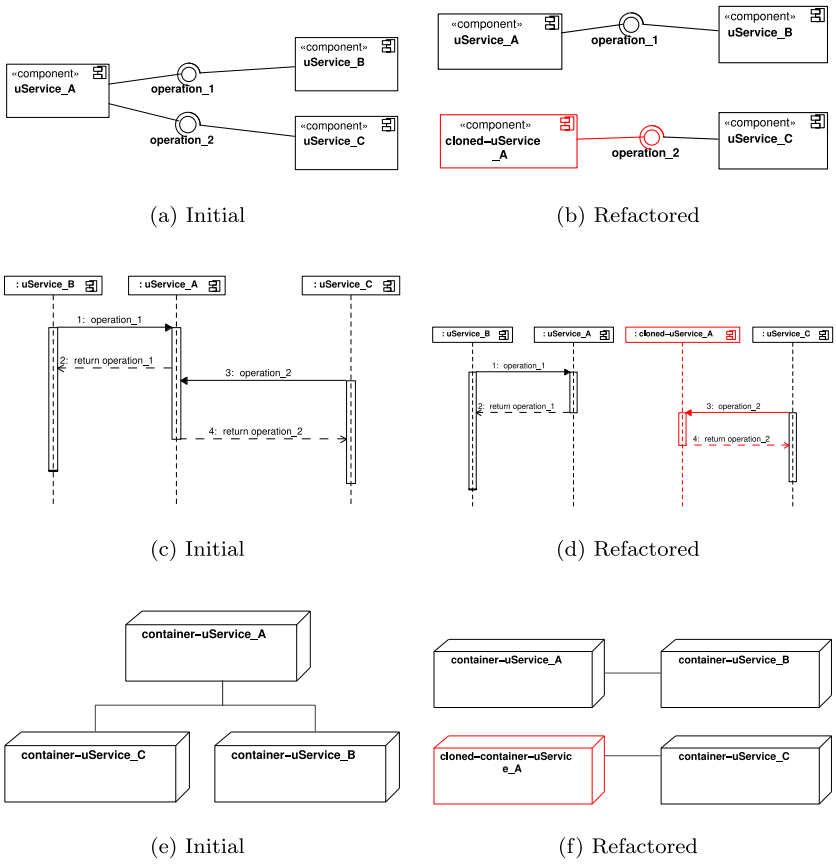


Fig. 12. The *move operation* refactoring action example on *operation_2* through a UML Software Model.

design model example, while Figs. 12(b), 12(d) and 12(f) show the refactored version. In the example, we move *operation_2* of *uService_A* microservice. Thus, the action creates a replica of this microservice (i.e., *cloned-uService_A*) and then it changes the behavior and deployment views, respectively.

Differently to the Clone refactoring action, the move operation involves the dynamic view. Therefore, a new UML Lifeline (i.e., *cloned-uService_A*) representing the replicated microservice is created (see Fig. 12(d)). Then, every message referring to operation_2 is now transferred towards the new lifeline. Furthermore, the Deployment Diagram is refactored as well. A new UML Node (i.e., *cloned-container-Service_A*) is created and, finally, the action defines the newly required connections among UML Nodes (i.e., the connection between the UML Node *cloned-container-uService_A* and *container-uService_C*). In particular, the new node is linked to all other nodes that were originally connected to the node on which the microservice hosting the "critical" operation is deployed.

3.4. System refactoring

In this step, as depicted in Fig. 13, the refactoring actions performed on the model are translated into changes of a microservice-based system.

Refactoring actions on the system have been implemented using the *Docker Client*[14] Java library for the operations performed on docker instances. Regarding the online modifications of configuration files, we developed a Java library that is publicly available.[15]

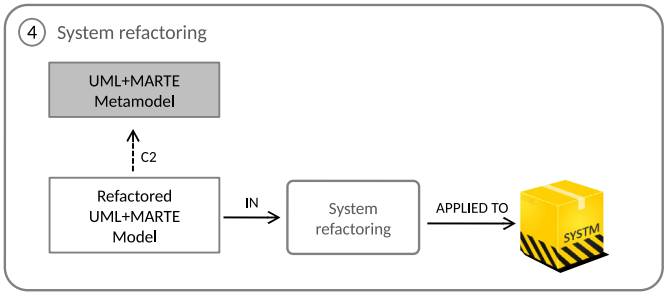


Fig. 13. System refactoring.

Clone refactoring. This action creates a replica of a microservice. In the running system, this translates into the creation of a new container deploying the same *Spring Boot* microservice we intend to clone. This refactoring is achieved by exploiting the Docker API to create and start a new container using the image of the original microservice. Once the replica is up and running, we need to balance it along with the original microservice. Specifically, we want to ensure that half of the traffic that was targeted at the original microservice is now redirected to its clone. Depending on the technology used to forward requests among microservices, three different scenarios are open:

- *Zuul and Eureka.* This scenario is straightforward because the combination of the *Zuul*,[16] proxy with the *Eureka*[17]

14 Docker Client: https://github.com/spotify/docker-client.
15 Microservices refactoring library: https://git.io/JLEJZ.
16 Zuul: https://github.com/Netflix/zuul.
17 Eureka: https://github.com/Netflix/eureka.

registration service automatically balances the additional microservice. When the new microservice registers to *Eureka Zuul* adds it to the physical locations available for requests forwarding. Internally, *Zuul* uses *Ribbon*[18] to balance incoming requests using a round-robin policy. Therefore, no further modifications are required in this case.
- *Nginx*. When *Nginx*[19] is used as a reverse proxy, we need to modify its configuration to add a new server group containing the original and the cloned microservices. Moreover, the mapping of requests has to be updated to address the requests to the newly created server group. By default, Nginx uses a round-robin algorithm to balance the servers in a server group.
- *No proxy*. Finally, when no proxy is deployed in front of the original microservice, we can add one without disrupting the running system. In this case, we preferred to deploy *HAproxy*,[20] because it is able to automatically generate a configuration by deriving the composition of server groups from the network links among docker instances.

Move operation refactoring. This action moves an operation from a service to a newly created one that has the purpose of exclusively offering the operation. This is implemented by creating a replica of the original service that contains the operation we want to move and, consequently, by forwarding all the requests that were intended for the moved operation to the replica. Similarly to the clone refactoring action, this action can be implemented in the same three scenarios:

- *Zuul and Eureka*. Once the replica has been created and has registered itself to *Eureka*, *Zuul* automatically detects it. In order to forward the requests for the moved operation to the replica, we need to add a new route in the *Zuul* configuration file. Such route is needed to map the path of the moved operation to the endpoint URL of the replica.
- *Nginx*. Analogously to the previous scenario, also when using *Nginx* as a reverse proxy, we need to add a route to redirect the requests to the moved operation. This is accomplished in *Nginx* by using the *location* directive to map a path to an endpoint URL.
- No proxy. In this case, a new proxy is added to the application. Both *HAproxy* and *Nginx* can serve this purpose with similar configurations for redirecting URL paths.

