The Journal of Systems and Software 112 (2016) 156–180

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Evaluating REST architectures—Approach, tooling and guidelines

Bruno Costa [a,*], Paulo F. Pires [a], Flávia C. Delicato [a], Paulo Merson [b]

[a] Department of Computer Science, Federal University of Rio de Janeiro (UFRJ), Rio de Janeiro, Brazil
[b] Federal Court of Accounts (TCU), Brasília, Brazil

ARTICLE INFO

Article history:
Received 27 November 2014
Revised 18 September 2015
Accepted 30 September 2015
Available online 8 October 2015

Keywords:
Software architecture evaluation
Scenario-based evaluation guidelines
REST

ABSTRACT

Architectural decisions determine the ability of the implemented system to satisfy functional and quality attribute requirements. The Representational State Transfer (REST) architectural style has been extensively used recently for integrating services and applications. Its adoption to build SOA-based distributed systems brings several benefits, but also poses new challenges and risks. Particularly important among those risks are failures to effectively address quality attribute requirements such as security, reliability, and performance. A proved efficient technique to identify and help mitigate those risks is the architecture evaluation. In this paper we propose an approach, tooling, and guidelines to aid architecture evaluation activities in REST-based systems. These guidelines can be systematically used along with evaluation methods to reason about design considerations and tradeoffs. To demonstrate how the guidelines can help architecture evaluators, we present a proof of concept describing how to use the guidelines in an ATAM (Architecture Tradeoff Analysis Method) evaluation. We also present the results of a survey conducted with industry specialists who have performed architecture evaluations in real world REST-based systems in order to gauge the suitability and utility of the proposed guidelines. Finally, the paper describes a Web tool developed to facilitate the use of the evaluation guidelines.

1. Introduction

Architectural decisions determine the ability of the implemented system to satisfy functional and quality attribute requirements. A functional requirement specifies a function that a system must be capable of performing. In turn, a quality attribute requirement (also known as "non-functional requirement") qualifies the functional requirements by describing how the software will perform them (Chung et al., 2001). For example, considering a functional requirement defined as "*in order to keep track of stock, as a store owner, he/she wants to add items back to stock when they are returned*", some associated quality attributes may be: (i) performance, describing how long should it take to perform the task of adding the items back to stock; (ii) usability, describing how easy is the system's interface to the user; or (iii) security, specifying that only authorized users can perform that functionality. Since architectural decisions are among the first to be made during the software development life cycle, and they affect later stages of the development process, the impact of architectural mistakes is high. Therefore, it is important to inspect the architecture early in the development process to identify and mitigate any risks of the software solution not satisfying the quality attribute requirements. This inspection activity is referred to as software architecture evaluation (Bass et al., 2012).

Software architecture evaluation is a Software Engineering methodology composed of several activities that use as input the requirements and the architecture description. The evaluation is performed through the analysis of the design approaches and decisions present in the architecture vis-à-vis their ability to fulfil the software requirements. The architecture description typically consists of multiple architecture views and may follow, for example, the 4+1 architectural view model (Kruchten, 1995).

Among the architecture evaluation methods, scenario-based methods (Kazman et al., 1996) assess the architecture based on several scenarios of interest defined by the architects, users and other stakeholders. Such scenarios are descriptions of the desired behaviour of a system under some condition regarding quality attribute. For example, a desired performance scenario for a Web-based sales system could be: "*when a user submits the purchase confirmation on the Web form the system shall perform the transaction and show the response/status message to the user in less than 5 seconds*".

Since the 1990s, several architecture evaluation methods have been proposed, such as SAAM (Scenario-based Software Architecture Analysis Method) (Clements et al., 2001), SBAR (Scenario-based Architecture Reengineering) (Bengtsson & Bosch, 1998), and SAAMCS (SAAM for Complex Scenarios) (Lassing et al., 1999). Among

* Corresponding author. Tel.: +5521988868883.
E-mail addresses: brunocosta.dsn@gmail.com (B. Costa), paulo.f.pires@gmail.com (P.F. Pires), fdelicato@gmail.com (F.C. Delicato), pmerson@acm.org (P. Merson).

http://dx.doi.org/10.1016/j.jss.2015.09.039
0164-1212/

all these methods, ATAM (Architecture Tradeoff Analysis Method) (Kazman et al., 1998) has been the most widely adopted in industry and academia. Extensive literature, including several books and published case studies describe the application of ATAM. Professional training and professional certification on ATAM is also available. Bertolino and colleagues state that ATAM can be considered one of the most mature and often used from among the existing methods (Bertolino et al., 2013). Furthermore, according to Babar and Gorton, ATAM provides a comprehensive process support in comparison with other scenario-based methods (Babar & Gorton, 2004). It involves eliciting quality attribute requirements among stakeholders and assessing the architecture by searching for risks to address the quality requirements and for the tradeoffs between them. As it occurs in other architecture evaluation methods, ATAM describes the steps that a team should follow to evaluate the architecture. However, the inspection is strongly based on the empirical knowledge of the evaluators, on the clear understanding of the impact of applicable architectural patterns, and other technical guidance that helps to assess the fitness of the architecture. Technical guidance that clarifies benefits and drawbacks of specific design decisions is particularly important when the assessed architecture is strongly based on a new or complex architectural style.

The Software Engineering Institute (SEI) and other research organizations have published technical reports that work as guides to assist in different software engineering activities. These guides document the best practices used by specialists, analysed in light of available theoretical knowledge and validated by professionals from industry and academia. One of these guides, presented in the next paragraph, was written to help the evaluation of the architecture of SOA-based systems (Bianco et al., 2007).

In order to apply proven good design structures, the architecture is often designed according to existent architectural patterns and styles (Bass et al., 2012). An architecture style is a specialization of element and relation types, together with a set of design constraints on how they can be used (Clements et al., 2010). A design constraint, in turn, can be defined as a limiting factor which specifies the conditions that a viable design solution must satisfy (Tang & van Vliet, 2009). The use of architecture styles has direct impact on quality attributes (Bass et al., 2012, Bianco et al., 2007, Clements et al., 2010). As an example, Service-oriented Architecture (SOA) is an overarching architectural style for distributed systems commonly used to address interoperability and integration requirements. Since services are distributed components, the risk and impact of SOA are pervasive across applications, so it is critical to perform an architecture evaluation early in the software lifecycle (Bianco et al., 2007). To assist the architecture evaluation of SOA-based systems, in 2007 researchers at the SEI and industry experts published a technical report titled "*Evaluating a Service-Oriented Architecture*" (Bianco et al., 2007). The report contains guidelines about SOA design considerations, impact of the different design alternatives (e.g., synchronous vs asynchronous service) in quality attributes and the tradeoffs between them. With respect to the communication mechanism between service consumers and services, there are two main approaches to implement SOA-based services: SOAP and REST. In the aforementioned report, the focus is on the SOAP approach and limited information is presented about the REST approach. In recent years, REST services became widely used to build Web-based distributed systems. Several development frameworks were created and vast technical literature was produced about the design and implementation of REST services. Therefore, the importance of REST in the context of SOA-based systems has gained momentum.

REST was originally proposed as a software architectural style for distributed systems in Roy Fielding's Ph.D. dissertation (Fielding, 2000). Fielding presents a set of *REST constraints*, such as uniform interface, stateless, and client–server, which are design restrictions prescribed by the REST architectural style. By using REST, some quality attributes of the system, such as interoperability and modifiability, are positively impacted, whereas others, such as performance and reliability can be negatively impacted. Architecture evaluators should identify places and design decisions in the architecture that influence the ability of the system to meet the quality attribute requirements. This task requires a clear understanding of the tradeoffs between quality attributes. The widespread use of REST has recently brought to the minds of project managers, architects and architecture evaluators (AEs) the following question: "*what is the impact of changing towards the SOA paradigm?*" (Naab, 2013). The report written by Bianco and colleagues (Bianco et al., 2007) can help to answer this question in the more general context of SOA, but does not provide up-to-date and in-depth analysis on the impact of REST constraints (e.g., uniform interface and stateless) on quality attribute requirements.

Designing REST services poses two main challenges: adequately fulfil the aforementioned REST constraints (Haupt et al., 2014) and properly address quality attribute requirements (Costa et al., 2014). Recent studies have shown that in most cases REST constraints are not addressed to their full extent (Wilde & Pautasso, 2011, Renzel et al., 2012; Maleshkova et al., 2010). Departure from REST constraints can negatively impact quality attribute properties. Beyond the REST constraints defined in Fielding's dissertation, there are several decisions in the design of REST-based solutions known today to affect the achievement of quality requirements. REST services can be a small part of a software solution, but often times they are pervasive and critical for the solution. The evaluation of a software architecture that positions REST services as principal components requires sound understanding of the REST constraints and other design decisions. To properly conduct such evaluation, AEs would certainly welcome a guide that provides reliable and detailed information about REST principles, constraints, design decisions, and their impact on quality attributes.

The purpose of this paper is to provide a guide for architecture evaluation activities in systems that use the REST architectural style. We discuss several design aspects and provide guidelines on important inspection points. We also discuss how the architecture can be probed by the evaluation team and propose the types of questions that should be asked during the evaluation process in order to ensure the realization of desired quality attributes taking into account the REST principles and constraints. Because of its maturity and reach across the software architecture community, we chose the ATAM method to illustrate how elements of our evaluation guide can help in different activities of an architecture evaluation. However, the guide presented in this paper can be used within other scenario-based architecture evaluation methods, as explained in Section 3. To demonstrate how the guidelines can help evaluators to mitigate risks in an architecture evaluation, we present a proof of concept project that also used ATAM along with the design questions and general scenarios we developed for REST architecture evaluations. We also present the results of a survey conducted with industry specialists who have performed architecture evaluations in real world REST-based systems. The survey tried to gauge the suitability and utility of our evaluation guidelines. Finally, the paper describes a Web tool developed to support the use of the evaluation guidelines. The architecture evaluation guide presented in this paper is an extended and improved version of our previous work (Costa et al., 2014). The main improvements are the survey conducted with industry specialists and the description of the implemented Web tool. The guide was also improved by: (i) adding new general quality attribute scenarios (Section 6) and new design questions (Section 7) to reason about service replication, dependency, packaging and deployment of microservices; (ii) including information to aid the evaluator in the creation of concrete scenarios (Section 6), and; (iii) a more in-depth explanation of the employed research method (Section 4).

The rest of this paper is organized as follows: The background section (Section 2) introduces the topic by discussing several aspects of

software quality and architecture evaluation. In Section 3 we emphasize our contributions. Section 4 describes the research protocol used to develop this work. In Section 5 we present the architectural foundations of REST from the point of view of an AE. REST general quality attribute scenarios are described in Section 6. Design questions that affect quality attributes are discussed in Section 7. A proof of concept evaluation that uses the guidelines introduced in this paper is described in Section 8. The survey conducted with AEs is described in Section 9. In Section 10 we present the Web tool we developed to assist evaluations. Section 11 analyses related work. Finally, Section 12 presents the conclusions and future work.

2. Background

2.1. Software quality

In the ISO/IEC 25010 standard (ISO 25010:2011 2011), software quality is defined as the *degree to which a software product satisfies stated and implied needs when used under specified conditions*. Such needs can be seen as the requirements of the system, which encompass the following categories (Bass et al., 2012): (i) *Functional requirements*, which state what the system must do, and how it must behave or react to runtime stimuli; (ii) *Quality attribute requirements*, which indicate the degrees to which the system must exhibit various properties, such as performance, availability, and modifiability, and; (iii) *Design constraints*, which are decisions imposed to the software development team, such as the use of a certain programming language or development framework.

As presented in ISO/IEC/IEEE 42010 standard (ISO/IEC/IEEE 42010:2011 2011), these systems' requirements are defined by stakeholders, who are individuals, teams, organizations, or classes thereof, with interests in a system. Such interests are called in the standard as stakeholder's *concerns*. Examples of stakeholders, their responsibilities and concerns are (Bass et al., 2012): the *End User*, who is responsible for the definition of the system's functional requirements and is primarily concerned about the realization of such requirements; the *Implementer*, who is responsible for the development of the system and is concerned on the constraints and exploitable freedoms on development activities; the *Architect*, who is responsible for the development of the architecture and is concerned about making tradeoffs among competing requirements and design approaches, and; the *Evaluator*, who is responsible for conducting a formal evaluation of the architecture against some clearly defined criteria and is concerned about the architecture's ability to deliver required behaviour and quality attributes.

Therefore, software quality is measured against the degree to which stakeholders' requirements and other concerns are met in the system. Such statement can lead to another conclusion: measuring software quality is challenging since the measurement must consider the perspectives of different stakeholders. The quantifiable or testable aspects of a system that indicate to stakeholders the software quality are the quality attributes.

2.2. Quality attributes

The IEEE Glossary of System and Software Engineering defines quality attributes as *a feature or characteristic that affects an item's quality* (ISO/IEC/IEEE 24765:2010 2010). Bass and associates state that *a quality attribute (QA) is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders* (Bass et al., 2012). While there are some quality models that formally define quality attributes (such as Bohem's Quality Model (Boehm et al., 1978), McCall's Quality Model (McCall et al., 1997), and ISO/IEC 25010 Quality Model (ISO 25010:2011 2011)), they can be categorized into two broad groups: attributes that can be quantitatively measured and attributes that can be qualitatively analysed. The difference between both is that the former has analytical models that provide an accurate quantitative analysis to enable prediction of the behaviour of a designed system (Bass et al., 2012). Some examples of quality attributes that have analytical models are: (i) reliability, assessed by Markov and Statistical Models (Franco et al., 2012); (ii) performance, assessed by Queuing theory and Real-time scheduling theory (Koziolek, 2010), and; (iii) modifiability, assessed by coupling and cohesion metrics (Chidamber & Kemerer, 1994). For other quality attributes (and even for those that have analytical models but they are not mature or their usage is deemed too expensive), guidelines and checklists exist to allow the architect to assess compliance or to guide the architect when making design decisions. One of the most used techniques to specify quality attribute requirements is quality attribute scenarios.

2.3. Quality attribute scenarios

A *quality attribute scenario* is a structured description of a quality attribute requirement that is unambiguous and testable (Bass et al., 2012). Quality attribute requirements can be expressed by two types of scenarios: *(i) general quality attribute scenarios* (hereinafter *general scenarios*), which are generic and can be applied to any software system; and *(ii) concrete quality attribute scenarios* (or *concrete scenarios*), which are specific to the particular system under consideration (Bass et al., 2012). Concrete scenarios are usually created in light of general scenarios.

There are a variety of published quality attribute models, as mentioned above. However, Bass and associates (2012) advocate that there are problems in the application of such quality attribute models: (i) The definitions provided for an attribute are generic, not testable; (ii) the discussion about the quality attributes leads you to classify a particular quality requirement based on a quality attribute taxonomy, and actual quality attribute requirements may fit different classifications; and (iii) each attribute community has developed its own vocabulary. To avoid such problems, architects could use quality attribute models simply to assist them in the definition of quality attribute scenarios. Indeed, the ISO 25010 standard itself states that quality models should be tailored before use as part of the decomposition of requirements to identify those characteristics and sub characteristics that are most important to the application (ISO 25010:2011 2011). Therefore, quality models can be used as starting point to define quality attribute scenarios: they provide the general information about the quality attributes that are interesting to the definition of the scenario. However, the stakeholder must complement the information with characteristics that are specific to the system and impact the quality attribute.

The specification of a quality attribute scenario has six parts: (1) **Source of stimulus**: Some entity (a human, a computer system, or any other actuator) that generates a given stimulus; (2) **Stimulus**: A condition that requires a response when it arrives at a system; (3) **Environment**: The conditions for the stimulus occurrence; (4) **Artefact**: The artefact that is stimulated; (5) **Response**: The activity undertaken as the result of the arrival of the stimulus, and; (6) **Response measure**: The response of the system, which should be measurable in some way so that the requirement can be tested. Despite the six-part scenarios provide an efficient form to specify quality attribute scenarios, in general, a well-formed scenario has only a stimulus-and-response structure (Bass et al., 2001):

$$\underbrace{\text{Thesystemreceivesastimulus}}_{\text{Stimulus}} \text{ and } \underbrace{\text{somedesirableresponseisobserved}}_{\text{Response}}$$

Commonly, stimulus-and-response structures are used to specify general scenarios and six-part forms specify concrete scenarios. Moreover, general scenarios are used as reference for generating concrete scenarios (Bass et al., 2012, Bianco et al., 2007). In turn, concrete scenarios are input to the software architecture specification.

2.4. Software architecture

The architecture of a software system is the set of structures needed to reason about it; such structures comprise elements, relations among them, and properties of both (Clements et al., 2010). Each structure may exhibit different types of elements and relations. In structures that focus on the organization of the code, the architectural elements are generically called modules, which are implementation units, such as Java or C# classes. In other structures that describe the system at runtime, we find components and connectors, which are elements that have runtime presence. But there are many other possible structures, such as the allocation of components to the hardware infrastructure, the organization of deployment artefacts, and the data entities and their relationships.

