Research Article

Modeling and simulation framework for quality estimation of web applications through architecture evaluation

María Julia Blas[1] · Horacio Leone[1] · Silvio Gonnet[1]

Received: 3 December 2019 / Accepted: 3 February 2020

Abstract
The explosive growth of the cloud computing industry in recent years has paying attention to problems related to software services quality. Given that quality models serve as frameworks for quality evaluation, this paper proposes a modeling and simulation framework that measures properties derived from ISO/IEC 25010 quality model as main quality concerns of cloud computing applications. The simulation models are obtained by translating the architectural design to an equivalent functional description that, with aims to obtain the quality evaluation, explores all possible component states. Moreover, the framework automatically builds the simulation models using a set of predefined behaviors as components descriptors. Such models are combined with an experimental frame in a simulation scenario that helps to estimate quality employing the performance of the architectural design. Therefore the simulation process is hidden to software architects, providing an evaluation process able to be executed by any developer without knowledge of discrete-event simulation. Two general architectures are used as case study in order to show how works the modeling and simulation framework.

Keywords Cloud computing applications · Discrete-event simulation · Software quality evaluation · Routed DEVS formalism

1 Introduction

Cloud computing is a model for enabling convenient, on demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction [1]. Then, cloud computing applications (CCA) can be studied as software services.

Cloud-providers are able to seamlessly update their cloud applications without requiring users to install any sort of upgrade or patch since all cloud software resides on servers located in the provider's data centers [2]. Thus, consumers no longer need to invest in infrastructure. Using the cloud infrastructure, a customer can get access to their applications anytime, and from anywhere, through a connected device to the network [3]. However, due to the increasing demand for cloud services and many consumers deploying and hosting Information Technology services to cloud computing it is difficult for consumers to select the services appropriate to their needs [4]. Unlike many software vendors, if a new company puts up a competitive service of higher quality, customers will almost immediately shift their application to the new service once they discover it [5]. Then, customers expect a certain level of maturity in software products and services, a level that turns software quality into a competitive advantage [6]. To remain competitive, software services must deliver high quality applications. Hence, developers must ensure the

✉ María Julia Blas, mariajuliablas@santafe-conicet.gov.ar; Horacio Leone, hleone@santafe-conicet.gov.ar; Silvio Gonnet, sgonnet@santafe-conicet.gov.ar | [1]Instituto de Diseño y Desarrollo INGAR, Consejo Nacional de Investigaciones Científicas y Técnicas - Universidad Tecnológica Nacional, Avellaneda 3657, 3000 Santa Fe, Argentina.

Published online: 10 February 2020

software quality using conventional quality features prior deploying the CCA in the environment.

Software quality evaluation is a complex activity. According to IEEE [7], the software product quality refers to the degree to which a system, component, or process meets specified requirements. As software size and complexity grows, the quality evaluation becomes a problem. The intrinsic features of CCA require more rigorous quality measurements than traditional software products [8].

In this context, it is important to understand that an early software quality evaluation has more value than a later evaluation of source code. In early stages of software development, the challenge is achieve an adequate quality level to guarantee the application acceptance. However, there is little guidance available to help quality engineers to understand their applicability in the cloud service quality delivery chain [9]. Conventional methodologies do not effectively support the dynamic required in the quality properties evaluation of the cloud computing environments based on specific features such as scalability. This lack of guidance challenges software and quality engineers to develop new solutions for quality measurement. These solutions should be seen as complementary evaluation techniques of the traditional ones. This paper proposes a new simulation framework for estimating the CCA quality using the architectural description for structuring the simulation model. The components used for building the architecture are obtained from cloud computing patterns. All simulation models are specified using DEVS [10] and RDEVS [11]. The main objective is to provide a Modeling and Simulation (M&S) framework that allows studying a restricted set of quality properties over the CCA design prior its implementation.

This paper is organized as follows: Sect. 2 resumes the evaluation approaches commonly used for studying software products quality and distinguishes them from the one presented in this paper; Sect. 3 summarizes of CCA structure including the quality properties and architectural patterns used as basis of the framework; Sect. 4 presents the design and implementation of the M&S framework; Sect. 5 details two case study based on comprehensive CCA architecture patterns; finally, Sect. 5 is devoted to the conclusions, final remarks and future research.