4. Evaluation

In this section, we discuss the evaluation we have performed with the aim of answering the following research questions:

- *RQ1: Do the proposed model refactoring actions improve the performance of the running system?*
- *RQ2: To what extent does performance antipattern (PA) removal improve the whole performance?*

4.1. Case studies setup

In order to validate the approach, we considered the following case studies:

- *E-Shopper*[21] is an e-commerce web application. The application is developed as a suite of small services, each running in its own Docker container and communicating with RESTful HTTP API. E-Shopper is composed by 9 application microservices developed with the Spring framework, each requiring a different database to operate.
- *TrainTicket*[22] is a web ticketing application within the railway domain. It has been developed by Zhou et al. and it has been also presented in Zhou et al. (2018a,b, 2019). It is made up of 40 microservices and uses different programming languages. The most used framework is again Spring, since the most used programming language in Train Ticket is Java, and for this reason we have selected it for our evaluation. In our previous work we reverse engineered its UML representation and presented it as a reference case study (Di Pompeo et al., 2019), while here we employ it to test our performance improvement approach.

The UML models (OMG, 2015) of both case studies are augmented with the MARTE profile stereotypes (OMG, 2008). We adopt MARTE because it is the standard profile widely adopted to annotate UML models with real-time and performance attributes, such as *response time values*. In particular, we adopt the *Generic Quantitative Analysis Model (GQAM)* package, which has been introduced to support accurate and trustworthy evaluations based on formal quantitative analyses.

We have generated different scenarios for each application to stimulate different parts of the system and discover which ones may suffer from performance degradation under concurrent usage.

In particular, we have identified the scenarios that are more typically triggered in these applications, thus to validate our approach in performance-critical contexts. We have observed these scenarios running under different workloads, and then we have picked the workloads that stress the application performance, but that at the same time do not lead to fully system saturation, because our approach is intended to work before extreme degradation of performance occur.

For the E-Shopper application we have realized three scenarios with specific workloads:
- *Desktop*, the request of the homepage, with a workload of 3.8 user/sec;
- *Mobile*, the request of a specific service through an API call (e.g., it maps a call from the E-Shopper mobile app), with a workload of 225 user/sec;
- *Warehouse*, the request from a warehouse worker to set and control the availability of items, with a workload of 17.5 user/sec.

For the Train Ticket case study, we have realized two scenarios with respective workloads:
- *Rebook Ticket*, the scenario on which a customer can change a ticket reservation, with a workload of 4.5 user/sec;
- *Update User*, the scenario on which the admin changes user information, with a workload of 2.75 user/sec.

It is worth noticing that we stimulated both the applications with workloads heavy enough to stress them, but not too heavy to originate errors and timeouts. In order to achieve steady-state performance, for each case study we executed a 20 min initial warm-up with additional 10 min warm-up after the application of each refactoring action. Intermediate warm-ups were necessary because the applications were never restarted to perform the refactoring actions, thus to simulate a production environment. Response time and utilization were continuously measured for 10 min after (the warmup following) every refactoring action. Finally, all tests were repeated three times in order to avoid circumstantial external influence.

18 Ribbon: https://github.com/Netflix/ribbon.
19 Nginx: https://nginx.org/.
20 HAproxy: http://www.haproxy.org/.
21 E-Shopper: https://github.com/SEALABQualityGroup/E-Shopper.
22 Train Ticket: https://github.com/SEALABQualityGroup/train-ticket.

All the performance measurements are performed on a server with dual Intel Xeon CPU E5-2650 v3 at 2.30 GHz, for a total of 40 cores and 80GB of RAM.

Performance measurements data as well as all the models resulting from this validation are available online.[23]

4.2. RQ1: Performance improvement

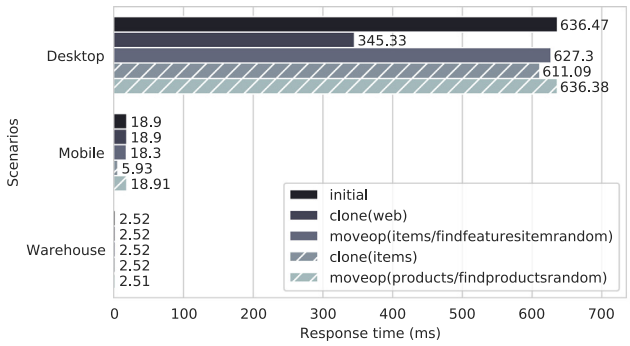
In order to answer RQ1, we evaluate if the approach is able to produce refactoring decisions that improve the performance metrics computed on the software models, and consequently the performance of the running system. To this end, we show how the approach applies refactoring actions that are promising on the basis of the model, and how these actions are propagated to the running system, in the context of the considered case studies. Moreover, we compare refactoring actions that were induced by performance antipatterns against other actions that we randomly perform on the model and on the running system. In this way, we are able to show that: (i) our approach can select refactoring actions which improve the performance of the running system, and (ii) basing the selection of such actions on the combination of QN analysis and performance antipatterns detection is more effective than randomly refactoring the system.

We start the assessment of the refactoring impact on the model by comparing utilization and response time before and after the modifications. Figs. 14 and 15 show a comparison of response times and utilization computed on the QN for both case studies. Utilization is reported only for relevant microservices, that are those involved in at least one scenario and for which the utilization varies among refactoring actions.

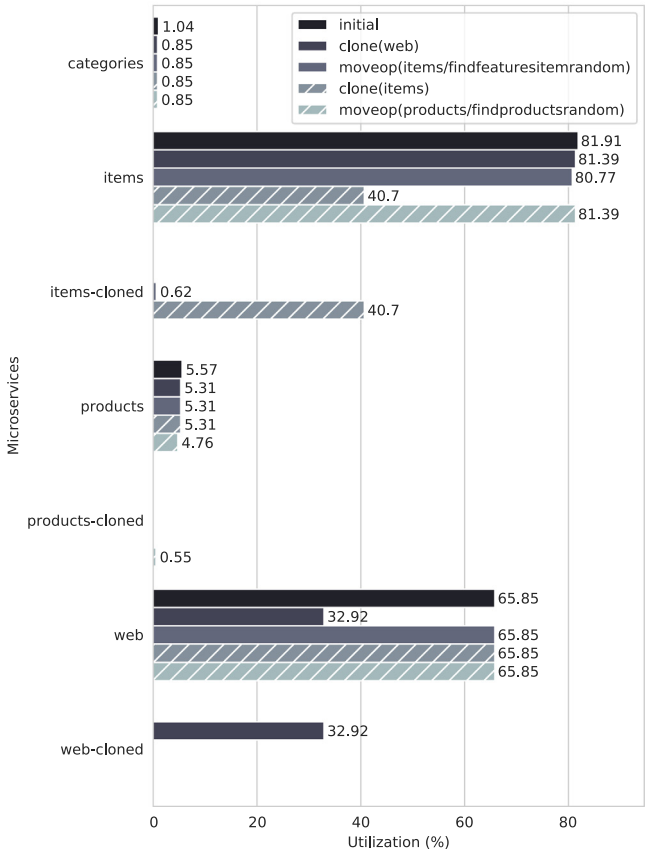
In the E-Shopper case study, starting from the initial system, our approach proposed two alternative refactoring actions to improve the *Desktop* scenario: cloning the *web* microservice (*clone(web)*), and moving the *findfeaturesitemrandom* operation from the *items* microservice to a newly created one (*moveop(items/findfeaturesitemrandom)*). Among the other feasible actions, we randomly selected cloning the *items* microservice (*clone(items)*) and moving the *findproductsrandom* operation from the *products* microservice to a newly created one. These randomly selected actions are marked with a pattern in the histograms of Fig. 14. We can see how cloning the *web* microservice is remarkably beneficial for the response time of *Desktop* scenario, while the randomly selected action of the same type, that is cloning the *items* microservice, has a negligible impact on the same scenario and a small impact on the *Mobile* one. When comparing the actions that move an operation to a new microservice, we can notice a difference in response time, even if small, in favor of the refactoring targeting the operation *findfeaturesitemrandom*.