The architecture plays a central role in software development because it will guide the transition from stakeholders' requirements and concerns to the implemented software system. A key aspect of architecture is to enable the realization of quality attribute requirements. In fact, the architecture specification should provide evidence that the stakeholders' requirements are addressed before the system implementation and roll out. However, it is not a trivial task to ensure that a given software architecture can lead to an implementation that meets all requirements. The Software Architecture community has been researching and proposing techniques to efficiently predict quality attributes in the software architecture specification. One of the more mature techniques to analyse a software architecture to tell whether quality attribute requirements are adequately addressed is the scenario-based architecture evaluation.

2.5. Scenario-based architecture evaluation

Scenario-based evaluation methods evaluate the software architecture's suitability according to a set of scenarios of interest. These methods typically include five activities (Babar & Gorton, 2004): (1) evaluation planning and preparation; (2) explain software architecture approaches; (3) elicit quality attribute sensitive scenarios; (4) analyse software architecture approaches, and (5) interpret and present results.

In the last decade, several architecture evaluation methods have been proposed (Roy & Graham, 2008). For example, the SBAR (Bengtsson & Bosch, 1998) estimates how well the specified architecture fulfils the requirements according to development-time quality attributes, also called operational quality attributes, such as maintainability and reusability. The method is comprised of four different techniques for assessing the architecture against quality attributes: (i) *Scenarios*, as the specification of the quality attributes; (ii) *Simulation*, which complements the scenarios in order to evaluate the software qualities, such as performance and fault-tolerance; (iii) *Mathematical models*, providing analytical approaches that can be used to quantitatively assess the architecture, and (iv) *Experience-based reasoning (EBR)*, which takes into account the experience of the evaluation team and their logical arguments about the adequacy of the architecture to address quality attribute scenarios. The general scenarios proposed by our guide (Section 6) can assist the evaluation team to generate the concrete scenarios used in an SBAR evaluation. The design questions (Section 7) can help to elicit and discuss tradeoffs in the EBR.

Another example of scenario-based evaluation method is the SAAMCS (Lassing et al., 1999). The objective of SAAMCS is to handle complex scenarios that are hard to implement. Such scenarios are defined according to three factors that influence their complexity: (i) *level of impact*, which estimates the level of impact of the scenario on the software architecture; (ii) *need for coordination*, where components in one scenario can affect components that belong to different scenarios, and (iii) *presence of version conflicts*, which deals with the management of versioning of components that can impact in different scenarios. The SAAMCS method relies on the scenario-based SAAM (Clements et al., 2001), one of the first proposed methods, and has six activities: (1) *Specify requirements and design constraints*: The evaluation team collects the functional and quality attribute requirements, as well as design constraints; (2) *Describe software architecture*: The candidate architectures are described using a well-known Architecture Description Language (ADL); (3) *Elicit scenarios*: Scenarios are elicited and specified with the presence of the relevant stakeholders; (4) *Prioritize scenarios*: The stakeholders prioritize scenarios according to their importance; (5) *Evaluate architectures with respect to scenarios*: The evaluation team verifies the impact of the scenarios on the architecture, and; (6) *Interpret and present results*: The evaluation team interprets the results according to the evaluation goals and present the results.

In addition to SBAR and SAAMCS, there are several other scenario-based architecture evaluation methods. In our work we have chosen ATAM as a reference method because it is one of the most mature and most used from among the existing architecture evaluation methods. Also, the technical and scientific literature available about ATAM provides better comprehensive process support in comparison with any other scenario-based method.

2.6. Architecture Tradeoff Analysis Method

The ATAM provides a principled way to evaluate the fitness of a software architecture with respect to multiple competing quality attribute requirements (Kazman et al., 1998). The method was created to uncover the risks and tradeoffs reflected in architectural decisions related to quality attribute requirements. The ATAM method consists of nine steps that comprise six main activities, as depicted in the UML activity diagram in Fig. 1: (1) **Present the method**: The evaluation team presents a quick overview of the method steps, the techniques used, and the outputs of the process activities; (2) **Present the business goals**: The system manager briefly presents the business goals and context for the architecture; (3) **Present the architecture and identify architectural approaches**: The architect presents an overview of the architecture and the evaluation team identifies the architectural tactics and patterns used in the design; (4) **Generate quality attribute scenarios:** The architect, along with several other stakeholders and the evaluation team identify the quality attribute requirements and describe them as concrete scenarios. General scenarios can be used in this step as reference for generating concrete scenarios; (5) **Analyse the architectural approaches according to concrete scenarios and business goals**: The goal is to assess whether the architecture can satisfy the top priority scenarios. The evaluation team probes the architectural approaches in light of the concrete scenarios and business goals. To probe the architecture, the team uses design questions that are typically based on their practical experience, but could also come from architecture evaluation guides; and (6) **Present the results**: The evaluation team presents the outcome of the architecture evaluation, which includes: prioritized quality attribute scenarios, documentation of the architectural approaches in use with rationale, uncovered risks and the requirements that are threatened by them.

3. Architecture evaluation guide for REST-based systems

REST has seen increasing popularity in the past 10 years. Today it is a common approach for distributed systems, especially those that involve mobile apps, public APIs, and SaaS. REST is also often used in corporate IT as the communication mechanism across enterprise applications. Our architecture evaluation guide is useful in these ever more common situations where REST has been chosen as the primary mechanism for service interaction. Nonetheless, it should be clear to software architects and evaluators that REST is not the best alternative for all systems, and the primal choice for REST should arise

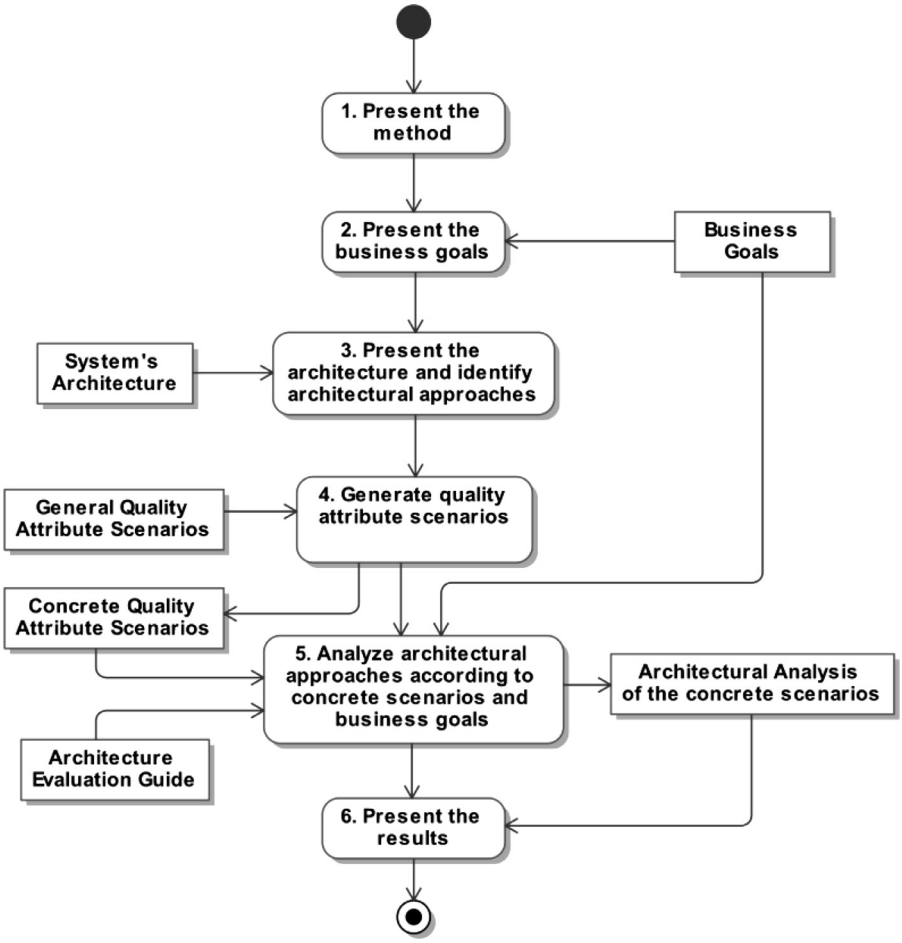


Fig. 1. Main activities and artefacts of ATAM.

from clear understanding of the quality attribute requirements, design constraints and organizational context.

The main contribution of this paper is to propose an architecture evaluation guide that includes general scenarios, design questions, and guidelines specific to REST-based solutions. The guide can be used to validate the overall choice for REST and to evaluate smaller scope design decisions, such as authentication mechanism for REST services and packaging and deployment of REST services as microservices. The general scenarios, design questions and guidelines can assist the evaluators primarily in two typical activities of scenario-based architecture evaluation methods: elicit quality attribute scenarios, and analyse architectural approaches. In an ATAM evaluation, general scenarios help stakeholders and evaluators to generate system specific, concrete scenarios. The design questions and guidelines in the evaluation guide contain a discussion of the tradeoffs involved in each design alternative and hence help evaluators to assess the architectural approaches against concrete scenarios. Section 8 gives an example of the application of the guide in an ATAM evaluation process.

Although the evaluation guide was created by taking ATAM as the reference architecture evaluation method, we believe it can be equally useful in other scenario-based evaluation methods. For example, in the EBR technique that is part of the SBAR method, the design questions and guidelines can assist evaluators by filling any gap of knowledge or experience regarding REST constraints, design alternatives and tradeoffs involved. SAAMCS evaluations can also benefit from our evaluation guide. In the third activity of SAAMCS, the elicitation of scenarios is developed based on the quality attributes required by the stakeholders. When the architecture is based on REST, the general scenarios in our guide can be used to help the evaluation team and stakeholders to create the concrete scenarios that specify the desired quality attributes. In addition, the design questions may assist the evaluation team in the fifth activity of SAAMCS by providing means to investigate the design decisions in the architecture against the concrete quality attribute scenarios. Furthermore, SAAMCS differentiates two types of scenarios: direct scenarios, which are already supported by the architecture, and indirect scenarios, which require changes in the architecture. For indirect scenarios, the cost of performing the necessary changes is estimated. The proposed evaluation guide can help evaluators to determine what changes are needed and what the effort to perform them is. Finally, the design questions can help to define the impact of changes and need for coordination between components, since they provide information about interactions among competing quality attributes.

4. Research method

The evaluation guide proposed in this paper was developed following Evidence-Based Software Engineering (EBSE) techniques. EBSE aims to provide knowledge about when, how and in what context technologies, processes, methods or tools are more appropriate for Software Engineering practices (Kitchenham, 2004; Dyba et al., 2005). In our work, we combined two EBSE techniques: survey and systematic mapping (SM). We conducted two surveys, with different purposes: (i) in-depth interviews with AEs to collect their concerns regarding architecture evaluation of REST-based solutions; (ii) interviews with industry experts in order to construct the set of REST general quality attribute scenarios and design questions based on their

Table 1
Survey categories, objectives, and participants.

Survey category	Objective	Participants
In-depth interviews	Gather an initial set of REST-specific architecture evaluation concerns from a practitioners' point of view	Architecture evaluators
Interviews	Provide empirical knowledge in order to address the evaluation concerns gathered in the in-depth interviews with architecture evaluators	Specialists

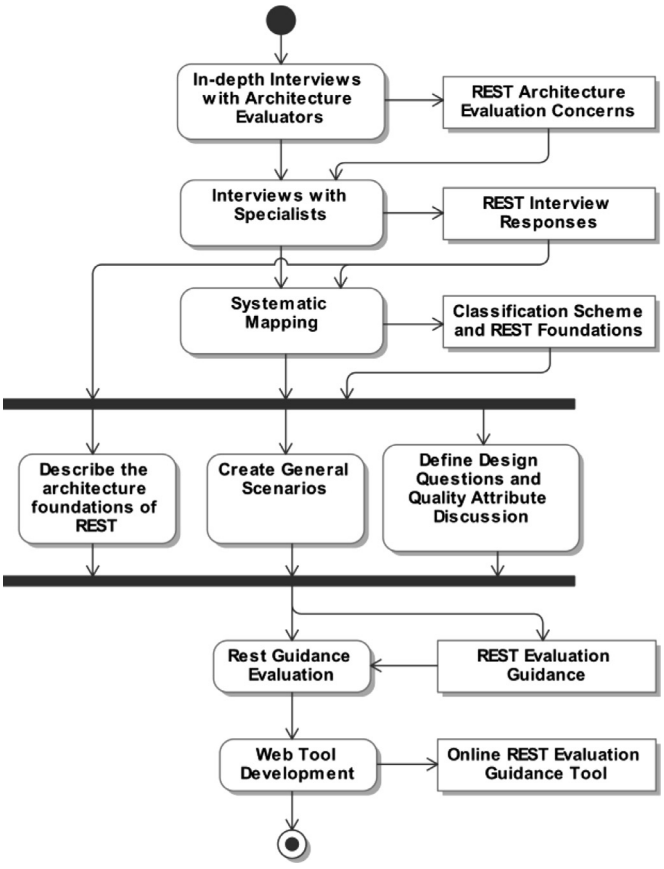


Fig. 2. Activities in our research method.

empirical knowledge about REST principles and design issues. Besides the surveys, we performed an SM in order to create a classification scheme to be used in the organization of the evaluation guide, establish the foundations of REST, and complement the interviews responses with the available knowledge from literature. Having concluded the SM, we performed three activities in parallel: describe the architecture foundations of REST, craft general scenarios involving REST solutions, and define design questions and quality attribute discussion applicable to the evaluation of REST-based systems. The evaluation guide was the final outcome of these three activities. We then sent the evaluation guide to AEs in order to assess if it had achieved its goals. The following subsections detail the process we followed, which is depicted in the UML activity diagram in Fig. 2.

4.1. Surveys

A survey is a comprehensive system for collecting information to describe, compare or explain knowledge, attitudes and behaviour (Pfleeger & Kitchenham, 2001). Basically, a survey asks the respondents to answer questions in order to describe a phenomenon of interest or assess the impact of some intervention (Kitchenham & Pfleeger, 2001). To achieve its goals the survey must be: (i) resilient to bias: results that are likely not to be unduly swayed by a particular faction, aspect or opinion, but reflect the reality of the situation; (ii) appropriate: makes sense in the context of the population (in our context, AEs), and; (iii) cost-effective: the administration and analysis are within the resources allocated to the research.

Surveys fall into two categories: questionnaires and interviews (distinguished from in-depth interviews) (Kasunic, 2005). Questionnaires are forms (nowadays, usually Web forms) composed of a set of questions sent to a pre-defined subgroup of the population that is called "sample". A survey interview is one in which an interviewer asks questions face-to-face, by telephone or Web communication based on a questionnaire tailored to the interview goals. In-depth interviews are a kind of interview where the researcher may work from a list of topics and possible questions just as a start point but the interviewer is free to add new questions and topics. Regardless of the category, the first step to start any survey research is defining objectives. In our study, we have two different objectives related to two different surveys. Each survey was performed with different participants. Table 1 shows the two surveys we conducted and their respective objectives. The following subsections detail each survey.

4.1.1. In-depth interviews with architecture evaluators

The first activity consisted of several in-depth interviews with AEs to gather an initial set of REST-specific architecture evaluation concerns (AECs) from a practitioners' point of view. Following a systematic methodology (Kasunic, 2005; Oishi, 2002), we interviewed two experts in architecture evaluation that have participated in more than 50 architecture evaluations in industry using ATAM. By interviewing two architecture evaluation leaders, it was possible to compare and analyse their common needs, and avoid bias. As the result of the interviews, the evaluators indicated that a REST architecture evaluation guide must address three main issues. In our work, we call these issues *REST AEC's*:

AEC1–"Explain the architectural foundations of REST from an architecture evaluator's point of view".
AEC2—"Discuss quality attributes and general scenarios impacted by REST principles".
AEC3—"Discuss (in detail) how REST contributes to the quality attributes and where typical tradeoffs are".

4.1.2. Interviews with specialists

We next engaged several experts, practitioners and researchers who have long been designing and studying REST solutions. The interaction with these experts was based on the methodology proposed by Kitchenham and Pfleeger (Pfleeger & Kitchenham, 2001; Kitchenham & Pfleeger, 2001, 2002a, 2002b, 2002c) and Oishi (2002). All of the interviewees have worked with REST for more than five years and have experience with 5–10 different REST projects. The interviewees were: three researchers, three developers of REST services, three consultants, and three developers of REST service consumers. The researchers have published several papers, participated or organized conferences in the field, such as the International Workshops on RESTful Design (WS-REST). The developers of REST services work in mid to large companies from the TV, Internet movies, news media, and online maps domains. Overall their REST services are used by 10–30 million clients running on one thousand different kinds of devices. The consultants have worked on several REST lifecycle activities, such as conception, API design, and resources definition. Other developers have extensively used REST services from various providers in their software solutions. The aggregated experience about REST design of these specialists was used as the first source of knowledge to address

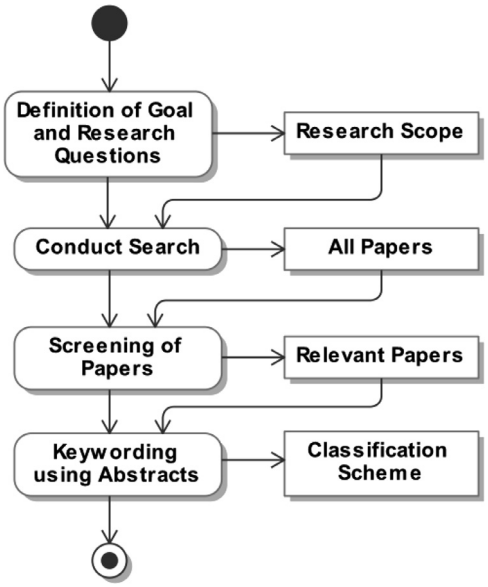


Fig. 3. The systematic mapping process.

the REST AECs. The interviews followed a list of questions developed to extract information from the specialists in order to address the AECs:

1. What is your experience in developing systems that implement or use REST concepts?
2. How many software projects that implement REST concepts have you worked on?
3. According to your experience, what are the main benefits of the REST style?
4. To achieve the aforementioned benefits what elements have to be carefully designed in architecture?
5. What are the main limitations of the REST style?
6. In a REST-based architecture life cycle what are the main issues?
7. What are the main design questions in a REST-based architecture?
8. Could you mention some important inspection points in a REST-based architecture that are related to quality attribute requirements (interoperability, usability, security, among others)?

The interviews with experts produced a significant amount of data that was captured in interview datasheets, text documents, annotations, and literature excerpts. The data were analysed and used as a source for the development of REST general quality attribute scenarios and design questions. The activity that followed, namely SM, had two goals: organize all information in a structured document to generate the evaluation guide, and carry on a comprehensive literature review on REST.

4.2. Systematic mapping

In the last few years, much information about the design and implementation of REST services has been published. These publications proved very useful to address the *REST AECs*. In order to build a classification schema for the empirical knowledge gathered through interviews aligned to the literature review, we conducted an SM (Petersen et al., 2008).

An SM is a well-defined method to build a classification scheme and to structure a field of interest. It provides mechanisms to analyse and to structure studies that have been published by categorizing them and giving a summary of their results. In our SM, we followed the methodology proposed by Petersen and colleagues (2008) shown in the UML activity diagram in Fig. 3 and described next.

Table 2
Electronic databases selected as sources for the search process in the SM.

Database	URL
IEEE Xplore	http://ieeexplore.ieee.org
ACM Digital Library	http://dl.acm.org
ScienceDirect.com	http://www.sciencedirect.com
CiteSeerX	http://citeseer.ist.psu.edu
SpringerLink	http://link.springer.com
Cornell University Library	http://arxiv.org

4.2.1. Definition of goal and research questions

The research questions guide the entire process of an SM. The goal of an SM is to provide an overview of a research area based on research questions that are defined at the beginning of the process. The research questions (RQ) below were defined for our SM, based on the *REST AECs* that were obtained in the first step of our research (interviews with AEs). These research questions guided our search for publications related to REST in the SM process.

RQ1: What are the foundations of REST?
RQ2: What are the quality attributes impacted by REST principles?
RQ3: How quality attributes are addressed in REST-based systems?

4.2.2. Conduct search

Based on the research questions, keywords were selected to compose a search string that was used to find primary studies on scientific databases or to manually browse through relevant conference proceedings and journal publications. Two main keywords were initially identified: *REST* and "*quality attribute*". In addition, possible variations such as synonyms, singular/plural forms, and terms related to service design were considered, thus resulting in the following search string:

(REST OR RESTful OR "representational state transfer") AND (basics OR "service design" OR "design patterns" OR "quality attribute" OR "quality attributes")

The term "basics" refers to studies that provide the foundations of the REST style. We included the terms "service design" and "design patterns" aiming to retrieve papers that describe architectural styles and design decisions that affect quality attributes. Finally, the term "quality attribute" and its plural were included in order to capture papers that explicitly provide a discussion of quality attributes in the context of REST.

We chose six electronic databases that are the most commonly used in systematic reviews in the Software Engineering domain, as pointed out by Dyba and colleagues (2005) and Kitchenhan and Charters (2007). Applying the search string to the following databases, 384 studies were obtained. Their publication dates ranged from 2000 January to 2014 June (Table 2).

4.2.3. Screening of papers

The inclusion (IC) and exclusion criteria (EC) are used to exclude studies that are not relevant to answer the defined research questions. For a publication to be selected, it must meet at least one IC and must not match any of the EC.

The considered IC were:

IC1: The study presents foundations of REST.
IC2: The study discusses quality attributes in the context of REST.

The established EC were:

EC1: The study is not available in its complete form (full-access).
EC2: The study is not directly related to REST.

Table 3
Classification scheme and related publications.

Topic	Publications
Foundations of REST	Fielding (2000), Richardson and Ruby (2007), Erl et al. (2012), Webber et al. (2010), Vinoski (2007), Wilde and Pautasso (2011), Abeysinghe (2008)
Design of services	Adamusiak et al. (2011), de Cluni et al. (2012), Doboš et al. (2013), Dodero and Ghiglione (2008), Pautasso et al. (2014), Sletten (2013), Xu et al. (2010), Xu et al. (2008)
Representation and identification	Abeysinghe (2008), Burke (2009), Dodero and Ghiglione (2008), Immonen and Pakkala (2014), Li and Chou (2009), Lu et al. (2012), Sletten (2013), Gulden and Kugele (2013)
Documentation and testing	Abeysinghe (2008), Bianco et al. (2007), Gambi and Pautasso (2013), Pautasso et al. (2014)
Behaviour	Abeysinghe (2008), Adamusiak et al. (2011), Daigneau (2011), Gambi and Pautasso (2013), Jamal and Deters (2011), Li and Chou (2009), Lu et al. (2012), Sundvall et al. (2013), Xu et al. (2008)
Security	Bianco et al. (2007), Burke (2009), Dodero and Ghiglione (2008), Li and Chou (2009), Lu et al. (2010), Pautasso et al. (2014), Sundvall et al. (2013)

EC3: The study does not address any quality attribute in the context of REST.
EC4: The study consists of a compilation of other works.
EC5: The study is a previous version of a more complete paper about the same research.
EC6: The study is not written in English.

The search on the databases retrieved 384 studies that were systematically analysed by their title and abstract. After this step, 42 scientific papers were selected based on the screening criteria. In addition, eight books and nine technical reports were selected.

4.2.4. Keywording using abstracts

After screening the selected studies, a systematic process was performed to build the classification scheme based on abstracts. This activity is named *keywording* and is done in two steps. First the abstracts are read in order to find keywords and concepts that reflect the contribution of the paper, and to identify the context of the research. The result is a set of keywords from different papers. The second step is to combine the keywords together to develop the categories that will comprise the classification scheme. If it is not possible to identify keywords by reading only the abstract, reviewers can choose to analyse also the introduction or conclusion sections of the paper.

In our study, six categories were created, derived from the keywords extracted from the abstracts: (i) foundations of REST, which is related to the fundamentals of REST taking into account the impact on quality attributes; (ii) design of services, that is related to the design of operations, capabilities, data elements and other resources that will be exposed in a service interface; (iii) representation and identification, which is related to the formats and mechanisms used to represent and identify REST resources; (iv) documentation and testing, which is related to the documentation of REST resources with respect to identification, representation, security, and design; (v) behaviour, which is related to the service runtime behaviour when service consumers perform actions on resources, and; (vi) security, which is related to authentication, authorization, and privacy of REST services. These categories were used to classify the publications in our SM that were used as input to create the evaluation guide, complementary to the information extracted from the interviews. These six categories were later used to classify the design questions that are part of the architecture evaluation guide and are presented in Section 7.

4.2.5. Data extraction and mapping process

With the classification scheme, the relevant papers were sorted into the scheme. The result of this activity is shown in Table 3. The analysis of the studies focused on classifying relevant papers, books and technical reports that contributed to the information retrieved from interviews, relating them to their topics.

4.3. Evaluation guide creation

With the classification scheme, along with information obtained from the selected publications and the interviews, the evaluation guide was created in three parallel activities: (i) describe the architecture foundations of REST, presented in Section 5; (ii) craft general scenarios, which are presented in Section 6; and (iii) define the design questions and quality attribute discussion, which are presented in Section 7. Such activities correspond to the architecture evaluation concerns AEC1, AEC2, and AEC3, respectively.

4.4. Evaluation of the REST guide

Once the evaluation guide was created, validation was required to assess whether it achieved its goals, that is, whether the guide could indeed help AEs in their activities by addressing the *REST* AECs. To do so, we sent the guide to several architecture evaluation teams and professionals from SEI, Fraunhofer IESE, among others, and asked them to use the guide in their REST-based architecture evaluation activities. After that, based on a Likert Scale, we asked them to rate statements in order to reflect their perceived quality of the guide. The results are presented in Section 9.

5. Foundations of REST for architecture evaluation (evaluation concern AEC1)

One of the main goals of an architecture evaluation is to identify risks to address the quality attribute requirements in a software architecture. The next sections describe the foundations of REST with special attention to the impact in quality attributes. The foundations are based on the performed interviews and literature review.

5.1. REST constraints

Representational State Transfer (REST) is a software architectural style for distributed systems (Fielding, 2000). Roy Fielding has described six constraints that define the REST style, each of which promotes a different set of quality attributes. REST can be described as (1).

$$\text{REST} = (\text{C} - \text{S}, \text{S}, \$, \text{U}, \text{L}, \text{CoD}) \quad (1)$$

Each variable in expression (1) is a constraint. Next, we briefly describe each constraint along with respective impact on quality attributes and important points that evaluators should inspect to assess that impact. In Section 7 the constraints will be distilled into REST design questions.

5.1.1. Client–server (C-S, S, $, U, L, CoD)

Client–server is a frequently found architectural style for network-based applications. This style describes distributed systems that involve separate clients that request services from server components over a network connection through a request/reply connector. In REST, requests are initiated by *user agents* (clients) and ultimately processed by an *origin server* (server), which provides services through a resource hierarchy. REST has a strong focus on decoupling client and server; however, several REST-based architectures show

challenges to define which components will act either as a client or as a server. The main benefits of the client–server style are separation of responsibilities, independent evolution and maintainability. Evaluators should inspect the definition of the boundary between client and server according to cohesion and the independent evolution of each one.

5.1.2. Stateless (C-S, S, $, U, L, CoD)

The stateless constraint describes that all information needed to understand the conversation state data between *origin servers* and *user agents* must be included in the request and response messages. That means that no conversational state is kept on the server between requests; state is maintained, updated and communicated by the client. Important inspection points are related to server state data flow. The stateless constraint enables replication of servers and hence promotes availability, scalability and reliability; on the other hand, it may decrease performance due to the need for sending the conversational state data embedded in request and response messages.

5.1.3. Cache (C-S, S, $, U, L, CoD)

The cache constraint is added in order to improve performance. A cache element acts as a mediator between client and server. The objective is to avoid a request–response interaction when the information is present in the cache, thus avoiding network traffic. In multi-tier solutions (layered system constraint), a cache may also be present in intermediary tiers. Although cache is an ordinary feature of the World Wide Web, it is not always used in REST services that are part of SOA solutions. Evaluators should identify whether the client design will include a cache mechanism and what data elements (resources) should be cacheable. The degree to which the cache will increase network efficiency and hence performance, depends on the cache strategy. However, the use of a cache may decrease reliability in cases when the client may consume stale data from the cache.

5.1.4. Uniform interface (C-S, S, $, U, L, CoD)

Uniform interface across components is the central feature that distinguishes REST from other network-based styles (Fielding, 2000). Uniform interface is closely related to resources, identifiers and representations. A resource is any important concept in the system domain that we want to make accessible through a uniform interface. A service consumer addresses resources through a unique resource identifier (URI) and a representation. The representation of a resource refers to a hypermedia document that brings any useful information about the state of the resource and links to other associated resources. Finally, the representation of a resource must be accessed by a simple interface that defines an identification for the resource and common methods to access. This interface is the uniform interface. The uniform interface constraint positively impacts interoperability and discoverability.

5.1.5. Layered system (C-S, S, $, U, L, CoD)

The layered system constraint as described by Fielding is not exactly the more general layered pattern often found in the patterns literature. In fact, since 2000 the architectural pattern more easily identified with the REST layered constraint as described by Fielding has become known as *multi-tier*. Multi-tier is an architectural style that is a specialization of the client–server style (Clements et al., 2010), where an intermediary tier acts as server to the previous tier and as client to the subsequent tier. Fielding suggests that elements in a tier should not "see" beyond the next tier hence containing overall system complexity. Legacy systems, databases and backend systems in general are often enclosed in the rearmost tier. Elements in an intermediary tier can be load-balanced for improved availability, scalability, and system throughput. However, the deployment of components across several layers may increase the response time for a given request, due to the multi-hop communication across layers. Although simple REST services can be available on the "server-side" of client–server architectures, REST services are often found on "server-side" tiers of multi-tier applications developed using Java EE, .NET, or other implementation platform suitable for multi-tier applications.

5.1.6. Code-on-demand (C-S, S, $, U, L, CoD)

Code-on-demand is an optional constraint that enables a dynamic architecture where the user agent logic can be extended by code received from the service. Code can be for instance an applet or JavaScript function used in Web pages to render widgets or validate user input. As another example, consider a client that obtains from the server the code that is required to run in order to encrypt message payloads. Code-on-demand is a form of runtime variability since the functionality and behaviour of the user agent can change at runtime per the code received in response messages to a REST service. *CoD* promotes extensibility. On one side, the basic client implementation devoid of any code received on demand is lightweight, but on the other side, the complexity of this client implementation increases to handle the runtime code additions. Interoperability may decrease since the downloaded code has to be compatible among all service consumers. Security is also a concern to prevent malicious code to reach the clients.

5.2. Pragmatic REST

The constraints (C-S, S, $, U, L, CoD) we just described represent what we call "pure REST". They do not define a specific technology or platform for implementing REST-based systems. In recent years, REST has become a popular architectural style for designing and developing distributed systems. Several language-specific APIs, development frameworks and auxiliary tools have been created for designing, implementing and testing REST services. We call these solutions "pragmatic REST", because they focus on practical aspects of the development of services that follow the REST architecture style.

Pragmatic REST uses Internet technologies, primarily http and media types. The REST constraints (C-S, S, $, U, L, CoD) can be applied over http and related technologies to implement Web services that are called REST services or RESTful services (Richardson & Ruby, 2007). Architects, developers, and AEs of REST-based solutions should be familiarized particularly with the http protocol. Http is primarily specified in RFC 2616 (Fielding et al., 1999). The uniform interface constraint and its related concepts (resources, identifiers and representations) are very important to pragmatic REST and rely on http and other basic Internet technologies. That constraint distinguishes REST services from other Web-based services, such as SOAP services.

6. REST general quality attribute scenarios (evaluation concern AEC2)

Table 4 enumerates general quality attribute scenarios for 10 quality attributes that are important to consider when using the REST style, according to the interviewed experts (Section 4.1.2) and literature review (Section 4.2). The list of general scenarios is by no means exhaustive, but it should cover qualities and situations that are typical in REST-based solutions. For each general scenario, we present recommendations for the definition of concrete scenarios. We also indicate design questions that can help evaluators to identify risks in the architecture related to addressing that kind of scenario. In the next section, we explain where these questions come from and further discuss each question and the quality tradeoffs involved.

Scenarios differ greatly according to the quality attribute they refer to. The REST general scenarios have placeholders, such as resource R1, platform X, service B. Based on interviews with architect evaluators (Section 4.1), the process to derive a concrete scenario, represented using stimulus-response structure, from a general scenario, is

Table 4
General REST quality attribute scenarios, recommendations and design questions.

Quality attribute	General QA scenario	Recommendation for concrete scenarios	Design questions
Interoperability	**I1**—A service consumer 'A' created using development platform X requests a resource 'R1' using plain/text format from service 'B' created using development platform Y and correctly receives the representation of the actual state of 'R1' in the response message.	In concrete scenarios, development platforms X and Y should match the various platforms expected to be used to create REST services and clients in the project.	**A.1**—What types of service consumers will interact with the REST services? **D.8**—Does the REST service use code-on-demand?
	I2—A service consumer 'A' created using development platform X requests a resource 'R1' in a specific format (media type) from service 'B' created using development platform Y; service consumer 'A' is able to correctly parse the response that arrives in the specified format.	In concrete scenarios, different media types should be chosen according to the needs of the project.	**A.3**—Are resource representations standardized within the entire application, department, or enterprise? **B.1**—What format is used to represent resources?
	I3—A service consumer 'A' created using development platform X requests a resource 'R1' from service 'B' created using development platform Y and filling up the Authorization http header with the user credential; service 'B' correctly obtains the header information, validates the requests and responds to consumer 'A'.	Concrete scenarios should specify standard or custom http headers that are expected to be used for messaging metadata in the project.	**B.2**—Is a standardized vocabulary defined for the resource?
Reliability	**R1**—A service consumer 'A' requests a resource 'R1' in a specific version specified directly in the URI and receives the representation of the actual state of 'R1' in the response message using the specified representation version.	A concrete scenario should be created for a service that is expected to handle different resource representation versions over time.	**A.6**—What other components does the REST service depend on? **A.7**—How are services packaged and deployed? **B.4**—What is the approach for resource versioning?
	R2—Based on the domain model, a service consumer 'A' builds the URI of resource 'R1' and receives the valid XML representation of the actual state of 'R1' in response.	A concrete scenario can be created for services that return data in a somewhat complex XML (or JSON) representation that can benefit from schema validation.	**A.1**—What is the domain model of the application? **A.2**—What data will be exposed as resources? **B.3**—How do you design resources URIs?
Security	**S1—**A service consumer 'A' with insufficient privileges requests confidential information to a service interface 'X'; 'X' denies the request and informs 'A' about the lack of authorization via the 401 http error code in the response.	The concrete scenarios may exercise specific authentication/authorization situations and indicate other error codes as defined for the project.	**A.5**—Is confidential information exposed as a resource? **D.3**—Can the resource communicate with Open APIs? **D.8**- Does the REST service use code-on-demand? **E.1**—Did service design consider information classification?
	S2—An authenticated and authorized service consumer 'A' requests a confidential resource 'R1' which it has access to and receives the representation of the actual state of 'R1' in response.	For this "happy path" scenario, a concrete scenario can be built for a service that requires security validation and indicate the specific authorization levels that are required.	**A.4**—What types of service consumers will interact with the REST services? **D.8**—Does the REST service use code-on-demand? **E.2**—What are the security mechanisms for service consumers to perform actions on resources?
Testability	**T1**—A developer wants to test a service. If an error occurs when processing the request, the service can be configured to provide in the response all information to identify the error, including the execution trace.	Concrete scenarios may specify what pieces of information should be made available and how.	**C.2**—How service consumers can perform tests in resources?