In order to improve the *Rebook Ticket* scenario of the Train Ticket case study, our approach proposed to clone the *rebook* microservice (*clone(rebook)*), or alternatively to move the operation *generate* from the *verification-code* microservice to a newly created one (*moveop(verification-code/generate)*). The randomly selected actions are: cloning the *admin-user* microservice (*clone(admin-user)*), and moving the *login* operation from the *sso* microservice to a new one. Also in this case, the actions selected on the basis of performance antipatterns induced a larger improvement in the response times of the targeted scenario, as we can notice from Fig. 15.

As introduced before, we are also interested in evaluating if the application of the refactoring actions on the running system improves its software performance by comparing utilizations and response times before and after the modifications. Figs. 16 and 17



(a) Comparison of the effect of refactoring on response times.



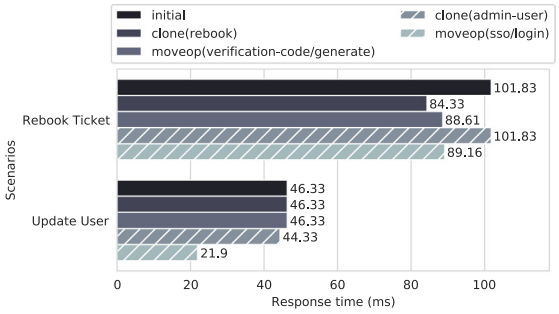
(b) Comparison of the effect of refactoring on the utilization of microservices.

Fig. 14. Average response times and utilizations computed on the QN for the E-Shopper case study.

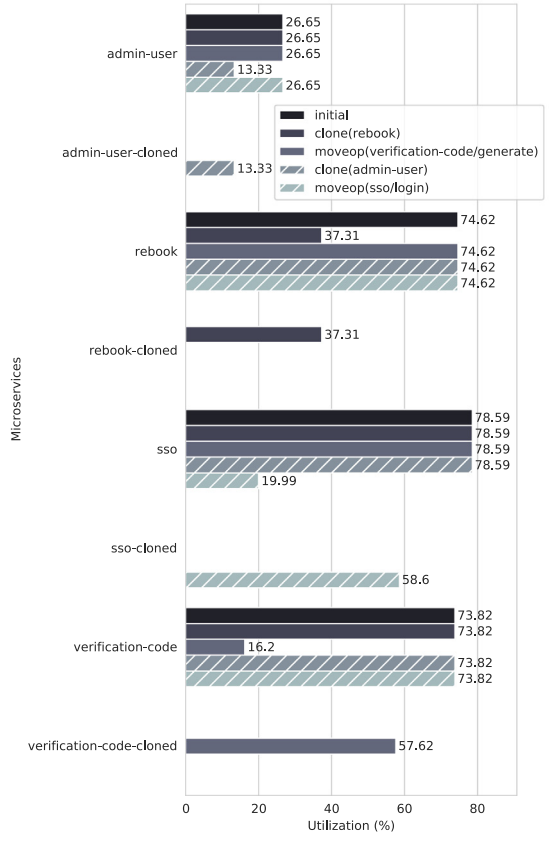
show the measures obtained by monitoring both case studies as described in Section 2.1.

The results show that, in both case studies, the refactoring actions that were selected on the basis of antipatterns are more effective in improving the response times of the targeted scenarios. Furthermore, there are some interesting aspects to notice. For instance, by just looking at the utilizations computed on the QN for the E-Shopper case study (Fig. 14(b)), a performance analyst would have probably guessed that cloning the *items* microservice would be the best action to perform. Instead, such refactoring only marginally decreases the response time of the *Desktop* scenario on the QN and on the system. In this case, cloning the

[23] Replication package: https://zenodo.org/record/4756322.

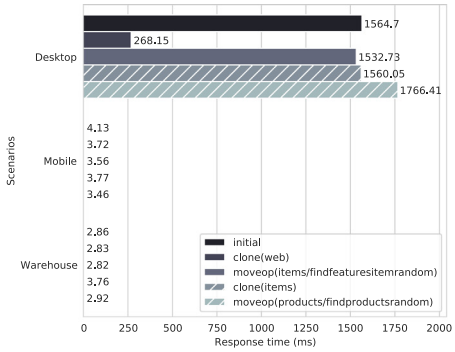


(a) Comparison of the response times of scenarios.

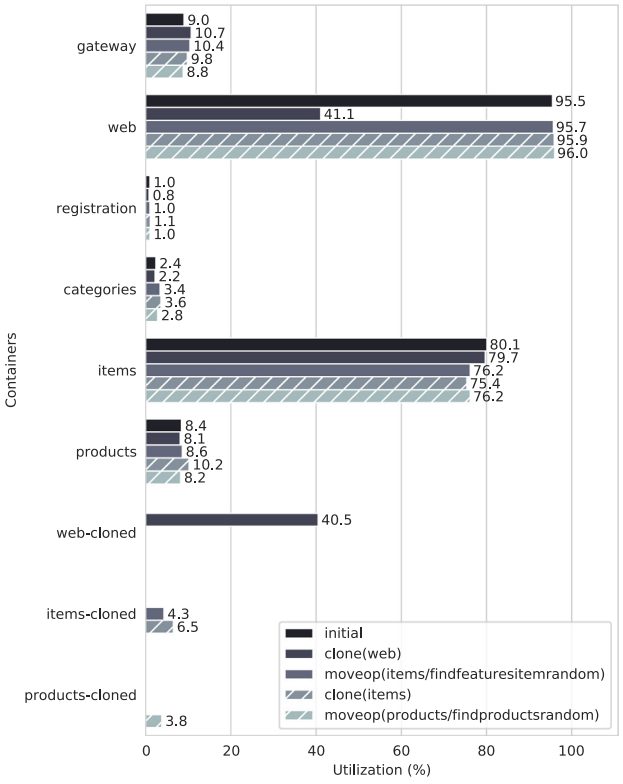


(b) Comparison of the utilization of microservices.

Fig. 15. Average response times and utilizations computed on the QN for the Train Ticket case study.



(a) Comparison of the effect of refactoring on response times.



(b) Comparison of the effect of refactoring on the utilizations of microservices.

Fig. 16. Average response times and utilizations measured on the running system for the E-Shopper case study.

web microservices was far more effective as also shown by the measures obtained from the running system.

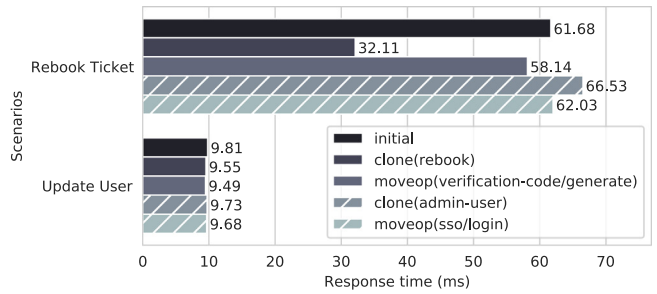
We can notice an even more extreme situation in the Train Ticket case. Both the actions that were randomly selected actually increased the response time of the *Rebook Ticket* scenario on the running system (Fig. 17(a)). Even if, in general, the cloning and moving operation refactoring actions are designed to produce a performance improvement, when applied without taking into consideration the design of the application may indeed result in degrading the performance.