(*continued on next page*)

Table 4 (*continued*)

Quality attribute	General QA scenario	Recommendation for concrete scenarios	Design questions
Performance	**P1—**A service consumer 'A' sends a request to service 'B' during a peak load period and receives the response in less than n milliseconds.	A concrete scenario should pick a service that has a strict response time requirement, and may further specify the conditions (environment).	**A.3**—Are resource representations standardized within the entire application, department, or enterprise? **A.6**—What other components does the REST service depend on? **D.4**—Is pagination in resources necessary? **D.5**—Is it possible to include the subset of desired attributes in the URI? **D.7**—Does the design of the service client use a cache mechanism? **D.9**—Is there replication of the REST service at runtime?
	P2—During peak periods, the system can process M requests per second sent to service 'B'.	A concrete throughput scenario should use services that are expected to receive a high number of requests and/or are part of many service compositions.	**D.9**—Is there replication of the REST service at runtime?
Availability	**Av1—**The Web server where the REST services run is flooded with a number of requests N percent higher than normal and the services remain responsive.	The concrete scenario may further specify the allowed decrease on response times.	**A.6**—What other components does the REST service depend on? **D.6**- How to protect the Web server from request overload?
	Av2—A Web container where a REST service runs is shut down for maintenance and the service remains available to service consumers.	The concrete scenario may indicate conditions and measures that are specific to the cluster or grid technology to be employed for service replication.	**A.6**—What other components does the REST service depend on?
Modifiability	**M1**—A developer modifies the core logic and internal data sources of a service, but the service contract (uniform interface, supported URIs and representations) remains the same; the effort to effect these changes is bound to N person-days.	The concrete scenario may indicate a specific modification that is expected in that project, such as the core logic being adapted to interact with a new backend system.	**A.3**—Are resource representations standardized within the entire application, department, or enterprise? **A.7**—How are services packaged and deployed?
	M2—The representation structure of a resource 'R1' and its relations to other resources change, and resource identification (URI) is not affected.	The concrete scenario may pick a specific resource which structure is expected to evolve in future releases.	**A.7**—How are services packaged and deployed? **B.3**—How do you design resources URIs?
	M3—The resource representation used in service 'B' is modified and the service correctly processes requests from service consumers that use the old version and consumers that use the new version of the resource under the same URI.	The concrete scenario should select a service that has external consumers or consumers that cannot be readily updated if the representation structure changes, thus requiring the service to handle requests to both old and new resource representation structures.	**B.4**—What is the approach for resource versioning?
Safety	**Sa1—**A service 'B' receives several parallel or serial requests to update a given resource to the same value X by using the idempotent method http PUT; the resource has value X after the first request is processed and the same value X at the end of the last request.	The concrete scenario should use a resource that may be updated by different consumers and update requests may occur in parallel.	**B.5**—How do you map operations on resources to http verbs?
	Sa2—A service consumer 'A' performs requests by using safe methods (such as http GET or OPTIONS) and the resource value is not modified.	The so-called "safe methods" in REST must not alter data	**B.5**—How do you map operations on resources to http verbs?

(*continued on next page*)

Table 4 (*continued*)

Quality attribute	General QA scenario	Recommendation for concrete scenarios	Design questions
Discoverability[a]	**D1**—The developer of a service consumer uses simple keywords to search a registry or catalogue and finds clear and up-to-date documentation about the service contract, and how to call the service.	To attain good discoverability, SOA environments rely on a service registry and associated governance rules. The registry itself typically is not part of the REST-based system itself, but is essential to achieve good levels of service reusability and composability. This scenario calls for an efficient service registry,	**C.1**—How are services documented in the registry?
	D2—Based on the format used in the design of resource URIs, a developer can intuitively build a service consumer 'A' to access different resource URIs.	This scenario addresses interpretability, which is an aspect of discoverability. Interpretability is related to the "communications quality" of the service interface and allows a developer to understand the purpose and how to call a service, once it is discovered out of a registry (Erl, 2008).	**B.3**—How do you design resources URIs? **B.5**—How do you map operations on resources to http verbs?
	D3—A service consumer 'A' requests a resource 'R1' that contains a non-fixed set of associated resources; the representation of 'R1' in the response message contains the URIs to access the associated resources.	This scenario will make sense for a service that should offer the consumer a mechanism to dynamically navigate and bind to other resources that are associated with the service.	**B.2**—Is a standardized vocabulary defined for the resource?
Functionality	**F1**—A service consumer 'A' calls service 'B' to create a resource and receives the URI of the new resource in the response.	A concrete scenario can be specified for a service that allows resource creation.	**D.2**—Does the response to a resource creation request contain the location of the new resource?
	F2—A service consumer 'A' needs a subset of a collection of resources or a subset of the attributes of a resource; it sends a request to service 'B' specifying a filter condition and receives a response that contains only the needed data.	A concrete scenario should be created for services that need to apply pagination on the response (to return only a limited number of resources in a collection) or a filter condition on the resources.	**D.4**—Is pagination in resources necessary? **D.5**—Is it possible to include the subset of desired attributes in the URI?
	F3—A service consumer 'A' wants to perform operations in a resource and 'A' can only use http primitives for that.	Http is flexible enough to allow the design of a service that only accepts http get requests and different operations and parameters are specified via query string parameters. Such design is not a proper application of the REST style. This scenario is a reminder that REST services should make use of the http primitives.	**B.3**—How do you design resources URIs? **B.5**—How do you map operations on resources to http verbs? **D.1**—What are the http status codes in responses? **D.3**—Can the resource communicate with Open APIs?
Deployability[b]	**De1—**The core logic of a service has been updated and the service can be packaged and deployed to the production runtime environment without the need to package or redeploy other services that are part of the software solution.	This scenario is applicable when the organization wants to stay away from the deployment model where many or all services are packaged and deployed as a single deployment file.	**A.7**—How are services packaged and deployed?

[a] *Discoverability* is commonly used in context of REST when talking about services that are supplemented with communicative metadata by which they can be effectively discovered and interpreted (Erl et al., 2012)
[b] *Deployability* indicates how easy it is to build, package, install (or update) a new version of a system on the host platform (Bass et al., 2012).

Table 5
Concrete REST quality attribute scenarios derived from general scenarios (stimulus-response structure).

Quality attribute	Concrete QA scenario
Interoperability (derived from scenario I1)	A service consumer 'MobileAPP' created using development platform .Net requests a resource 'Book' using plain/text format from service 'BookstoreAPI' created using development platform Java and correctly receives the representation of the actual state of 'Book' in the response message.
Performance (derived from scenario P2)	During peak periods, the system can process 10 requests per second sent to service 'StoreAPI'.
Modifiability (derived from scenario M2)	The representation structure of a resource 'Person' and its relations to other resources change, and resource identification (URI) is not affected.

Table 6
Concrete REST interoperability scenario derived from general scenarios (six-part structure).

Quality attribute	Concrete scenario
Interoperability (derived from scenario I1)	**(Source)** MobileAPP' (.Net platform) **(Stimulus)** MobileApp request a resource 'Book' **(Artefact)** BookstoreAPI's interface (Java Platform). **(Environment)** Normal operation **(Response)** The actual state of 'Book' in the response message **(Response Measure)** The BookstoreAPI responds to the MobileApp's request with HTTP status 200 (informing that the message was successfully processed) and in the HTTP body with the representation of the actual state of 'Book'

usually ad-hoc but involves two activities: (i) ask whether the situation described by the general scenario applies to the software solution being evaluated; (ii) if it does, replace the placeholders (if they exist) by concrete elements of the software being evaluated. Table 5 presents some concrete scenarios derived from general ones.

The process to derive concrete scenarios, described as six-part scenario structures (*source of stimulus, stimulus, environment, artefact, response*, and *response measure*), from general scenarios, is similar to the aforementioned derivation process. However, it requires more effort than only filling up the placeholders with concrete elements. The general scenarios, along with all the information presented in Table 4, should be used by the evaluators to derive the content of each one of the six fields of the concrete scenario. Table 6 shows an example of a concrete REST interoperability scenario derived from the respective general scenario.

7. REST design questions that affect quality attributes (evaluation concern AEC3)

This section presents a detailed discussion about the design questions listed in Table 4 that are relevant when creating or evaluating the design of REST-based systems. These questions were defined primarily based on interviews with specialists (Section 4.1) but also based on the literature review. Apart from the *Foundations of REST* (Section 5), the questions were grouped according to the other categories derived from the findings of the SM, namely: *Design of services* (Section 7.1); *Representation and identification* (Section 7.2); *Documentation and testing* (Section 7.3); *Behaviour* (Section 7.4); and *Security* (Section 7.5). In each topic, we present design questions that AEs can ask, and we relate the questions to the general scenarios presented in Section 7.

7.1. Design of services

Design of services is related to the design of operations, capabilities, data elements and other resources that will be exposed in a service interface. In addition to interface design, other design questions in this group are related to the service implementation, packaging and deployment. When evaluating service design, the following questions can help to identify risks:

7.1.1. What is the domain model of the application?

The domain model represents the knowledge about the domain of the application. It describes the various entities, their attributes, roles and relationships. The domain model can be represented, for example, by a UML class diagram. Evaluators should check if the domain model was validated by the application stakeholders. The reliability scenario R2 will be affected.

7.1.2. What data will be exposed as resources?

Evaluators should have an overall understanding of what data entities will be exposed as resources via REST services. The reliability scenario R2 will be affected.

7.1.3. Are resource representations standardized within the entire application, department, or enterprise?

The representation of a given resource should be the same in terms of format (e.g., XML, JSON) and structure across all services that deal with that resource. Ideally, this standardization should be enterprise-wide, but minimally it should span the entire software system. If a given resource has different representations across the system, interoperability and modifiability are impaired, for example, related to scenarios I2 and M1, respectively. Performance (scenario P1) may also be negatively affected due to the need for data format transformations (e.g., JSON to XML) or data model transformation (e.g., XSLT transformation from one schema definition to another).

7.1.4. What types of service consumers will interact with the REST services?

This question should elicit the nature of the software components that will act as consumers to REST services. They can be Web pages on a browser, a Java or .NET component running on a server machine, mobile apps, standalone applications, or any other kind of program that can communicate via http. Evaluators should also elicit the nature and properties of the communication channel. Are service consumers on the same local network or intranet where the REST service is? How fast and reliable is the connection? Is the communication channel encrypted? These questions impact interoperability scenario I1 and security scenario S2.

7.1.5. Is confidential information exposed as a resource?

General scenario S1 describes general security issues that have to be analysed when designing resources. Therefore, evaluators should know whether data confidentiality is a requirement for any REST service.

7.1.6. What other components does the REST service depend on?

Often a REST service needs to interact with an external database, other services, or other backend components when it is processing a request. These interactions impact the service autonomy (Erl, 2008) and may negatively affect performance (like scenario P1), reliability (like scenario R1), and availability (like scenarios Av1 and Av2). It is quite common for a service to compose other services or make database calls, and these interactions often do not pose a problem. Nonetheless, AEs should assess what is the risk of this decrease in autonomy to compromise the quality attribute requirements. There are SOA design patterns that address service autonomy issues. For example, the service data replication pattern (Newman, 2015) can help when the service needs to interact with a data repository that is too distant or offers poor connectivity.

7.1.7. How are services packaged and deployed?

REST services, like many other types of components, are packaged and then deployed to an execution environment, typically a Web server. Packaging can be done in many different ways and an important factor is how many and which services should be placed together in the same deployment artefact. At one end of the spectrum of possibilities, you can package all services of your SOA solution as a single deployment artefact. This approach is often called the monolithic deployment model. At the other end of the spectrum, you can have one REST service per deployment artefact. This granular approach for deployment is a key characteristic of microservice architectures (Lewis and Fowler, 2014). Evidently there are possibilities in between the two ends of the spectrum. For instance, you can still have granular deployment artefacts but instead of one service you have a few per deployment artefact. The grouping of services can be dictated by cohesion in terms of their functional context. For example, in a Web store, the service for placing an order is packaged together with the service for querying past orders.

Microservices have recently become a trending topic in the SOA industry. However, it is not a good alternative for all cases and, again, in an architecture evaluation we need to understand the tradeoffs involved. With respect to deployment, microservices (packaging one or only a few services per deployment artefact) have the following benefits over the monolithic model: (i) increased agility to deploy new versions due to shorter building, testing, and deployment cycles; (ii) increased modifiability (as in scenarios M1 and M2) because a service can more easily be adapted to work with a new database, library, or framework independent of other services and the rest of the application that is packaged separately; (iii) increased reliability (scenarios R1 and R2) because, on one hand, a service will not be affected by failures in other independent parts of the application, and, on the other hand, service faults such as resource leaks at runtime will affect that service alone—in comparison, one misbehaving service deployed in a monolithic deployment artefact may bring down the entire monolith (Richardson, 2014); (iv) improved overall availability of the application because rolling out a new version of a service requires short downtime of that service alone—in the monolithic approach, typically the entire monolith has to be restarted to update a single service; (v) more flexibility to employ replication, security, and monitoring mechanisms tailored for individual services—for example, a critical service can be replicated to more server machines to improve scalability and availability. Another benefit of microservices that is not directly related to quality attributes of the application is that microservices give project managers more flexibility to break up the development effort into small teams. Each team is responsible for one or more microservices and can work more independently than in a scenario where all the software is bundled together as a monolith.

In comparison with the monolithic deployment model, microservices may show some disadvantages and challenges: (i) performance (as in scenario P1) may be negatively affected since microservices need to communicate over the network whereas services within the monolith can communicate via local calls or benefit from other optimizations; (ii) the deployment of a collection of microservices yield a large number of deployment jobs, scripts, transfer areas, configuration files, and other elements of the deployment infrastructure, resulting in significant increase to operations complexity; (iii) in the monolithic model changes to the service contract may be a lesser issue because all services and consumers within the monolith move forward in lockstep whereas with microservices contract changes require a deployment plan that accounts for dependencies between services within the same software solution; (iv) a microservice may interact with other independently deployed microservices, so if not modelled carefully, this interaction may require a distributed transaction, which may compromise reliability, in turn, services that interact within a monolith are deployed together and hence should share database connections and not need distributed transactions; (v) services in the monolith share classes and libraries, whereas for microservices these classes and libraries may need to be replicated, increasing the overall footprint and total size of deployed artefacts; (vi) when operating on slightly out-of-date data is acceptable, a dedicated copy of the data a service requires may be created to avoid distributed transactions or simply to improve the service autonomy (this approach is the service data replication pattern (Newman, 2015) mentioned in the previous design question, which improves service autonomy but increases the overall complexity of the solution because a mechanism to synchronize the dedicated copy and the master database needs to be in place); (vii) to allow microservices to be deployed or moved to different locations more easily, a service registry may be used for service discovery and dynamic binding; services deployed together in the monolith are less likely to require dynamic binding, in which case service consumers avoid the performance overhead of interacting with the registry prior to calling the service; (viii) testing distributed components is more challenging than testing components that are co-located, so microservices may increase the complexity of automated integration tests; (ix) microservices also require extra effort for runtime monitoring because there are more independent runtime components to oversee, more log files to sift through, more components and point-to-point interactions that can present latency and reliability issues (Newman, 2015).

To minimize the drawbacks of microservices, the architect needs to carefully select which services should be deployed together and find the right balance for the granularity of the deployment artefacts. An approach that has been advocated to guide the modelling of services all the way to deployment as microservices is Domain-Driven Design (Newman, 2015).

7.2. Representation and identification

The following questions related to the representation and identification of resources can help to spot risks in the design of the REST service contracts:

7.2.1. What format is used to represent resources?

A REST resource is represented by a document that uses a given data format that is identified by a media type (such as application/json), and declared in Content-Type http headers. XML (application/xml) is a very common representation format for resources, although other formats, such as HTML, ATOM, and JSON, are also widely used. Development frameworks and platforms provide different levels of support for different formats. The choice of representation format also affects performance and reliability, as these formats need to be parsed, validated and sometimes transformed on both service consumer and provider ends. To address interoperability quality attribute scenarios (like I2), a REST service may need to use different representations of resources suitable for different devices and systems.