These are just some examples of how the combination of design and runtime knowledge can induce a more thorough selection of the convenient refactoring to perform. More generally, design-runtime traceability provides additional knowledge that can be automatically maintained while supporting design decisions, also when the software is running in production.

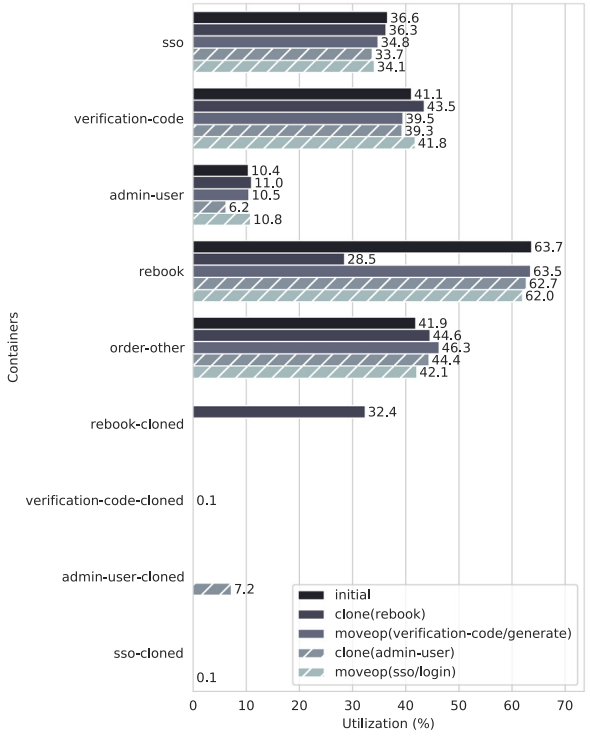
4.3. RQ2: Performance antipatterns

In order to answer RQ2, we discuss the benefits of employing detection and refactoring of PAs to performance improvement forecasting. We consider here performance antipatterns that can suitably fit with microservice-based systems, namely Blob and PaF (Pipe and Filter) (Smith and Williams, 2002).

As described in Section 2.3, literals of a performance antipattern first-order logic representation are compared to thresholds. The definition of fixed values for these thresholds is an



(a) Comparison of the response times of scenarios.



(b) Comparison of the utilizations of microservices.

Fig. 17. Average response times and utilizations measured on the running system for the Train Ticket case study.

application-dependent task that can become very complex in some cases. Therefore, we employ in this paper the concept of fuzzy thresholds (Arcelli et al., 2015). Instead of deterministically identifying the occurrence of a performance antipattern, fuzziness in thresholds induces a probability for an antipattern to occur, as the combination of probabilities of threshold violations. The probability for an element x to violate a fuzzy threshold Th_k on k metric is defined as follows:

$$P_k(x) = 1 - \frac{UBTh_k - F_k(x)}{UBTh_k - LBTh_k} \tag{3}$$

This expression considers $UPTh_k$ and $LBTh_k$ as the upper and lower bounds of Th_k threshold, respectively, and $F_k(x)$ as the value that the k metric assumes in the x element.

Blob. Each inequality of Expression (1), described in Section 2.3, undergoes a fuzzy evaluation like the one in Expression (3). Hence, the probability of a Blob occurrence is obtained as follows:

$$P(Blob) = P_{numClientConnects}(C_x) * P_{numMsgs}(C_x, C_y) * P_{maxHwUtil}(P_{xy}) \tag{4}$$

Table 1
Probability of *Web* being a Blob. M_0 is the initial model, M_1 is the model refactored through a *Clone* refactoring action on *Web*, and M_2 is the model refactored through a *Clone* refactoring action on *Items*.

Performance Antipattern Literal	M_0	M_1	M_2
$P_{numClientConnects}(Web)$	1	1	1
$P_{numMsgs}(Web)$	1	1	1
$P_{maxHwUtil}(Container - Web)$	.80	.39	1
$P_{Blob}(Web)$	.80	.39	1

Pipe and filter (paf). Each inequality of Expression (2), described in Section 2.3, also undergoes a fuzzy evaluation like the one in Expression (3). Hence, the probability of a PaF occurrence is obtained as follows:

$$P(PaF) = P_{resDemand}(Op) * P_{maxHwUtil}(P) \tag{5}$$

We remark that the second literal in Expression (2) must be always equal to one to trigger a PaF, thus it can be omitted in Expression (5) that quantifies the PaF occurrence probability. □

In the following, Tables 1–5 describe the effects on performance antipattern probabilities of refactoring actions that are either randomly selected or driven by performance antipattern detection. In each table, the M_0 column lists the initial probability, i.e., the value obtained on the initial configuration, the M_1 column lists the probability obtained after applying the action suggested by PADRE to remove the antipattern, and the M_2 column lists the probability obtained after applying a random refactoring action. Each row represents a literal of the performance antipattern expression, while the last row reports the occurrence probability of the whole antipattern.

4.3.1. E-Shopper

Tables 1 and 2 describe the probability of *Web* and *Items* being Blob performance antipatterns, respectively, in the E-Shopper case study. In particular, column M_1 of Table 1 corresponds to the *Clone* refactoring action on *Web* microservice as suggested by PADRE, while column M_2 corresponds to the *Clone* refactoring action on *Items* as a randomly selected recfactoring action. It is noteworthy that the probability drops from 0.80 to 0.39 after the refactoring action suggested by PADRE, while the probability grows to 1 after the random action of *Items* cloning.

We recall that the application of the same refactoring actions on the running system, as shown in Section 4.2, in case of cloning *Web* reduces the utilization of that microservices by 50.2%, whereas cloning *Items* does not change the utilization of the Web microservice.

Although the initial probability of *Items* being a *Blob* is lower than in the *Web* case, the *Move Operation* on *items/findfeature-sitemrandom*, suggested by PADRE, reduces the initial probability (M_0) from 0.25 to 0.06, while the application of the *Move Operation* on the randomly selected *products/findproductsrandom* does not change the probability at all.

We recall that in the running system, as shown in Section 4.2, the effect of the PADRE suggestion is negligible in terms of performance, as the utilization of *Items* is reduced by 0.4% and the response time (see Fig. 16) is quite the same. However, the effect of the random action in terms of performance is twofold: the utilization of *Items* is decreased by 4%, and the utilization of *Web* is increased, albeit it is close to being saturated.

4.3.2. Train ticket

We report in Tables 3–5 performance antipattern probabilities due to different refactoring actions applied to the Train Ticket case study.

Table 3 reports the probability of *verification-code/generate* operation being a Pipe and Filter (PaF). In particular, column

Table 2
Probability of *Items* being a Blob. M_0 is the initial model, M_1 is the model refactored through a *Move Operation* refactoring action on *items/findfeaturesitem-random*, and M_2 is the model refactored through a *Move Operation* refactoring action on *products/findproductsrandom*.

Performance Antipattern Literal	M_0	M_1	M_2
$P_{numClientConnects}(Items)$	.5	.25	.5
$P_{numMsgs}(Items)$	.5	.25	.5
$P_{maxHwUtil}(Container - Items)$	1	1	1
$P_{Blob}(Items)$	.25	.06	.25

Table 3
Probability of *verification-code/generate* being a Pipe and Filter (PaF). M_0 is the initial model, M_1 is the model refactored through a *Move operation* refactoring action on *verification-code/generate*, and M_2 is the model refactored through a *Move operation* refactoring action on *sso/login*.

Performance Antipattern Literal	M_0	M_1	M_2
$P_{resDemand}(verification - code/generate)$	.88	.88	.88
$P_{maxHwUtil}(Container - Verification)$	.91	0	.98
$P_{PaF}(verification - code/generate)$	.80	0	.87

Table 4
Probability of *rebook* being a Blob. M_0 is the initial model, M_1 is the model refactored through a *Clone* refactoring action on *rebook*, and M_2 is the model refactored through a *Clone* refactoring action on *admin-user*.

Performance Antipattern Literal	M_0	M_1	M_2
$P_{numClientConnects}(rebook)$	.5	.5	.5
$P_{numMsgs}(rebook)$	1	1	1
$P_{maxHwUtil}(Container - Rebook)$	.92	.24	.93
$P_{Blob}(rebook)$	.46	0.12	.46

M_1 refers to a *Move operation* refactoring action on *verification-code/generate* suggested by PADRE, while column M_2 refers to a randomly chosen *Move Operation* refactoring action effects on *sso/login* operation. We notice that the PaF probability decreases from 0.80 to 0 by applying the refactoring action suggested by PADRE. If we look at the running code (see Fig. 17), this refactoring action decreases the utilization of *Container-Verification* microservice as well as the response time of the *Rebook Ticket* scenario. The random action, instead, increases the probability of *verification-code/generate* being a PaF to 0.87, and if we look at the running system, this refactoring action leads to increase both the utilization of *Container-Verification* and the response time of *Rebook Ticket* scenario.

Table 4 reports the probability of *rebook* being a Blob. In particular, column M_1 lists the probabilities related to the *Clone* refactoring action on *rebook*, as PADRE suggests, while the column M_2 lists the probabilities related to the same refactoring action on *admin-user*. We notice that probability decreases from 0.46 to 0.12 by applying the refactoring action suggested by PADRE, while it remains unchanged after the randomly chosen action. If we look at the running code (see Fig. 17), the effect of the random action of cloning the *admin-user* microservice on the utilization is negligible, but it increases the response time of the *Rebook Ticket* scenario by 7%. Instead, the effect of the refactoring action suggested by PADRE on system performance is twofold: (i) the response time of the *Rebook Ticket* scenario decreases by 47.9%, and (ii) the utilization of the *Container-Rebook* microservice decreases by 55%.

Table 5 reports the probability of *verification* being a Blob. In particular, column M_1 lists the probabilities of the *Move Operation* refactoring action on *verification-code/generate*, as PADRE suggests, while column M_2 reports the probabilities of the same refactoring action on *sso/login*. We notice that the probability decreases from 0.46 to 0 by applying the refactoring action suggested by PADRE, which also reduces the *Verification.code* utilization and the response time of the *Rebook Ticket* scenario (as shown in Fig. 17). The application of the random action instead increases the probability of *verification-code* being the Blob to 0.98.

Table 5
Probability of *verification-code* being a Blob. M_0 is the initial model, M_1 is the model refactored through a *Move Operation* refactoring action on *verification-code/generate*, and M_2 is the model refactored through a *Move Operation* refactoring action on *sso/login*.

Performance Antipattern Literal	M_0	M_1	M_2
$P_{numClientConnects}(verification - code)$	.5	0	1
$P_{numMsgs}(verification - code)$	1	0	1
$P_{maxHwUtil}(Container - Verification)$	.91	0	.98
$P_{Blob}(verification - code)$	.45	0	.98

4.4. Summarizing discussion

On the basis of the results obtained, we can state that the removal of performance antipatterns leads to a performance improvement of a running system. On the other hand, ignoring the performance antipatterns knowledge induces either unchanged performance, in the best case, or performance detriment in all other cases.

Indeed, we experience on the Train Ticket case study a performance improvement in terms of lower hardware utilization from 0.411 to 0.395 (i.e., by about 4%) when the *verification-code/generate* operation is moved, and the response time of the *Rebook Ticket* scenario is reduced by 5% as well. It is noteworthy that the same refactoring action also removes the "Pipe and Filter" performance antipattern (as shown in Table 3).

Also, we experience on the E-Shopper case study a lower hardware utilization from 0.955 to 0.411 (i.e., by about 57%) when the *Container-Web* is cloned (as suggested by PADRE), and the response time of the *Desktop* scenario is reduced by about 83% (i.e., from 1,564 ms to 268 ms as shown in Fig. 16), and the probability of Web being a "Blob" performance antipattern decreases by 39%, as shown in Table 1.

We have also noticed that the application of a random refactoring action induces either unchanged performance or performance detriment. For example, the Train Ticket case study has shown a higher utilization from 0.411 to 0.418 (i.e., an increment by 1.7%), while the response time of the *Rebook Ticket* scenario remained unchanged. The probability of *verification-code/generate* being a "Pipe and Filter" performance antipattern is increased by 11% (as shown in Table 3. Instead, the random action on the E-Shopper case study has caused no changes both for the utilization, the response time, and the probability of *Web* being a "Blob" performance antipattern (as shown in Table 1).

5. Threats to validity

In this section, potential threats to validity associated with the validation are discussed.

Conclusion validity concerns the reliability of the measures. As explained in Section 4.1, we properly and rigorously designed our case studies setup. We attempted to avoid any bias by: (i) generating different scenarios with respective workloads to simulate different parts of the systems; (ii) stressing the execution of the applications to check performance under realistic conditions; (iii) repeating the measurements several times in order to avoid circumstantial external influences. We cannot assert how realistic the selected scenarios are, because these applications have been mostly used in literature for sake of validation in controlled environments. For example, in Grohmann et al. (2021) Train Ticket has been used by exposing numerous services to

a base load that varies between 3 and 22 requests per second, and the workload that we have adopted in this paper for the same application falls within this range. However, all the adopted application settings are among the ones that lead the considered applications on performance boundary situations that enable to validate the effectiveness of the approach proposed here.

Furthermore, during the evaluation, we ensured that the observed performance improvement was actually induced by the refactoring actions selected by our approach. To this end, we compared our solutions with performance variations caused by randomly selected refactoring actions.

Internal validity concerns any extraneous factor that could influence our results. In general, the implementation of the approach could be defective, as well as the results of the analysis could be inaccurate. In order to avoid any bias: (i) we have assured that the implementation was aligned with the software design to generate accurate traceability models and consistently annotate the performance models; (ii) we completely delegated the analysis of the considered performance model to an external consolidated tool; (iii) we employed technologies that are widely tested in production (e.g., Zuul, Eureka, Nginx) that provide unified interfaces to allow refactoring of microservices without exposing the internal structure. Furthermore, we provided a detailed discussion of the code instrumentation and made publicly available the source code in order to allow other researchers to reproduce and inspect our results.