7.2.2. Is a standardized vocabulary defined for the resource?

A resource representation should follow a predefined vocabulary. For XML documents, the vocabulary can be defined by an XML schema. The vocabulary should also be standardized across all services that handle that same resource, as previously discussed in question A.3. Using a vocabulary known by service consumers improves interoperability scenario I3. Also, all information needed to understand the resource must be included in the request and response messages (constraint Stateless, session IV). The HATEOAS (Hypermedia as the Engine of Application State) constraint (Fielding, 2000) is used for this purpose. This constraint states that the hypermedia document that represents the resource should be used to find associated resources. For instance, the following fragment of the representation of resource "course" from a university contains a link to another resource representing the lecturers who teach the course:

```
<course>
  < name > Software Architecture < /name >
  < area > Software Engineering < /area >
  < instructors >
    < uri > /courses/sa/lecturers < /uri >
  < /instructors >
< /course >
```

HATEOAS positively affects discoverability scenario D3. The response time for a service request decreases because the representation does not need to bring all information about the linked resources, but only references (hyperlinks). However, a greater number of network requests may be necessary to retrieve linked information and the overall performance may be negatively impacted. The resource design should balance how much information is embedded and how much information is only linked to in the response message.

7.2.3. How do you design resources URIs?

The URI is a string of characters used to uniquely identify a Web resource. A resource must have at least one URI that makes it addressable via http. Generic principles for designing URIs (Berners-Lee, 1996) can be used for the specification of REST URIs, but there are other resource URI design guidelines that can improve quality aspects of the REST service.

Designing descriptive URIs that are related to the hierarchy of the domain model improves reliability scenario R2. Examples of descriptive URIs are listed below:

- http://www.myweather.com/current/city/Brasilia
- http://www.ufrj.edu/courses/programming101

However, URIs are not always descriptive. An example of URI where a resource is identified by a numeric ID (52545) without a qualifier (descriptor) is http://www.resouceslib.com/52545. It is not clear what piece of information does ID 52545 refer to. Classes, modules, methods, operations, parameters, variables in traditional components should have meaningful, intention-revealing names (Martin, 2009). Likewise, REST URIs should use meaningful words because the URI is an important part of the REST service interface. Developers should be able not only to read but to guess the URIs for resources of similar types.

Evaluators should inspect the tradeoffs related to the strategy used for resource URI design. Resource identification by ID positively impacts the modifiability quality attribute scenario M2, but impacts negatively in reliability scenario R2. Establishing a pattern to define URIs with meaningful names is a good practice that positively impacts the discoverability scenario D2.

In general, URIs should use nouns, not verbs. According to the uniform interface constraint, the operation to be executed should be specified by the http verb, not in the URI. For example, instead of URI http://www.ufrj.com/getcourses, we should use http get on URI http://www.ufrj.com/courses. If verbs are used in the URI, functionality, as in scenario F3, could be negatively impacted.

7.2.4. What is the approach for resource versioning?

Representations of resources are accessed by their identification (URI). If a representation is modified, service consumers that request that resource can be negatively impacted. The reason for such modification can be evolution of the domain model that impacts in the resource representation. Versioning of resources is a good practice to solve this problem that positively affects the modifiability scenario M3. A commonly used strategy to version resources is to include a version number within the URL. For instance:

- http://www.ufrj.com/courses/v1/comperscience

Another alternative is to include the version in the http header, for example: "Accept: *Content-type: application/xml;* ***version = 1.0***". Evaluators should inspect the tradeoff between URI versioning and http header versioning. For example, for human users requesting resources via Web browsers, URI versioning may be a better solution and, in this case, reliability quality attribute scenario R1 is positively impacted.

7.2.5. How do you map operations on resources to http verbs?

This question is related to question A.2. As per the uniform interface constraint (Section 4), the interface must use common http methods (also called verbs) to indicate the action to be performed on a resource. Because the http methods allow some flexibility—for example, PUT or POST can be used to create a resource—the architecture should make clear which methods to use in each situation. Using http verbs consistently across services promotes discoverability scenarios like D2 and functionality scenarios like F3.

Important concepts when designing the resource exposure are idempotent and safe methods. Safe methods are http methods that do not modify resources (for instance GET and OPTIONS); they only request information. An idempotent http method is a method that can be called many times without different outcomes. For instance, consider two actions that assign a value to variable *var*: (a) var = 5; (b) var++. The first example (a) is idempotent; no matter how many times the method is executed the result will always be 5. The second example (b) is not idempotent. Different number of execution results in different outcomes. Http GET, HEAD, OPTIONS, PUT and DELETE should be idempotent. Safety quality attribute scenarios Sa1 and Sa2 are related to idempotent and safe methods.

7.3. Documentation and testing

Evaluators should inspect the documentation of REST services regarding resource identification, representation, security and design. Developers of service consumers need to understand the service interface. When evaluating documentation and tests, the following questions help determining risks:

7.3.1. How are services documented in the registry?

Rolling out a nicely modelled and implemented service is not enough for the service to be repeatedly reused and composed. Metadata (documentation) about the service must be added to a service registry or catalogue, so that developers searching the registry can easily locate the service and understand how to call it. Service discovery through a registry is especially important for highly reusable services that can be invoked by various corporate and/or external service consumers. Effective service interface documentation in the registry positively impacts discoverability scenario D1.

7.3.2. How service consumers can perform tests in resources?

It is important for service consumers and developers to perform tests in the service interface. Dynamic tests can be executed using a "service sandbox". Http methods like GET and PUT can be executed against the sandbox environment. This is a typical scenario to improve the testability quality attribute scenario T1.

7.4. Behaviour

Behaviour is related to the service runtime behaviour when service consumers perform actions on resources. When evaluating behavioural considerations, the following questions help determining risks:

7.4.1. What are the http status codes in responses?

REST services should use standard http status codes to report success and error when processing requests. Evaluators should ask about the use of standard http status codes. Their use contributes to functionality scenarios, such as F3.

7.4.2. Does the response to a resource creation request contain the location of the new resource?

In some transactions, a service consumer calls a service to create a resource and then performs actions on the newly created resource. In that case, the service implementation should include in the http responses the header field "Location" filled with the URI of the new resource. Service consumers can correlate the newly created resource with the URI in the Location header of the response message. This design solution directly addresses functionality scenarios like F1.

7.4.3. Can the resource communicate with Open APIs?

Open API (Musser, 2008) is a common trend for Web-based services in social networks and e-business, to publish public services over the Internet. If a service interface allows the use of Open APIs, functionality (for instance, scenario F3) is positively impacted. However, the tradeoff between functionality and security (as in scenario S1) should be considered.

7.4.4. Is pagination in resources necessary?

A service may return a collection of resources. Transmitting the entire collection in the response may negatively impact performance scenarios like P1. Evaluators should assess if pagination of the resources is needed. A good practice is to use query string parameters or the "Range" standard http header to specify page number or page range, resource count per page, ordering argument, and other pagination criteria. For example, a URI for "courses at UFRJ" that specifies the visualization for page one with up to 30 courses per page could be: http://www.ufrj.com/courses?page=1&size=30. Pagination is related with performance and functionality scenarios, such as P1 and F2.

7.4.5. Is it possible to include the subset of desired attributes in the URI?

A service consumer may only need a subset of the attributes of a lengthy resource representation. If the service interface only supports responses containing the full representation of the resource, performance and functionality scenarios like P1 and F2 are negatively impacted. In that situation, a good practice is to include a filter condition in the URI, possibly using query string parameters. For example, the following URI will retrieve only the name and number of credits for courses at UFRJ: http://www.ufrj.com/courses?attributes=name,credits.

7.4.6. How to protect the Web server from request overload?

Service consumers can perform a huge number of requests resulting in Web server overload. Evaluators should ask about mechanisms in place to prevent request overload, and hence promote availability, as in scenario Av1.

7.4.7. Does the design of the service client use a cache mechanism?

REST-based architectures focus on the REST services, so most architecture decisions are related to the design of the service interface, service implementation, and service compositions. However, in solutions where the service consumer (client) is within the scope of the software architecture, a design question about the use of cache may be relevant. Cache is one of the original REST constraints discussed in Section 4. It is pervasive in the World Wide Web since Web browsers in general provide a caching feature. However, for service consumer to service interaction in SOA solutions, the use of cache is not as common. If it is used, AEs should be aware of the tradeoffs involved, some of which are discussed in Section 4.1. The decision about cache mechanisms affects the performance scenario P1.

7.4.8. Does the REST service use code-on-demand?

Code-on-demand is another original REST constraint that may be used by services to deliver program logic to service consumers. When code-on-demand is applied, the service consumer is required to embed a virtual machine or engine to run the code that comes in service response messages. As discussed in Section 4.1, this requirement may negatively impact interoperability (like scenario I1). Security is another concern related to scenarios S1 and S2, since there is the risk of malicious code being delivered to the service consumer.

7.4.9. Is there replication of the REST service at runtime?

It is not unusual for components that process a high load of http requests to be replicated. The runtime environment usually offers replication mechanisms, such as clusters of Web servers and grids. REST services in general are well suited to benefit from these mechanisms, since they should be stateless. Section 4.1 describes the Stateless REST constraint and the quality tradeoffs related to service replication. The decision of services replication positively impacts on performance scenarios P1 and P2.

7.5. Security

Design questions about security are related to authentication, authorization, and privacy. When evaluating security, the following questions help determining risks:

7.5.1. Did service design consider information classification?

This question is associated with design question A.2 (*What data will be exposed as resources?*). Evaluators should pay special attention to the risk of exposing private information in public resources. Different parts of a resource may have different security classifications. Such situation may impact granularity of the services (e.g., breaking up the resource into two, one with public information and the other with private information). Scenarios such as S1 are affected.

7.5.2. What are the security mechanisms for service consumers to perform actions on resources?

Some resources are restricted to specific groups of service consumers. Security mechanism to ensure authentication and authorization of service consumers may be required. Some commonly adopted standards that can be employed for REST service security are OAuth (OAuth Community Site, 2015) and OpenID (OpenID Foundation, 2015). Http Basic Authentication can also be used. A typical scenario related to this design question is S2.

8. Evaluation proof of concept

The goal of this section is to show the use of the proposed evaluation guide in a proof of concept system evaluation. The scenario-based method employed is the ATAM (Kazman et al., 1998) and it was applied to a Web-based system, briefly described as follows.

8.1. Usage of the guide in ATAM

The general scenarios proposed in this paper can be used in different activities of ATAM. Our proof of concept focused on the following activities: 3 (Present architecture), described in Section 8.3;

4 (Identify architectural approaches), which is descripted in Section 8.4; concrete scenarios of activity 5 (Generate quality attribute utility tree), described in Section 8.5, and on the use of the evaluation guide in light of activity 6 (Analyse architectural approaches), which is described in Section 8.6.

8.2. System description

The system used to evaluate the guidelines proposed in this paper was EcoDiF (*Web Ecosystem of Physical Devices*) (Delicato et al., 2013). EcoDiF is a middleware platform for the internet of things (IoT; Atzori et al., 2010) that integrates heterogeneous devices to provide real-time data control, visualization, processing, and storage. In EcoDiF, devices, information, users and applications are integrated to create an IoT ecosystem in which new ideas and products can be developed in an organic way. IoT devices are typically heterogeneous, ranging from household appliances to sophisticated cell phones. They encompass applications built using different programing languages, running over distinguished operating systems and on different hardware platforms. Therefore, interoperability is major quality attribute requirement for EcoDiF, to deal with such a degree of heterogeneity. Thus, the architecture was designed according to REST principles and constraints. EcoDiF is available at http://ubicomp.nce.ufrj.br/ecodif/index_en.html. Fig. 4 shows an overview of EcoDiF, illustrating the integration of heterogeneous physical devices and the use of data provided by different kinds of users.

EcoDiF has four types of stakeholders: (i) *devices manufacturers*, which develop drivers to their devices to make them compliant with the EcoDiF Open API, as well as data profiles that specify the metadata that describes the type of data provided by the their devices; (ii) *data providers*, which are device owners that want to make the data produced by their devices available at the IoT ecosystem through EcoDiF; (iii) *application developers*, which build Web applications or services that use as inputs the data available at EcoDiF as well data produced by any other Web accessible resource, and; (iv) *information consumers*, which are users that interact with the EcoDiF ecosystem.

8.3. EcoDiF architecture overview

Fig. 4 also shows the main modules in EcoDiF. The *Devices Connection module* enables the connection of physical devices to EcoDiF and, consequently, to the Internet. Manufacturers configure their devices according to the EcoDiF interface to enable the integration with the platform. Users connect their preconfigured devices to perform the operations of the provided interface. This connection between the device and EcoDiF is enabled by a customized driver to the specific device, so that data providers can use such driver for connecting their devices and make the data obtained from the devices available at EcoDIF. Data messages (called feeds) are represented using EEML (*Extended Environments Markup Language*) (OMA EMML, 2013), an XML-based language that describes data obtained from devices in a specific context. The data are sent by the device to EcoDiF through a driver through http PUT, and users can manipulate the data through EcoDiF *Data Manipulation Component*.

The *Visualization and Management module* provides a Web interface for users managing the devices connected to EcoDiF. These users can monitor the state and location of their devices, create alerts and notifications (*triggers*) about the environment, and visualize historical data. The *Collaboration module* allows EcoDiF users to perform searches for devices and applications registered with the platform based on their metadata (type, user, location, etc.). The *Storage module* consists of two repositories, a relational database for the data feeds, and a repository for *storing application* scripts in a file system.

Finally, the *Applications module* provides a model and environment for programming and executing applications that can use the data (feeds) available at EcoDiF and generate new information to be

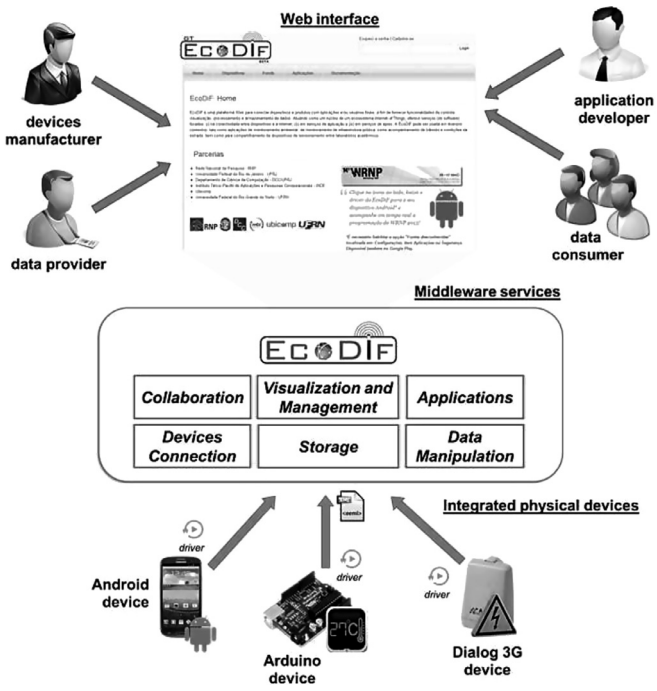


Fig. 4. EcoDiF architecture.

available at the platform. In EcoDiF, these applications are built as Web *mashups* (Guinard et al., 2009). The EMML (*Enterprise Mashup Markup Language*) (Extended Environments Markup Language, 2008) XML-based language is used to develop Web mashup applications by integrating data from several sources, including Web services, third-party APIs, and relational databases.

8.4. Architectural approaches of EcoDiF

The layered system (*L*) constraint is applied by grouping responsibilities in three tiers (Fig. 4): Web interface, middleware services, and integrated physical devices. EcoDiF acts as a Web server and physical devices act as clients (see Section 4). The stateless (*S*) constraint is applied by not keeping state of requests; all resource information is included in the response. It does not have HATEOAS. Code on demand (*CoD*) and cache (*$*) constraints are not applied. Resources were designed according to the EEML protocol. The only resource exposed to clients is a feed. When a user provider creates a feed, EcoDiF assigns an ID to uniquely identify the feed. The URI pattern for identifying the feed is: ecodif.com / api / feed / [ID]. The http method used to request a feed is GET. The same URI is used by user providers to update feeds using http PUT. When a feed is updated, EcoDiF responds with http status code 201, but does not include the URI of the updated feed. To perform a feed update by using PUT, the user credentials must be added as an http header. User providers get their credentials upon registration. If the user provider configures the feed as public, GET is performed without security validations. If the feed is private, only the creator can request the feed representation. The feed uses the application/xml media type. When a resource is requested, EcoDiF sends the full representation of a feed to the client. EcoDiF does not have versioning of resource URI. The EcoDiF interface documentation is available on the Web (http://ubicomp.nce.ufrj.br/ecodif/index_en.html).

8.5. Concrete scenarios

Table 7 shows three quality attribute scenarios for EcoDiF that are relevant to a REST-based architecture. Such concrete scenarios were

Table 7
Concrete scenarios for the EcoDiF.

Quality attribute	Concrete scenario
Scenario 1. Interoperability	**(Source)** EcoDif's driver/data provider **(Stimulus)** Provider sends data to EcoDiF platform **(Artefact)** Driver/EcoDiF's interface **(Environment)** Normal operation **(Response)** Sensor data is sent and Driver receives the confirmation and actual state of sensor object from EcoDiF's interface **(Response Measure)** The system responds to the provider's request with HTTP status 200 (informing that the message was successfully processed) and in the HTTP body with the representation of the actual state of sensor (including the last data sent).
Scenario 2. Interoperability	**(Source)** Data consumer **(Stimulus)** Data consumer requests feed data **(Artefact)** EcoDiF's interface **(Environment)** Normal operation **(Response)** The system sends to the user the resource representation of feed **(Response Measure)** The systems is able to correctly parse the response that arrives in the format specified in the request.
Scenario 3. Discoverability	**(Source)** Devices manufacture **(Stimulus)** Devices manufacture wants to create a new driver and perform tests in EcoDiF's API **(Artefact)** Devices connection **(Environment)** Normal operation **(Response)** The system offers documentation for driver's development **(Response Measure)** The devices manufacture can understand the documentation that is clear and up-to-date about the service contract, and how to call the service.

Table 8
Analysis for concrete scenario 1.

Scenario summary	Sensor data are sent to EcoDiF by the driver. System updates the feed and responds to the Driver in less than five seconds
Business goal(s)	Allow communication of heterogeneous devices with EcoDiF
Quality attribute	Interoperability, Availability, Performance, Functionality
Architectural Analysis	- Domain-based vocabulary (EEML) promotes Interoperability. (**see subsection B—Representation and identification, question 2)** - By using only XML media type interoperability is negatively impacted (**see subsection B—Representation and identification, question 1)** - The URI is designed by feed code. It promotes extensibility but negatively impacts in interoperability (**see subsection B—Representation and identification, question 3**) - The method used to update resources is PUT and it is idempotent. It promotes Reliability. (**see subsection B—Representation and identification, question 5**) - EcoDiF's interface responds with http code 201 but does not include the feed URI in response. (**See subsection D—Behaviour, question 2**)
Risks	- Drivers that do not use XML could not send information to EcoDiF - The design does not respond with the resource URI in header Location
Tradeoffs	- The use of different media types promotes interoperability but adds complexity to implementation and validation.

conceived in light of general scenarios presented in Section 6 and are described as six-form structures. The complete analysis of EcoDiF's architecture, including all elicited concrete scenarios, can be found at: http://ubicomp.nce.ufrj.br/restguidance/proof.html.

8.6. Architectural analysis

The architecture approaches were probed by using guidelines referenced in Section 7 for each scenario. In Tables 8 and 9 we present the result of the analysis of interoperability concrete scenarios 1 and 2, respectively. In Table 10 the analysis of discoverability concrete scenario 3 is presented. In interoperability scenarios, the architecture was analysed in light of the design questions: A.1 (What types of service consumers will interact with the REST services?); D.8 (Does the REST service use code-on-demand?); A.3 (Are resource representations standardized within the entire application, department, or enterprise?); and B.1 (What format is used to represent resources?). The testability scenario was analysed according to the design question C.2 (How service consumers can perform tests in resources?). Based on the details of each design question, it was possible to analyse the architectural risks and tradeoffs with respect to multiple competing quality attributes.

9. Evaluation of the REST guide

In order to analyse if the guidelines achieved their goals, and collect the criticisms and suggestions from experts, the guide was sent to several architecture evaluation teams (from Software Engineering Institute, Fraunhofer IESE, among others). The goal was to have the guide used in real architecture evaluations performed in REST-based systems. After that, based on a Likert Scale (Dukes, 2014), we asked them to rate five statements scaling from 0 (strongly disagree) to 4 (strongly agree), with 2 being neutral. The statements were created to gauge how much the *REST* AECs were addressed in the guide. Statements 1 and 2 were based on AEC1 ("Explain the architectural foundations of REST from an AE's point of view"); the statement 3 was based on AEC2 ("Discuss quality attributes and general scenarios that benefit from REST"), and statements 4 and 5 were created based on AEC3 ("Discuss in detail how REST contributes to the quality attributes and where typical tradeoffs are").

Altogether, 17 AE teams were invited to use the guide and provide us feedback, but only six answered the survey and rated the statements. As the participation was anonymous, we could not contact the evaluators that did not answer the survey. Table 11 shows the characterization of the AEs that answered the survey and Table 12 shows the provided ratings, followed by their analysis.

The first activity of the analysis was to look into the reliability of the data. Based on correlations between different items, internal consistency was the measure used to check the reliability of the ratings. The objective was checking the similarity between the evaluators' ratings. In order to analyse the internal consistency, Cronbach's alpha coefficient was applied (DeVellis, 2003). Such coefficient is defined as (2).

$$\propto = k/k - 1\left(1 - \sum_{i=1}^{K} \sigma_{Y_i}^2/\sigma_X^2\right) \quad (2)$$

where K is the number of statements (5), σ_X^2 the variance of the observed total test scores (4.067), and $\sigma_{Y_i}^2$ the variance of statement i for the current sample of persons (3.08). Nunnaly (1978) has

Table 9
Analysis for scenario 2.

Scenario summary	Consumer requests information about some feed and receives it in less than 1 second with the correct media type requested.
Business goal(s)	Allow easy visualization of feeds
Quality attribute	Interoperability, Availability, Functionality, Adaptability
Architectural analysis	- The URI is designed by the feed code. It promotes extensibility but negatively impacts usability (**see subsection B—Representation and identification, question 3**) - It is not possible to explicit the media type in requests. Interoperability is negatively impacted - It is not possible to explicit the version of requested resources. Adaptability is negatively impacted (**see subsection B—Representation and identification, question 4**) - It is not possible to include requested attributes in the URI. Performance and usability is negatively impacted (**see subsection D—Behaviour, question 5**) - Resources are not linked to others. Navigability is negatively impacted (**See subsection B - Representation and identification, question 2**)
Risks	- The design does not meet the requirement in this scenario. The system can only produces XML responses; it is not possible to specify neither the version nor desired fields in the resource URI; HATEOAS is not used, thus the service consumers cannot navigate to other associated resources.
Tradeoffs	- Adaptability is promoted when using versioning of resources, but it impacts in the configuration management of resources - Expliciting fields in request improve performance, but adds complexity to implement filtering in the service interface

Table 10
Analysis for scenario 3.

Scenario summary	Devices manufacture needs to create a new driver and perform tests in EcoDiF's API.
Business goal(s)	Facilitates the development of Drivers
Quality attribute	Discoverability, Interoperability, Testability
Architectural analysis	- EcoDiF's interface documentation offers information in text documents. (**see subsection C—Documentation and testing, question 1**) - It does not have a sandbox for testing the drivers (**see subsection C—Documentation and testing, question 2**)
Risks	-Without a sandbox for testing, device manufacturers cannot analyse the response behaviour or error messages before deliver the driver for data providers.
Tradeoffs	-Developing a sandbox is important to improve development and testing of new Drivers. However, the sandbox approach negatively impacts modifiability since modifications in service interface must be done in the sandbox as well.

Table 11
Architecture evaluators characterization.

	Experience in software architecture evaluation	Use of a guide (such as Bianco et al. (2007)) to assist the activities in architecture evaluation	Number of evaluations of software architectures that use REST concepts
AE1	3–4 years	Yes	1–2 architecture evaluations
AE2	More than 7 years	Yes	More than 7 architecture evaluations
AE3	1–2 years	Yes	1–2 architecture evaluations
AE4	5–6 years	Never	3–4 architecture evaluations
AE5	1–2 years	Never	1–2 architecture evaluations
AE6	More than 7 years	Yes	3–4 architecture evaluations

Table 12
Rating of statements on the evaluation of the REST guide (from 0—strongly disagree to 4—strongly agree).

AEC	Statement/architecture evaluator (AE)	AE1	AE2	AE3	AE4	AE5	**AE6**	Variance	Average
AEC1	1. The REST guide helped me to understand better about REST principles	4	2	2	2	4	2	1.067	2,7
	2. The design questions presented in the REST guide are helpful to reason about the REST architecture decisions	4	4	3	2	3	3	0.567	3,2
AEC2	3. The general scenarios proposed in the guide are useful to generate the quality attribute utility tree	3	2	1	1	3	3	0.967	2,2
AEC3	4. The REST guide is useful to analyse the architectural approaches	3	4	3	3	3	2	0.4	3,0
	5. With the REST guide I could identify the risks of the architecture better than without its using	4	3	2	3	3	1	1.067	2,7
	Total	18	15	11	11	16	11	4.067	

indicated the upper value of 0.7 to be an acceptable reliability coefficient. Applying the Cronbach's alpha in evaluators' ratings, the result was 0.713, indicating acceptable reliability of the data.

The statements were created in such a way that it would be possible to indicate whether the AECs were addressed in the evaluation guide, based on the respondents' ratings. Fig. 5 depicts a diverging stacked bar chart showing the ratings of all participants (Heiberger & Robbins, 2014) grouped by AECs. In the next paragraphs, we provide some analysis about the results.

Starting the analysis with AEC1, more than half of participants (67%) were neutral regarding to the effectiveness of the guidelines to provide better understanding about the REST guide. We believe that this result is due to the fact that all participants were already familiar with the REST architectural style and had worked with it. Therefore, the information provided by the guide did not bring many new ideas about REST to the majority of the participants. Regarding the second statement related to AEC1, 83% of participants agreed or strongly agreed that the design questions were helpful in architecture activities, corroborating to our belief that the guide is a useful tool to aid architects in their evaluation tasks.

AEC2 ("Discuss quality attributes and general scenarios impacted by REST principles") had the worst score in the expert evaluation: only 50% of the participants agreed that the general scenarios proposed in the guide are useful to generate the quality attribute utility

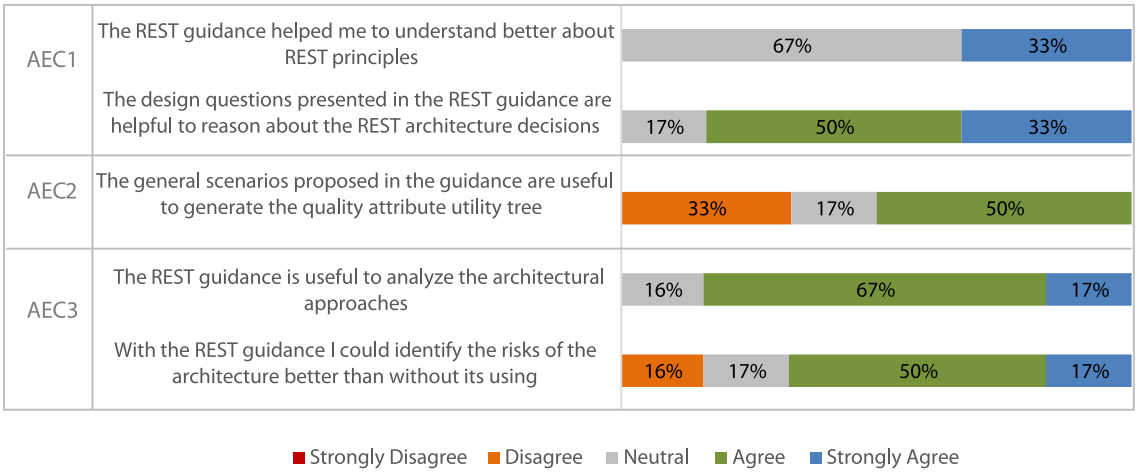


Fig. 5. Respondents' ratings about each AEC in the evaluation guide.

tree, while 33% disagreed and 17% were neutral. We believe the reason for this result is the fact that the guide fails to provide clear guidance to convert REST quality attribute general scenarios to concrete ones. The guide does not provide a systematic approach to derive concrete scenarios from general ones, leaving to the expert the responsibility to carry on this task based on his/her experience. The worst scores for AEC2 come from AE3 and AE4 participants. AE3 participant has few years of experience in architecture evaluation (1–2 years); in turn, AE4 participant has never used an evaluation guide to assist their evaluation activities. We believe that the limited experience in evaluation activities and the usage of evaluation guidelines could make extracting a utility tree from general scenarios a difficult task.

The best result is related to ACE3. For the first statement, 84% of the participants agreed or strongly agreed that the guide was useful to analyse the architectural approaches. In turn, 67% of the respondents agreed or strongly agreed that the guide helped them to identify risks of the architecture more easily than without using it.

Besides the ratings, the survey asked the respondents to provide criticisms and suggestions about the entire guide. The main criticism was related to the difficulty to manipulate the guide. As it is divided in three parts (REST foundations, generic scenarios, and design questions) and each part refers to another in many parts of the text, the reading becomes confused as the evaluator must go backward or forward to find the referenced information. For example, in Section 7, item B, question 1 describes: "(…) *The choice of representation format also affects performance and reliability. To achieve interoperability quality attribute scenarios like* ***I2***", so, it is necessary for the evaluator stop reading and find the description of the scenario. In order to mitigate this problem, we developed a Web tool that dynamically presents such references whenever they appear in the text. In addition, a common suggestion was related to a creation of a summarized version of the guide or a *cheat sheet.*[1] Such short version would present just the scenarios (Table 4) and their impact on quality attributes. In the Web tool, such a suggestion was also addressed. In Section 10, we describe the Web tool that was implemented to support the use of the proposed guide.

9.1. Threats to validity

In this section, we describe the different threats that could affect the validity of the evaluation. The discussed threats affect the conclusion, construct, internal, and external validity of the results. Overall, we tried to address the possible threats to validity through the careful planning of the evaluation.

9.1.1. Conclusion validity

Conclusion validity refers to issues that affect the ability of the evaluation to generate correct conclusion (Easterbrook et al., 2008). The conclusion validity threats were mitigated by the evaluation design where we develop the statements of Likert Scale associated with the AEC. Additionally, we use Cronbach's alpha coefficient in order to check the reliability of the ratings and statistics to analyse the results. However, as the conclusion validity can be affected by the number of observation, further replications on a larger dataset are required to confirm or contradict the achieved results.

9.1.2. Construct validity

Construct validity refers to the correct interpretation and measurement of the theoretical concepts (Easterbrook et al., 2008). Construct validity threats may be present in this evaluation were addressed by using a fairly simple and standard design. Furthermore, the statements of the Likert Scale were developed in order to be not ambiguous and clear to the respondents.

9.1.3. Internal validity

The objective of internal validity is to ensure that the collected data enables researchers to draw valid conclusions (Creswell, 2003). It is related to how the experiments are conducted, and the credibility of the used methods. The main threat to our study is the small number of participants in the evaluation of the REST guide, as reported in Section 8. Probably one of the reasons for the small number of participants is that in order to evaluate the guide, it would be ideally necessary using it in a real-world architecture evaluation. The evaluation of the guide would represent an additional effort to architecture evaluation teams. Moreover, invited participants may have not been engaged in any architecture evaluation during the period in which we conducted the survey. As a result, only a total of six participants performed a real-world evaluation, used our proposed guide during their activities, and provided us with their feedback. We are aware that such small number of participants poses a threat to validity to the statistical analysis used as evidence that the guide addresses the evaluators' concerns. However, the participants are heterogeneous in terms of having different experiences in architecture evaluation, use of evaluation guides, and the experience of conducting evaluation in a REST-based architecture. Such heterogeneity contributes to understand the viewpoints of a broader range of AEs.

[1] A *cheat sheet* is concise set of notes used for quick reference. It is commonly used in many activities of software architecture.

Fig. 6. First page of the Web tool.

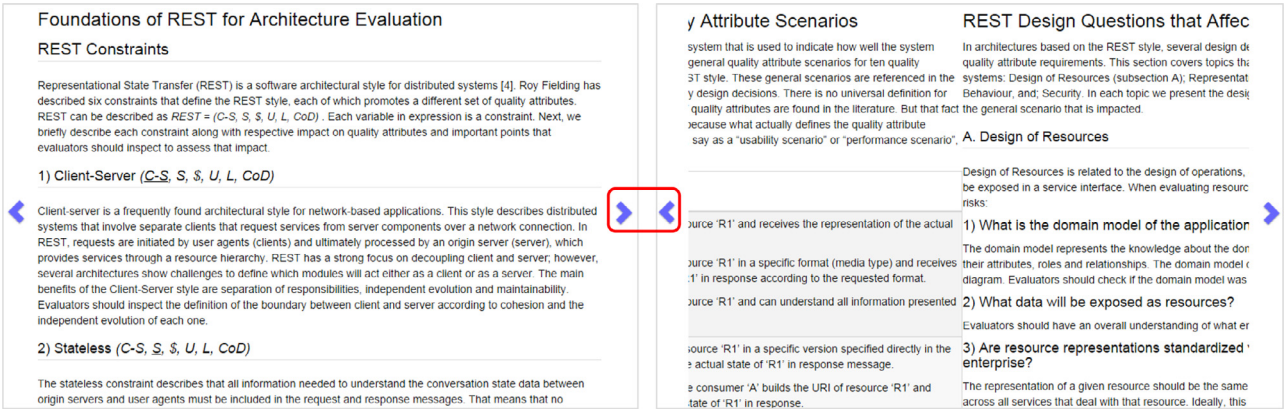


Fig. 7. Transitions between pages.

9.1.4. External validity

External validity determines the generalization degree of the findings of the study (Creswell, 2003). In our study, the external validity threat is the bias of the small number of participants of the survey in the results. Therefore, it is necessary to perform experiment replication with larger number of participants to confirm or contradict the results. However, the process and materials presented in this paper are enough to replicate the experiment.