Construct validity concerns any factor that can compromise the validity of the measurements and the resulting observations. Evaluation results are highly dependent on the quality of the considered measures. In the evaluation, we apply monitored data observed from running system and performance measures. As described above, we properly designed our case studies setup to avoid influence factors. For instance, we disabled as many system services (Linux *systemd* units) as possible to lessen the effects of context switching. To avoid interferences that may be caused by internal errors, we also monitored the machine by looking at system logs (*dmesg* buffer and *systemd* journal) for unexpected entries. Another threat may be posed by missing links in the generated traceability models. This may be caused by errors in the generation of Log models from monitoring traces, or by incorrectly matching the patterns defined in the correspondences specification, either for UML or Log models. To avoid this, we performed several manual assessments of the generated traceability models by looking for missing links or links connecting the wrong elements. Finally, we have annotated the performance indices in the UML-MARTE models in a consistent way with the consolidated literature in the field of software performance.

External validity refers to the generalizability of the obtained results. Case studies may be selected to facilitate a deeper understanding of the approach and this could affect their representativeness. In order to mitigate this, we selected two microservice-based web application benchmarks that have been successfully used in previous work for their size and heterogeneity (Arcelli et al., 2019; Di Pompeo et al., 2019). This choice also provides indicators for cases having similar properties. With reference to the used languages and tools, we adopted specific development and monitoring technologies (i.e., based on the Spring framework), as well as the specific modeling standards like UML and MARTE. These technologies are widely used both in academia and industry, thus supporting the approach to be generalizable and easily reproduced in other contexts.

Adopting specific tools (JTL and PADRE) could threat the generalizability of the approach. In this merit, very few other alternative tools are available for replacing JTL, and actually none for replacing PADRE to detect performance antipatterns in UML models. However, both tools have proven to be suitable for the specific purposes of the approach and to be used in heterogeneous contexts.

6. Related work

In this section we discuss existing work that proposes approaches for architectural-based improvement of software systems driven by the observation of monitoring data and/or their interleaving with the software design modeling. Researchers from several areas (e.g., self-adaptive systems, software engineering and continuous system engineering) have actively studied a wide variety of methods and techniques applicable at design time and/or at runtime. Hereafter, we focus on approaches dedicated to software performance and on approaches that make use of software models in the domain of microservices.

6.1. Software performance engineering approaches

A vast literature exists on performance modeling, performance monitoring, and performance problem identification techniques, as quite separate research domains. We report on the most significant papers that attempt at merging such domains.

Trubiani et al. (2018) have provided a systematic process to identify performance issues from runtime data, based on load testing coming from operational profile and application profiling. In particular, from runtime data, performance antipatterns are detected, aimed at identifying common performance issues and their solutions. Software refactoring is then (manually) applied to solve identified performance antipatterns. Apart from technological and implementation aspects, a methodological difference distinguishes our approach from the one in Trubiani et al. (2018), namely: we bring runtime data up to the design level, by annotating a UML model with the MARTE profile, and one of the main advantage of addressing performance issues at design level, among other, is to narrow down the search space for potential actions that can beneficially affect system performance.

Porter et al. (2016) have proposed the DeSARM approach, whose scope is the derivation of architectural models at runtime. Such models can be used in decentralized decision-making for architecture-based adaptation in large distributed systems. To this aim, DeSARM is able to identify important architectural characteristics of a running application, such as components, connectors, nodes and communication patterns. DeSARM has been used by Gomaa and Albassam (2017) to introduce runtime failure analysis and architectural recovery on the discovered system architecture. However, DeSARM has not been adopted for identifying performance problems, as we do in this paper.

Altamimi et al. (2016) deal with the automated generation of performance models from UML-MARTE architectural models, and the propagation of performance analysis results back to the latter. The main difference with our work is that this approach fully works at the model level, and it does not consider the availability of a running software system from which runtime information can be extracted for a more accurate identification of performance problems.

Logs obtained from monitoring a running software system are exploited by Vögele et al. (2018) to automatically extract workload specifications for load testing and performance models parameterization. However, Vögele et al. (2018) is limited to the parameterization of performance models, whilst our approach provides support for the interpretation of analysis results carried out by performance antipattern detection and their possible solutions.

Mazkatli et al. (2020) have presented an approach for continuous integration of performance models that considers parametric dependencies after analyzing the source code changes. The approach builds upon the Palladio approach and the goal is to automatically keep the performance models up-to-date to allow architecture-based performance prediction. In contrast with this

approach, we have provided an approach to identify model-based design alternatives to overcome detected performance problems and to apply them on the system.

Recently, Heinrich (2020) has proposed an approach to align architectural models used in development and operation by means of a correspondence model between implementation artifacts and component-based Palladio architectural models. In contrast with this work, our approach uses UML that is a widely adopted standard respect to the Palladio Component Model. They use an ad-hoc correspondence model that is then exploited by a complex pipeline of transformations, while we propose a general approach where the correspondences are generated from a declarative specification easily adaptable to other contexts. They use a specific infrastructure monitoring, while we combine application instrumentation and infrastructure monitoring. Also the scope is different, as their contribution is to assess software performance with the aim to support design decisions, whereas our approach is not limited to build design-runtime correspondences and predict performance issues. We indeed enable the identification of design alternatives and the implementation of system refactoring actions to improve performance. On the other hand, in contrast with our approach, workload characterization and model structure updates are proposed in Heinrich (2020), where the model is updated online and scalability and accuracy are validated. Finally, it is not limited to microservices domain like our approach does.

6.2. Model-based approaches for microservices

In self-adaptive systems, software models have been mostly used at development and specification time, and a few works considered software models for microservice application adaptations. In this respect, Rademacher et al. (2017) surveyed the use of models in microservice and service-oriented architectures. Also, Derakhshanmanesh and Grieger (2016) provided a vision and future challenges on the use of domain-specific modeling languages and model transformations across the full software lifecycle (including runtime) to define and evolve a microservice application at the architectural level.

Weyns (2019) described relevant aspects and future challenges in the field of software engineering for self-adaptive systems. In particular, the author puts the concrete realization of runtime adaptation mechanisms that leverage software models at runtime to reason about the system and its goals. The use of models at runtime (known as models@run.time) (Blair et al., 2009; Bencomo et al., 2019) has been proposed to extend the applicability of software models produced in MDE approaches to the runtime environment. Such models should represent the system and its current/updated state and behavior. The envisioned goal is to support adaptive systems, e.g., to drive subsequent adaptation decisions, to fix design errors or to explore new design decisions. As an alternative to models at runtime, we used traceability models to represent runtime information and its relation with design models. Such solutions allow us to exploit existing monitoring infrastructure and existing design models throughout all phases of the approach.

Düllmann and van Hoorn (2017) proposed a preliminary framework to generate microservice environments that can then be used for measurement-based evaluation of performance and resilience. The approach allows developers to create models and generate Java code and deployment files. The generated microservices are automatically instrumented to collect metrics at runtime. In contrast with our work, the proposed setup was developed and used as benchmarking environments for their evaluation of approaches for performance and resilience. Although the framework is able to generate microservice environments with specified properties, it has not been adopted for the adaptation of microservice-based systems, as we do in this paper.

Zúñiga-Prieto et al. (2017) proposed an incremental and model-driven approach that supports the integration of cloud service applications and their dynamic architecture reconfiguration. Models are used to generate skeletons of microservices, integration logic, and also scripts to automatically deploy and integrate the microservices in the specific cloud environment where a microservice will be deployed. The approach supports the integration of increments composed of several microservices, whereas it has not been adopted for the improvement of existing services; also, runtime information is not considered.