10. Web tool

In order to meet the aforementioned criticisms and suggestions made by the evaluators in the survey, we developed a Web tool to facilitate the use of the guide. The goal of the tool was to present the entire guide proposed in this paper in a way that is easy to navigate. The tool used a responsive Web design, so it can be accessed by a variety of devices, such as computers, tablets, and smartphones. Furthermore, after the first access, the tool is cached on the client platform, so that it can be used off-line. The Web tool can be found at http://ubicomp.nce.ufrj.br/restguidance.

In the first page (Fig. 6), by clicking on the button "Guide", the evaluator is redirected to the guide. The guide presents three different pages: (i) Foundations of REST for architecture evaluation (Section 5), (ii) Examples of REST general quality attribute scenarios (Section 6), and; (iii) REST design questions that affect quality attributes (Section 7). By using the functionality *Carrousel Slide*, the user can easily navigate between the sections, as showed in Fig. 7.

One of the main functionalities presented by the tool can be found in the design questions page. As suggested by the evaluators, for each generic scenario referenced in the description of the design questions, we created a dynamic popover window that presents its description. For example, the text corresponding to question 5 of item A (design of services) describes that the general scenario S1 is impacted. Therefore, the Web tool prevents that the evaluator has to stop reading and go back to the description of the aforementioned generic scenario. With the Web tool, the user only needs to click (if the device in use is a computer) or touch (if it is a tablet or smartphone) and the tool shows the text of the respective scenario. By avoiding the disruption of having to go back and forth between pages, we believe that the tool can increase the ease of use of the guide. Fig. 8 shows this functionality.

The Web tool presents the cheat sheet suggested by evaluators. In the first page, by clicking on the button "Cheat Sheet", all design questions and their impacts on quality attributes are presented in a short version. With this functionality, evaluators can access a quick reference, for example, in order to conduct a final check list. Fig. 9 shows the cheat sheet page.

Another suggestion from AEs was to provide the example introduced in Section 8 in such a way that it was not necessary to stop reading and find the reference each time that some general scenario or design question appear in the text. The Web tool presents the proof-of-concept evaluation (Section 8) in such a way that the referenced design questions are dynamically shown. As depicted in Fig. 10, the user only clicks on the referenced question (in the figure,

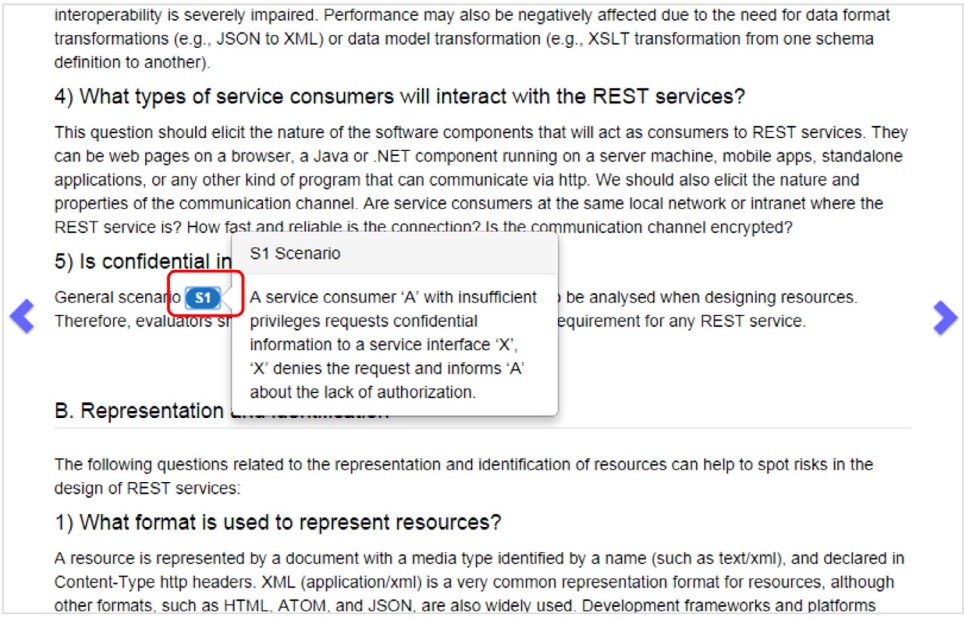


Fig. 8. Scenario description in design questions.

REST Architecture Evaluation Guidance Cheat Sheet

Topic	Design Question	Quality Attributes Directly Affected
Design of Resources	What is the domain model of the application?	Interoperability
	What data will be exposed as resources?	Interoperability
	Are resource representations standardized within the entire application, department, or enterprise?	Interoperability and Performance
	What types of service consumers will interact with the REST services?	Interoperability and Security
	Is confidential information exposed as a resource?	Security
Representation and identification	What format is used to represent resources?	Performance and Interoperability
	Is a standardized vocabulary defined for the resource?	Interoperability
	How do you design resources URIs?	Reliability, Modifiability, and Functionality
	What is the approach for resource versioning?	Interoperability, Modifiability, and Reliability
	How do you map operations on resources to http verbs?	Functionality and Safety

Fig. 9. REST guide cheat sheet.

"subsection B, question 1") and the description of the question appears. Without this functionality, AEs would have to stop reading and go back to read the question description.

The URL of the Web tool was sent to the AEs that are already using it in their architecture evaluations. Also, as informed by one of the evaluators, the Web tool has been used as an educational tool for less experienced team members.

11. Related work

To place our work in the landscape of related checklists and guidelines that allow the architect to assess compliance or that guide the architect when making design decisions, we surveyed prior work in the area of software quality and architecture evaluation. The solution presented in this paper brings about guidelines to assist AEs to mitigate risks and balance tradeoffs related to quality attribute requirements when evaluating REST-based architectures. Although there are some attempts providing useful information for architecture evaluation in areas such as legacy systems integration (Makki, 2006; Agirre et al., 2014), metadata representation (Oliveira et al., 2007; Ma et al., 2011), and security tradeoffs (Lai; et al., 2010; Musgrove et al., 2014), they are set on a track significantly different from our approach. However, regarding SOAs, there are similar approaches to assist architecture evaluation proposed in the last years (Bianco et al., 2007; Wang et al., 2011; Xiao-Jun, 2009; Abelein et al., 2009).

Regarding systems integration, Makki proposes an in-depth analysis of legacy systems integration with distributed newest systems (Makki, 2006). Makki's analysis provides a knowledge base that can be used in the architecture specification and evaluation of a distributed application that needs to be integrated with legacy systems. Agirre and associates introduce an architecture and implementation for a component-based platform that provides a basic resource management for distributed applications (Agirre et al., 2014). The architecture specification provides useful information for architecture evaluation activities, such as the analysis of resource management mechanisms that a component-based platform should implement in

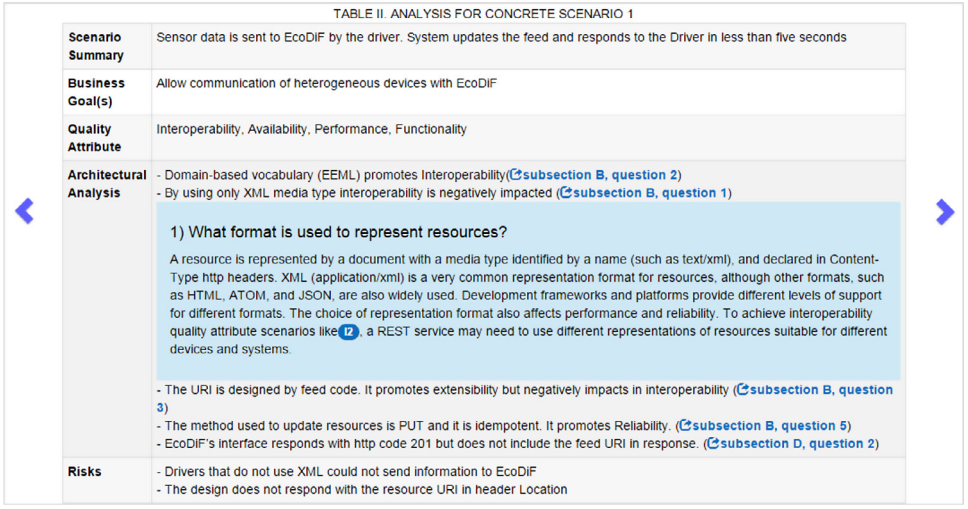


TABLE II. ANALYSIS FOR CONCRETE SCENARIO 1

Scenario Summary	Sensor data is sent to EcoDiF by the driver. System updates the feed and responds to the Driver in less than five seconds
Business Goal(s)	Allow communication of heterogeneous devices with EcoDiF
Quality Attribute	Interoperability, Availability, Performance, Functionality
Architectural Analysis	- Domain-based vocabulary (EEML) promotes Interoperability(subsection B, question 2) - By using only XML media type interoperability is negatively impacted (subsection B, question 1) **1) What format is used to represent resources?** A resource is represented by a document with a media type identified by a name (such as text/xml), and declared in Content-Type http headers. XML (application/xml) is a very common representation format for resources, although other formats, such as HTML, ATOM, and JSON, are also widely used. Development frameworks and platforms provide different levels of support for different formats. The choice of representation format also affects performance and reliability. To achieve interoperability quality attribute scenarios like 12, a REST service may need to use different representations of resources suitable for different devices and systems. - The URI is designed by feed code. It promotes extensibility but negatively impacts in interoperability (subsection B, question 3) - The method used to update resources is PUT and it is idempotent. It promotes Reliability. (subsection B, question 5) - EcoDiF's interface responds with http code 201 but does not include the feed URI in response. (subsection D, question 2)
Risks	- Drivers that do not use XML could not send information to EcoDiF - The design does not respond with the resource URI in header Location

Fig. 10. Question 1 referenced in the architecture analysis dynamically shown.

order to support higher-level services such as admission control, fault tolerance, and load balance.

Looking at metadata representation, Oliveira and associates present a methodology and tool to assist architects to identify artefacts that can be considered as reusable assets (Oliveira et al., 2007). Their methodology is based on a metadata representation model, implemented in the tool, providing an interactive graphic visualization of artefacts that are candidates for reuse. In architecture evaluation activities, the methodology provides a knowledge base that can assist evaluators when analysing software reuse requirements. Ma and associates propose a three-layer metadata model, including data source metadata, business metadata, and topic metadata to assist the realization of heterogeneous data integration (Ma et al., 2011). The architecture of their three-layer approach is described by using the HTTP protocol and Web services that are based on the REST style. Regarding heterogeneous data integration requirements, their paper provides a metadata model that helps evaluators to mitigate risks related to interoperability requirements.

Security tradeoffs in reusable biometric systems are analysed by Lai and colleagues (2010). They propose a deep analysis, in which the same biometric information is reused for several systems. The paper also presents a discussion about conditions under which the architecture promotes performance, considering situations where there are legacy systems required to be kept intact. Similarly, Musgrove and associates introduce an approach for modelling and analysis of performance and security tradeoffs (Musgrove et al., 2014). Such approach is evaluated through the analysis of performance sensitivity of a kind of system called Border Inspection Management System (BIMS). Examples of BIMS are biometric systems for passenger identification.

In all of these approaches, the focus is on providing an architectural solution that can be used as a knowledge base for assisting the evaluation and specification of integrated systems. Our proposal, in contrast, focuses on providing a well-structured guide specifically for scenario-based architecture evaluation activities. In the aforementioned studies, AEs must extract the general scenarios and design questions based on the information available in the study, whereas in our work we provide them explicitly in the guide.

Regarding the state of the art in quality attribute analysis on SOAs, an important publication is the report titled "Evaluating a Service-Oriented Architecture" that was mentioned in the Introduction (Bianco et al., 2007). Other evaluation guides for SOA systems have also been proposed. For example, Wang proposes a methodology to evaluate SOAs (Wang et al., 2011) that not only integrates existing and legacy systems, but also analyses the capacity of SOA through SWOT analysis—a strategic planning methodology that is used to evaluate the strengths, weaknesses, opportunities and threats of a project or a business venture. Xiao-Jun has an approach for defining metrics centred on design concerns of loosely coupled services and granularity (the optimal scope of business functionality in a service operation) (Xiao-Jun, 2009). The metrics are based on information-theoretic principles and are used to predict the quality of the final software product. Abelein and associates introduce an overview of a framework for describing and evaluating the business benefits of SOA (Abelein et al., 2009). Based on the identified gaps of existing work, their proposed model was designed to match some quality attribute criteria.

Although the aforementioned publications provide design considerations that assist architects and evaluators in their activities, none of them focuses on the REST style. These publications deal with SOA services that are primarily based on SOAP. To the best of our knowledge, there is no published guide to help AEs to assess REST-based systems.

12. Conclusions and future work

The main contribution of the paper is the collection of general quality attribute scenarios and design questions that can assist evaluators in REST-based architecture evaluations. As discussed in Section 3, the proposed guidelines can be used in scenario-based evaluation methods such as ATAM to help evaluators to probe architectural approaches in order to identify tradeoffs and risks to addressing quality attribute requirements. In addition, the results of the evaluation of the guide itself (Section 9) make us believe that the guide provides real value for software architecture evaluations performed on REST-based systems.

The developed Web tool presented in Section 10 can be a valuable resource for AEs in their activities. As presented in Section 10, the entire list of general quality attribute scenarios and design questions is available on the Internet. Evaluators can also find a summarized list of questions and the related impact on quality attributes.

The research methodology applied in this paper has a limitation related to the validation of design questions: it does not define an activity to evaluate the design questions that were elicited in interviews and literature review. Therefore, future work should include validation activities in the protocol. It is also important to investigate other elements that influence design questions and quality attribute requirements for REST systems, such as: (i) integration with legacy systems; (ii) metadata representation; and (iii)

security tradeoffs with SSL. We also intend to investigate how Cloud Computing features affect our set of REST design questions and general quality attribute scenarios. Based on design questions and scenarios presented in this paper, we plan to create a framework to appraise REST APIs (something similar to a REST-based API maturity model). The objective is to build a reference model to evaluate the quality of such APIs, to help service consumers and service providers to choose and/or analyse REST-based services. Another future project is to include in the guide a systematic approach to extract concrete scenarios from general ones (research in this direction is currently been carried out by the authors and preliminary results were already published; Costa et al., 2015). Finally, we intend to perform new experiments in order to evaluate the Web tool presented in this paper. In particular, it is our goal to invite the AEs that suggested the development of the Web tool to participate in such experiment.

Acknowledgments

This work was partially supported by Brazilian Funding Agencies FAPERJ, CNPq, COPPETEC/UFRJ, and UFRJ/CENPES. Flávia C. Delicato and Paulo F. Pires are CNPq Fellows (under grants 307378/2014-4 and E-26/110.468/2012 for Flavia C. Delicato; and under grants 457783/2014-1 and 310661/2012-9 for Paulo F. Pires). We thank all the specialists who contributed to our work in particular Matthias Naab (Fraunhofer IESE), Daniel Jacobson (Netflix), Casare Pautasso (Università della Svizzera Italiana), Erik Wilde (UC Berkeley School of Information), Felipe Oliveira and Alexandre Saudate (SOA Expert), Luiz Cipriani and Luiz Rocha (Abril Midia), Kleber Bacili (Sensedia), Wanderlei Souza (Apontador.com), engineers from Google Maps API, and architecture evaluators from the SEI.

References

Abelein, U., Habryn, F., Becker, A., 2009. Towards a holistic framework for describing and evaluating business benefits of a service oriented architecture. In: Proceedings of Enterprise Distributed Object Computing Conference Workshops, pp. 282–289. doi:10.1109/EDOCW.2009.5331981.

Abeysinghe, S., 2008. Restful PHP Web Services. Packt Publishing.

Adamusiak, T., Burdett, T., Kurbatova, N., van der Velde, K.J., Abeygunawardena, N., Antonakaki, D., et al., 2011. OntoCAT–simple ontology search and integration in Java, R and REST/JavaScript. BMC Bioinform. 12 (1), 218.

Agirre, A., Estevez, E., Marcos, M., 2014. Resource management support for SCA based distributed applications. In: Emerg. Technol. Factory Automat., pp. 1–4. doi:10.1109/ETFA.2014.7005312.

Atzori, L., Iera, A., Morabito, G., 2010. The Internet of Things: A survey. Comput. Netw. 54, 2787–2805.

Babar, M.A., Gorton, I., 2004. Comparison of scenario-based software architecture evaluation methods. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference, pp. 600–607. doi:10.1109/APSEC.2004.38.

Bass, L., Clements, P., Kazman, R., 2012. Software Architecture in Practice, 3rd. ed. Addison-Wesley, Massachusetts.

Bass, L., M.H., Klein, G., Moreno, 2001. Applicability of General Scenarios to the Architecture Tradeoff Analysis Method. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania Tech. Rep. CMU/SEI-2001-TR-014. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=5637 .

Bengtsson, P.O., Bosch, J., 1998. Scenario based software architecture reengineering. In: Proceedings of the International Conference of Software Reuse, pp. 308–317. doi:10.1109/ICSR.1998.685756.

Berners-Lee, T., 1996. Universal Resource Identifiers—Axioms of Web Architecture [Online]. Available from: http://www.w3.org/DesignIssues/Axioms.html.

Bertolino, A., Inverardi, P., Muccini, H., 2013. Software architecture-based analysis and testing: a look into achievements and future challenges. Comput. J. 95 (8), 633–648.

Bianco, P., Kotermanski, R., Merson, P., 2007. Evaluating a Service-Oriented Architecture. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania Tech. Rep. CMU/SEI-2007-TR-015, http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8443.

Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., McLeod, G., Merritt, M., 1978. Characteristics of Software Quality. North Holland.

Burke, B., 2009. RESTful Java with Jax-RS. O'Reilly Media.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20 (6) 476–493.

Chung, L., Nixon, B., Yu, E., Mylopoulos, J., 2001. Non-functional requirements in software engineering. Kluwer Academic Publishers, Boston.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., et al., 2010. Documenting Software Architectures: Views and Beyond, 2nd. ed. Addison-Wesley.

Clements, P., Kazman, R., Klein, M., 2001. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley Professional.

Costa, B., Pires, P.F., Delicato, F.C., Merson, P., 2014. Evaluating a representational state transfer (REST) architecture—What is the impact of REST in my architecture? In: Proceedings of the Eleventh Working IEEE/IFIP Conference on Software Architecture, pp. 105–114. doi:10.1109/WICSA.2014.29.

Costa, B., Pires, P.F., Delicato, F.C., Oquendo, F., 2015. Towards a view-based process for designing and documenting RESTful service architectures. In: Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW '15). ACM, New York, NY, USA doi:10.1145/2797433.2797485, Article 50, 7 pages.

Creswell, J.W., 2003. Research Design: Qualitative, Quantitative, and Mixed Method approaches. Sage Publications, CA.

Daigneau, R., 2011. Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services. Addison-Wesley Professional.

de Cluni, G.T., Serrao, T., Monteiro Braz, J.R., Serrão, T., Rangel, N., Castillo, A., et al., 2012. Developing an android based learning application for mobile devices. In: Proceedings of the Euro American Conference on Telematics and Information Systems (EATIS), pp. 1–7.

Delicato, F.C., Pires, P.F., Batista, T., Cavalcante, E., Costa, B., Barros, T., 2013. Towards an IoT ecosystem. In: Cuesta, Carlos E., Maldonado, José Carlos, Nakagawa, Elisa Y., Drira, Khalil, Zisman, Andrea (Eds.), Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems (SESoS '13). ACM, New York, NY, USA, pp. 25–28. doi:10.1145/2489850.2489855.

DeVellis, R.F., 2003. Scale Development. Theory and Applications. Applied Social Research Methods Series, 26. Sage Publications, California.

Doboš, J., Sons, K., Rubinstein, D., Slusallek, P., Steed, A, 2013. XML3DRepo. In: Proceedings of 18th International Conference on 3D Web Technology—Web3D '13. ACM Press, New York, p. 47.

Dodero, J.M., Ghiglione, E., 2008. REST-based web access to learning design services. IEEE Trans. Learn. Technol. 1 (3), 190–195.

Dukes, K.A., 2014. Likert Scale. Wiley StatsRef: Statistics Reference Online. 2014.

Dyba, T., Kitchenham, B.A., Jorgensen, M., 2005. Evidence-based software engineering for practitioners. IEEE Softw. 22, 58–65.

Easterbrook, S., Singer, J., Storey, M.-A., Damian, D., 2008. Selecting empirical methods for software engineering research. Guide to Advanced Empirical Software Engineering. Springer, pp. 285–311.

Erl, T., 2008. SOA Principles of Service Design. Pearson Education, Boston.

Erl, T., Carlyle, B., Pautasso, C., Balasubramanian, R., Wilhelmsen, H., Booth, D., 2012. SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST. Prentice Hall.

Extended Environments Markup Language [Online] (2008). Available: http://www.eeml.org/.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., et al., 1999. Network Working Group Request for Comments: 2616. Tech. Rep. Internet Engineering Task Force. The Internet Society [Online]. Available from: http://www.ietf.org/rfc/rfc2616.txt.

Fielding T.F., "Architectural Styles and the Design of Network-based Software Architectures" (Ph.D. dissertation). University of California, Irvine, 2000.

Franco, J.M., Barbosa, R., Zenha-Rela, M., 2012. Automated reliability prediction from formal architectural descriptions. In: Proceedings of Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture, pp. 302–309. doi:10.1109/WICSA-ECSA.212.50.

Gambi, A., Pautasso, C., 2013. RESTful business process management in the cloud. In: Proceedings of the International Workshop on Principles of Engineering Service-Oriented Systems (PESOS), pp. 1–10. doi:10.1109/PESOS.2013.6635971.

Guinard, D., Trifa, V., Pham, T., Liechti, O., 2009. Towards physical mashups in the Web of Things. In: Proceedings of the International Conference on Networked Sensing Systems (INSS), pp. 1–4. doi:10.1109/INSS.2009.5409925.

Gulden, M., Kugele, S., 2013. A concept for generating simplified RESTful interfaces. In: Proceedings of the 22nd International Conference on World Wide Web companion (WWW '13 Companion). Switzerland, pp. 1391–1398.

Haupt, F., Karastoyanova, D., Leymann, F., Schroth, B., 2014. A model-driven approach for REST compliant services. In: Proceedings of IEEE International Conference on Web Services, pp. 129–136. doi:10.1109/ICWS.2014.30.

Heiberger, R.M., Robbins, N.B., 2014. Design of diverging stacked bar charts for Likert scales and other applications. J. Stat. Softw. 57 (5), 1–32.

Immonen, A., Pakkala, D., 2014. A survey of methods and approaches for reliable dynamic service compositions. Serv. Orient. Comput. Appl. 8 (2), 129–158.

ISO 25010:2011. Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models. ISO/IEC, 2011.

ISO/IEC/IEEE 24765:2010—Systems and software engineering—Vocabulary. ISO/IEC/IEEE, 2010.

ISO/IEC/IEEE 42010:2011—Systems and software engineering—Architecture description. ISO/IEC/IEEE, 2011.

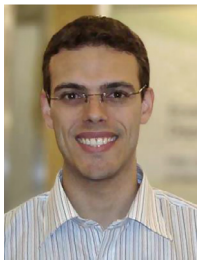
Jamal, S., Deters, R., 2011. Using a cloud-hosted proxy to support mobile consumers of RESTful services. Proc. Comput. Sci. 5, 625–632.

Kasunic, M., 2005. Designing an Effective Survey. Software Engineering Insitute. Tech. Rep. CMU/SEI-2005-HB-004, September.

Kazman, R., Abowd, G., Bass, L., Clements, P., 1996. Scenario-based analysis of software architecture. IEEE Softw. 13 (6), 47–55.

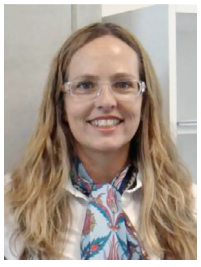
Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., Carriere, J., 1998. The architecture tradeoff analysis method. In: Proceedings of the International Conference on Eco-friendly Computing and Communication Systems (ICECCS), pp. 68–78. doi:10.1109/ICECCS.1998.706657.

Kitchenham, B.A., Pfleeger, S.L., 2001. Principles of survey research. Part 2: Desinging a survey. ACM SIGSOFT Softw. Eng. Notes 26 (6), 16–18.
Kitchenham, B.A., Pfleeger, S.L., 2002a. Principles of survey research. Part 5: Populations and samples. ACM SIGSOFT Softw. Eng. Notes 27 (5), 17–20.
Kitchenham, B.A., Pfleeger, S.L., 2002b. Principles of survey research Part 4: Questionnaire evaluation. ACM SIGSOFT Softw. Eng. Notes 27 (3), 20–23.
Kitchenham, B.A., Pfleeger, S.L., 2002c. Principles of survey research Part 3: Constructing a survey instrument. ACM SIGSOFT Softw. Eng. Notes 27 (2), 20–24.
Kitchenham, B., Charters, S., 2007. Guidelines for performing systematic literature reviews in Software Engineering. Tech. Rep., Keele University/University of Durham.
Kitchenham, B.A., 2004. Procedures for performing systematic reviews. Tech. Rep., Keele University/National ICT Australia Ltd.
Koziolek, H., 2010. Performance evaluation of component-based software systems: A survey. Perform. Eval. 67 (8), 634–658.
Kruchten, P.B., 1995. The 4+1 view model of architecture. IEEE Softw. 12 (6), 42–50.
Lai, L., Ho, S.W., Poor, H.V., 2010. Privacy-security tradeoffs in reusable biometric security systems. In: Proceedings of the IEEE International Conference on Acoustics Speech and Signal Processing, pp. 1722–1725. doi:10.1109/ICASSP.2010.5495470.
Lassing, N., Rijsenbrij, D., Vliet, H.v., 1999. On Software architecture analysis of flexibility, complexity of changes: Size isn't everything. In: Proceedings of the 2nd Nordic Software Architecture Workshop (NOSA '99).
Lewis, J., Fowler, M., (2014). Microservices [Online]. Available: http://martinfowler.com/articles/microservices.html.
Li, L., Chou, W., 2009. Infoset for service abstraction and lightweight message processing. In: Proceedings of the IEEE International Conference on Web Services, pp. 703–710. doi:10.1109/ICWS.2009.120.
Lu, Q., Xu, X., Tosic, V., Keung, J.W., Zhu, L., 2010. Integration of RESTfulBP with BDIM decision making. In: Middleware '10 Posters and Demos Track (Middleware Posters '10). ACM, New York, NY, USA Article 2, 4 pages. http://dx.doi.org/10.1145/1930028.1930030.
Lu, Q., Xu, X., Tosic, V., Zhu, L., 2012. Service-Oriented Computing, 7636. Springer Berlin HeidelbergHeidelberg, Berlin, pp. 404–419.
Ma, Z., Wang, C., Wang, Z., 2011. Research on three-layered metadata model for oil-gas data integration. In: Proceedings of IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS), pp. 500–504. doi:10.1109/CCIS.2011.6045118.
Makki, S.K., 2006. The integration and interoperability issues of legacy and distributed systems. In: Proceedings of Seventh International Conference on Web-Age Information Management, p. 21. doi:10.1109/WAIMW.2006.30.
Maleshkova, M., Pedrinaci, C., Domingue, J., 2010. Investigating Web APIs on the World Wide Web. In: Proceedings of Eighth IEEE European Conference on Web Services, pp. 107–114. doi:10.1109/ECOWS.2010.9.
Martin, R.C., 2009. Clean Code: A Handbook of Agile Software Craftsmanship. Pearson Education, Boston.
McCall, J., Richards, P., Walters, G., 1997. Factors in Software Quality, vol. 1–3, Italy.
Musgrove, J., Cukic, B., Cortellessa, V., 2014. Proactive model-based performance analysis and security tradeoffs in a complex system. In: Proceedings of the IEEE 15th International Symposium on High-Assurance Systems Engineering (HASE) 211–215.
Musser, J., 2008. Open APIs: State of the Market. [Online]. Available from: http://www.infoq.com/presentations/Open-API-John-Musser.
M., Naab, 2013. All architecture evaluation is not the same—Lessons learned from more than 50 architecture evaluations in industry. In: Architecture Technology User Network (SATURN). Software Engineering Institute.
Newman, S., 2015. Building Microservices: Designing Fine-Grained Systems. O'Reilly.
Nunnaly, J., 1978. Psychometric Theory. McGraw-Hill, New York.
OAuth Community Site (2015). [Online]. Available from: http://oauth.net/.
Oishi, S.M., 2002. How to Conduct In-Person Interviews for Surveys, 2nd ed. Sage Publications, Thousand Oaks, CA.
Oliveira, M., Goncalves, E.M., Bacili, K.R., 2007. Automatic identification of reusable software development assets: Methodology and tool. In: Proceedings of the IEEE International Conference on Information Reuse and Integration, pp. 461–466. doi:10.1109/IRI.2007.4296663.
OMA EMML (2013). Documentation [Online]. Available from: http://www.openmashup.org/omadocs/v1.0/index.http.
OpenID Foundation (2015). Website [Online]. Available from: http://openid.net/.
Pautasso, C., Wilde, E., Alarcon, R., 2014. REST: Advanced Research Topics and Practical Applications. Springer Publishing Company.
Petersen, K., Feldt, R., Mujtaba, S., Mattson, M., 2008. Systematic mapping studies in software engineering. In: Visaggio, Giuseppe, Baldassarre, Maria Teresa, Linkman, Steve, Turner, Mark (Eds.), Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE'08). British Computer Society, Swinton, UK, UK, pp. 68–77.
Pfleeger, S.L., Kitchenham, B.A., 2001. Principles of survey research Part 1: Turning lemons into lemonade. ACM SIGSOFT Softw. Eng. Notes 26 (6), 16–18.
Renzel, D., Schlebusch, P., Klamma, R., 2012. Today's top 'RESTful' services and why they are not restful. In: Proceedings of the 13th International Conference on Web Information Systems Engineering 7651. Springer, Berlin Heidelberg, pp. 354–367.
Richardson, C., (2014). Microservices: Decomposing Applications for Deployability and Scalability [Online]. Available from: http://www.infoq.com/articles/microservices-intro.
Richardson, L., Ruby, S., 2007. RESTful Web Services. O'Reilly Media, Sebastopol.
Roy, B., Graham, N., 2008. Methods for Evaluating Software Architecture: A Survey. Tech. Rep., Tech Republic.
Sletten, B., 2013. Resource-Oriented Architecture Patterns for Webs of Data. Morgan & Claypool Publishers.
Sundvall, E., Nyström, M., Karlsson, D., Eneling, M., Chen, R., Örman, H., 2013. Applying representational state transfer (REST) architecture to archetype-based electronic health record systems. BMC Med. Inform. Decis. Mak. 13 (1), 57.
Tang, A., van Vliet, H., 2009. Modeling constraints improves software architecture design reasoning. In: Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture (WICSA/ECSA'09), pp. 253–256. doi:10.1109/WICSA.2009.5290813.
Vinoski, S., 2007. REST eye for the SOA guy. IEEE Internet Comput. 11 (1), 82–84.
Wang, Y.H., Liao, J.C., Tsai, C.H., 2011. An objective concept for evaluating service oriented architecture. Proceedings of the Eighth International Conference on Fuzzy Systems and Knowledge Discovery vol. 4, 2349–2353. doi:10.1109/FSKD.2011.6019970.
Webber, J., Parastatidis, S., Robinson, I., 2010. REST in Practice: Hypermedia and Systems Architecture. O'Reilly Media, Sebastopol.
Wilde, E., Pautasso, C., 2011. REST: From Research to Practice. Springer.
Xiao-Jun, W., 2009. Metrics for evaluating coupling and service granularity in service oriented architecture. In: Proceedings of the International Conference on Information Engineering and Computer Science, pp. 1–4. doi:10.1109/ICIECS.2009.5362767.
Xu, X., Zhu, L., Kannengiesser, U., Liu, Y., 2010. An architectural style for process-intensive web information systems. Lect. Notes Comput. Sci. 6488, 534–547.
Xu, X., Zhu, L., Liu, Y., Staples, M., 2008. Resource-oriented architecture for business processes. In: Proceedings of the 15th Asia-Pacific Software Engineering Conference, pp. 395–402. doi:10.1109/APSEC.2008.52.



Bruno Costa is a doctoral student in the Department of Computer Science, Federal University of Rio de Janeiro, Brazil. He has experience on software development and architecture specification with focus on information systems. In the last years, he has been participating in cooperative industry—university software projects in the areas of Internet of Things, Big Data, and Oil and Gas. His research focuses on software architecture, self-adaptive systems, architecture-centric model driven engineering, and the application of Software Engineering techniques in the development of software systems for Internet of Things (IoT).

Paulo F. Pires is an associate professor in the Department of Computer Science, Federal University of Rio de Janeiro, Brazil, and member of the Centre for Distributed and High Performance Computing, University of Sydney, Australia. His main research interests include model driven development, software product lines, infrastructures for Web service composition, and application of Software Engineering techniques in the development of software systems for emerging domains, such as embedded, ubiquitous and pervasive systems. He holds a technological innovation productivity scholarship from the Brazilian Research Council (CNPq) since 2010 and is a member of the Brazilian Computer Society (SBC).



Flávia C. Delicato is an associate professor in the Department of Computer Science, Federal University of Rio de Janeiro, Brazil, and member of the Centre for Distributed and High Performance Computing, University of Sydney, Australia. She is a researcher fellow of the National Council for Scientific and Technological Development (CNPq) and a young researcher fellow from FAPERJ Brazilian Funding Agency. She has been participating in several research projects with funding from International and Brazilian government agencies. Her main research interests are: wireless sensor networks and ubiquitous computing; adaptive middleware; model-driven development; Web service technologies, and internet of things.

Paulo Merson received the B.Sc. degree in computer science from University of Brasilia, and a master's degree in software engineering from Carnegie Mellon University. He has 25 years of experience programming in the large and in the small. He is a software architect working with distributed systems at the Brazilian Federal Court of Accounts. He is a faculty member of the professional master program at University of Brasilia, as well as SOA trainer and consultant. He has also worked for the Software Engineering Institute as a visiting scientist where he wrote several reports on SOA.