Sampaio et al. (2019) proposed a platform-independent runtime adaptation mechanism to reconfigure the placement of microservices based on their communication affinities and resources usage. The authors propose to identify the runtime aspects of microservice execution that impact the placement of microservices by using models at runtime. In contrast with our approach, the main contribution is limited to the reconfiguring mechanism to manage the placement of microservices.

7. Conclusion

In this paper we have extended a previously introduced approach (Arcelli et al., 2019) aimed at supporting the identification and solution of performance problems on a running system by means of an integrated approach that exploits traceability relationships between the monitoring data and the architectural model to detect and solve performance antipatterns. The extensions introduced in this paper have consisted of: (i) collecting a larger set of metrics and performance measures from the running systems; (ii) translating refactoring actions (suggested by performance model analysis) into refactorings applied to a running system; (iii) closing the performance engineering loop on the original E-Shopper case study by working on its running system; (iv) fully applying the whole approach to the additional Train Ticket case study (at the model and running system level).

The application of our approach to two case studies has consolidated the insight that the definition and the use of traceability models is a key point for automating the exploitation of runtime information for sake of performance analysis and improvement. In fact, in absence of a collection of correspondences between monitored data and system design, the extrapolation of a large amount of data from a running application and the annotation of a software model would be very expensive.

By validating our approach also on Train Ticket, which is a benchmark of realistic size, we have realized that the application of refactoring actions at runtime is not only possible, but also easy to adjust on different technologies. The merit for this result may probably be found in the key concepts imposed by the microservices paradigm, like having small and focused components and emphasizing composability.

The extension of our approach to a larger set of performance metrics confirmed that design/runtime interactions should be automatically generated from a clear declarative specification. In this way, the introduction of new monitoring measures, or even different modeling notations, is just a matter of changing the name and the format of the elements to be matched by the correspondences specification. And this is evidently more convenient than relying on general purpose code to generate such correspondences.

Only by applying refactoring actions on a running system we have been able to appreciate the crucial role of design knowledge to the selection of those actions that better improve a target performance metric. Such knowledge may be proven very difficult to reconstruct just from monitoring. Indeed, while directly extracting performance models from runtime information may obtain accurate models in terms of the ability to predict a performance

change, it could be limiting. In fact, by discarding design models, such approaches are usually compelled to reversely engineered information, and they do not look at already available information, like the role of a component in a scenario or how intensely an operation is supposed to be invoked.

An interesting direction for future work would be to compare the results obtained by applying the refactoring actions suggested by our approach with the ones that would have been obtained by applying human suggestions in the same situations. Indeed, a fully automated process is difficult to be accepted in actual industrial contexts, where the human expertise could be exploited, for example, when multiple refactoring options occur.

We conceived and implemented the approach to be extensible. In this paper, we defined a dedicated metamodel that represents the logs format as represented by the used monitoring infrastructure. However, runtime information can be of different types (e.g., simulation or executable models, logs/traces, states or configurations of the system, test models, dynamic information or runtime measures), it can have different formats (e.g., textual, binary, datasets) and can be collected by means of various mechanisms (e.g., simulation, monitoring, execution, debugging, profiling, verification). As an enhancement, a generic log metamodel that deals with different runtime information can be specified and integrated to the approach.

As a further extension, also the adoption of different modeling languages can be supported by the approach and by the used tools. The designer can specify the correspondences between proper languages; JTL is indeed able to deal with any Ecore artifact conforms to EMF. Moreover, we envisage possibly redefining the model annotation step and the antipattern detection rules. PADRE indeed provides the designer with interfaces to write proper notation-specific rules.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Altamimi, T., Zargari, M.H., Petriu, D.C., 2016. Performance Analysis Roundtrip: Automatic Generation of Performance Models and Results Feedback Using Cross-model Trace Links, in: Proc. of CASCON, pp. 208–217.

Arcelli, D., Cortellessa, V., Di Pompeo, D., 2018. Performance-driven software model refactoring. IST J. 95, 366–397. http://dx.doi.org/10.1016/j.infsof.2017.09.006.

Arcelli, D., Cortellessa, V., Di Pompeo, D., Eramo, R., Tucci, M., 2019. Exploiting architecture/runtime model-driven traceability for performance improvement. In: IEEE International Conference on Software Architecture, ICSA 2019, Hamburg, Germany, March 25-29, 2019. pp. 81–90. http://dx.doi.org/10.1109/ICSA.2019.00017.

Arcelli, D., Cortellessa, V., Trubiani, C., 2015. Performance-based software model refactoring in fuzzy contexts. In: Egyed, A., Schaefer, I. (Eds.), Fundamental Approaches To Software Engineering - 18th International Conference, FASE 2015, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. In: Lecture Notes in Computer Science, 9033, Springer, pp. 149–164. http://dx.doi.org/10.1007/978-3-662-46675-9_10, URL https://doi.org/10.1007/978-3-662-46675-9_10.

Bencomo, N., Götz, S., Song, H., 2019. Models@run.time: a guided tour of the state of the art and research challenges. Softw. Syst. Model. http://dx.doi.org/10.1007/s10270-018-00712-x.

Blair, G., Bencomo, N., France, R., 2009. Models@run.time. Computer 42 (10), 22–27. http://dx.doi.org/10.1109/MC.2009.326.

Brunelière, H., Eramo, R., Gómez, A., Besnard, V., Bruel, J., Gogolla, M., Kästner, A., Rutle, A., 2018. Model-driven engineering for design-runtime interaction in complex systems: Scientific challenges and roadmap - report on the mde@derun 2018 workshop. In: STAF Workshops. In: Lecture Notes in Computer Science, 11176, Springer, pp. 536–543. http://dx.doi.org/10.1007/978-3-030-04771-9_40.

Chen, L., 2018. Microservices: Architecting for continuous delivery and DevOps. In: IEEE International Conference on Software Architecture, ICSA 2018, Seattle, WA, USA, April 30 - May 4, 2018. pp. 39–46. http://dx.doi.org/10.1109/ICSA.2018.00013.

Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A., 2010. JTL: a bidirectional and change propagating transformation language. In: SLE Proc.. pp. 183–202. http://dx.doi.org/10.1007/978-3-642-19440-5_11.

Cito, J., Leitner, P., Bosshard, C., Knecht, M., Mazlami, G., Gall, H.C., 2018. PerformanceHat: augmenting source code with runtime performance traces in the IDE, in: Proc. of ICSE Companion, pp. 41–44, http://dx.doi.org/10.1145/3183440.3183481.

Cortellessa, V., 2013. Performance Antipatterns: State-of-Art and Future Perspectives, EPEW Proc., pp. 1–6, http://dx.doi.org/10.1007/978-3-642-40725-3_1.

Cortellessa, V., Di Marco, A., Inverardi, P., 2011. Model-Based Software Performance Analysis. Springer, http://dx.doi.org/10.1007/978-3-642-13621-4.

Cortellessa, V., Di Marco, A., Trubiani, C., 2014a. An approach for modeling and detecting software performance antipatterns based on first-order logics. Softw. Syst. Model. 13 (1), 391–432. http://dx.doi.org/10.1007/s10270-012-0246-z.

Cortellessa, V., Marco, A.D., Trubiani, C., 2014b. An approach for modeling and detecting software performance antipatterns based on first-order logics. Softw. Syst. Model. 13 (1), 391–432. http://dx.doi.org/10.1007/s10270-012-0246-z.

Derakhshanmanesh, M., Grieger, M., 2016. Model-Integrating Microservices: A Vision Paper, in: Gemeinsamer Tagungsband Der Workshops Der Tagung Software Engineering 2016 (SE 2016), pp. 142–147.

Di Pompeo, D., Tucci, M., Celi, A., Eramo, R., 2019. A microservice reference case study for design-runtime interaction in MDE. In: Bagnato, A., Brunelière, H., no, L.B., Eramo, R., Gómez, A. (Eds.), STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop Co-Located with Software Technologies: Applications and Foundations (STAF 2019), Eindhoven, the Netherlands, July 15 - 19, 2019. In: CEUR Workshop Proceedings, 2405, CEUR-WS.org, pp. 23–32, URL http://ceur-ws.org/Vol-2405/06_paper.pdf.

Düllmann, T.F., van Hoorn, A., 2017. Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches. In: Companion Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017. pp. 171–172. http://dx.doi.org/10.1145/3053600.3053627.

Eramo, R., Pierantonio, A., Tucci, M., 2018. Improved traceability for bidirectional model transformations, in: Proc. of MDETools Workshop, MODELS, vol. 2245, pp. 306–315.

Gelfond, M., Lifschitz, V., 1988. The Stable Model Semantics for Logic Programming, in: ICLP, pp. 1070–1080.

Gomaa, H., Albassam, E., 2017. Run-time Software Architectural Models for Adaptation, Recovery and Evolution, in: Proc. of Models@Run.Time Workshop, MODELS, vol. 2019, pp. 193–200.

Grohmann, J., Straesser, M., Chalbani, A., Eismann, S., Arian, Y., Herbst, N., Peretz, N., Kounev, S., 2021. Suanming: Explainable prediction of performance degradations in microservice applications. In: ICPE '21: ACM/SPEC International Conference on Performance Engineering, Virtual Event, France, April 19-21, 2021. ACM, pp. 165–176. http://dx.doi.org/10.1145/3427921.3450248.

Heinrich, R., 2020. Architectural runtime models for integrating runtime observations and component-based models. J. Syst. Softw. 169, 110722. http://dx.doi.org/10.1016/j.jss.2020.110722, URL https://doi.org/10.1016/j.jss.2020.110722.

Kolovos, D., Rose, L., Paige, R., Garcıa-Domınguez, A., 2010. The EPSILON book. Structure.

Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C., 1984. Quantitative system performance: computer system analysis using queueing network models. Prentice-Hall, Inc..

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F., 2006. The DLV system for knowledge representation and reasoning. TOCL 7 (3), 499–562. http://dx.doi.org/10.1145/1149114.1149117.

Mazkatli, M., Monschein, D., Grohmann, J., Koziolek, A., 2020. Incremental calibration of architectural performance models with parametric dependencies. In: 2020 IEEE International Conference on Software Architecture (ICSA). pp. 23–34. http://dx.doi.org/10.1109/ICSA47634.2020.00011.

Newman, S., 2015. Building Microservices, first ed. O'Reilly Media, Inc..

OMG, 2008. A UML profile for MARTE: modeling and analysis of real-time embedded systems, OMG. URL http://www.omg.org/omgmarte/.

OMG, 2015. Unified Modeling Language, Version 2.5, OMG. URL http://www.omg.org/spec/UML/2.5/.

Paige, R.F., Drivalos, N., Kolovos, D.S., Fernandes, K.J., Power, C., Olsen, G.K., Zschaler, S., 2011. Rigorous identification and encoding of trace-links in model-driven engineering. SOSYM 10 (4), 469–487. http://dx.doi.org/10.1007/s10270-010-0158-8.

Porter, J., Menascé, D.A., Gomaa, H., 2016. DeSARM: A Decentralized Mechanism for Discovering Software Architecture Models at Runtime in Distributed Systems, in: Proc. of Models@Run.Time Workshop, MODELS, vol. 1742, pp. 43–51.

Rademacher, F., Sachweh, S., Zündorf, A., 2017. Differences between model-driven development of service-oriented and microservice architecture. In: 2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017. pp. 38–45. http://dx.doi.org/10.1109/ICSAW.2017.32.

Reiser, M., Lavenberg, S.S., 1981. Corrigendum: "mean-value analysis of closed multichain queuing networks". J. ACM 28 (3), 629. http://dx.doi.org/10.1145/322261.322275.

Sampaio, A.R., Rubin, J., Beschastnikh, I., Rosa, N.S., 2019. Improving microservice-based applications with runtime placement adaptation. J. Internet Serv. Appl. 10 (1), 4. http://dx.doi.org/10.1186/s13174-019-0104-0.

Schmidt, D.C., 2006. Model-driven engineering. IEEE Comput. 39 (2), 25–31. http://dx.doi.org/10.1109/MC.2006.58.

Smith, C.U., Williams, L.G., 2002. New software performance antipatterns: More ways to shoot yourself in the foot, in: 28th International Computer Measurement Group Conference (CMG), pp. 667–674.

Trubiani, C., Bran, A., van Hoorn, A., Avritzer, A., Knoche, H., 2018. Exploiting load testing and profiling for performance antipattern detection. IST J. 95, 329–345. http://dx.doi.org/10.1016/j.infsof.2017.11.016.

Vögele, C., van Hoorn, A., Schulz, E., Hasselbring, W., Krcmar, H., 2018. WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction - a model-driven approach for session-based application systems. Softw. Syst. Model. 17 (2), 443–477. http://dx.doi.org/10.1007/s10270-016-0566-5.

Weyns, D., 2019. Software engineering of self-adaptive systems. In: Cha, S., Taylor, R.N., Kang, K.C. (Eds.), Handbook of Software Engineering. Springer, pp. 399–443. http://dx.doi.org/10.1007/978-3-030-00262-6_11.

Winkler, S., von Pilgrim, J., 2010. A survey of traceability in requirements engineering and model-driven development. Softw. Syst. Model. 9 (4), 529–565. http://dx.doi.org/10.1007/s10270-009-0145-0.

Woodside, C.M., Franks, G., Petriu, D.C., 2007. The future of software performance engineering. In: FOSE Workshop, ICSE. pp. 171–187. http://dx.doi.org/10.1109/FOSE.2007.32.

Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Liu, D., Xiang, Q., He, C., 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In: Dumas, M., Pfahl, D., Apel, S., Russo, A. (Eds.), Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019. ACM, pp. 683–694. http://dx.doi.org/10.1145/3338906.3338961.

Zhou, X., Peng, X., Xie, T., Sun, J., Li, W., Ji, C., Ding, D., 2018a. Delta debugging microservice systems. In: Huchard, M., Kästner, C., Fraser, G. (Eds.), Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. ACM, pp. 802–807. http://dx.doi.org/10.1145/3238147.3240730.

Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., Zhao, W., 2018b. Benchmarking microservice systems for software engineering research. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (Eds.), Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. ACM, pp. 323–324. http://dx.doi.org/10.1145/3183440.3194991.

Zúñiga-Prieto, M., Insfran, E., Abrahão, S., Cano-Genoves, C., 2017. Automation of the Incremental Integration of Microservices Architectures, in: Complexity in Information Systems Development, pp. 51–68, http://dx.doi.org/10.1007/978-3-319-52593-8_4.