2 Quality evaluation in cloud computing

One of the main concerns of software engineering is the production of high quality software systems and thus software quality evaluation has always been a critical task for software professionals [12]. Hence, quality is important and it can be improved (no matter if the software product under development is a standalone program or web-based application).

In this context, software quality models are a well-accepted means to support the quality management of software systems [13]. According to Roy et al. [9], software quality engineering requires the use of a quality model with the capacity to support both definitions of quality requirements and their evaluation. Then, quality models are the cornerstone of a product quality evaluation system [7]. Ideally, quality evaluations should be guided by quality models. A quality model is a standard taxonomy of quality attributes. Key features of CCA distinguish their quality attributes from conventional software quality properties. However, in both cases there are static and dynamic properties. Consequently, quality evaluation methods estimate distinct types of quality properties. Moreover, quality attributes are not mutually exclusive; rather, they tend to interact with each other. The value of one quality attribute may depend on the value of another quality attribute. Then, an important part of any quality evaluation method is the identification of the suitable subset of quality attributes. Quality evaluation methods do not use the same subset of quality attributes.

Actually several authors have proposed quality models for cloud computing environments. In [14] authors claim that "existing conventional quality models are not competent enough for providing all software service specific features". Therefore, they present a quality model to assess the quality of software as a service on cloud computing environments around quality properties. However, they do not state how to use the quality model for quality evaluation. From this point of view, Bardsiri and Hashemi [15] provide a list of quality of service metrics that helps to study cloud services. A similar approach is developed in [8] were authors derive quality attributes from key features of cloud computing services and propose using the result of the measurement process as quality indicator. However, such evaluation requires a runnable version of the application (given that dynamic quality properties cannot be measured until the application is effectively executed). Then, the quality evaluation is performed after the software implementation.

In [16] the authors discuss a web quality evaluation method to assess web sites and applications. The paper is focused in fulfil quality requirements in web development projects and evaluate requirements in operational phases. The main concern of the authors is discovering absent features or poorly implemented requirements. Again, the quality evaluation is performed in late phases of the development process. Moreover, Lew et al. [17] propose extending the ISO 25010 standard to incorporate new characteristics and concepts in a flexible modeling framework. The final framework contributes towards a flexible, integrated

approach to evaluate web applications. The operability and learnability of a real web application are evaluated using the framework proposed by the authors.

In these researches, waiting until late in the development process may reduce the impact of the effectiveness and efficacy of any software quality improvement actions and increase their cost [18]. However, models available in later phases are expected to be more accurate than those available in earlier phases. Then, both models are needed. In fact, the study performed in [19] concludes that tools, metrics and evaluation research are needed to provide useful and trust worthy cloud computing services that deliver appropriate quality of service. Even when late quality evaluation is better than nothing, early quality evaluation may improve the development process in several ways. For example, an early identification of the set of software modules that is likely to be faulty helps practitioners take timely actions to improve the quality of these modules and reduce development costs in the remainder of the development process [18]. The earlier quality requirements are considered, the less effort is needed later in the software lifecycle to ensure a sufficient software quality levels [20].

The notion of software architecture has emerged as the appropriate level to deal with software qualities because sets the boundaries for the quality of the resulting system. Bass et al. [21] define software architecture as "the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them". Hence, there is no such thing as an inherently good or bad architecture [21]. A good architecture is one that meets all its requirements. Then, architecture evaluation of software products plays a central role in quality evaluation because it leads to correct designs that meet the set of quality requirements. Considering that Dobrica and Niemela [22] suggest that there are three phases of interest to architecture evaluation (early, middle, and post deployment), most architecture evaluation methods are able to be used across the development process. The issue, however, is getting the information required to perform the evaluation.

There are two basic classes of evaluation techniques available at the architecture level: *questioning* and *measuring* [21]. *Questioning* techniques generate qualitative questions to be asked over architecture design. These techniques are applicable to any quality property and their selection is not based on quality models. Commonly, *questioning* includes scenarios, questionnaires, and checklists. On the other hand, *measuring* techniques suggest quantitative measurements performed over the architecture such as metrics, prototypes and experiences. These techniques are used to answer specific questions and, therefore, usually are addressed to specific software qualities. Then, *measuring* techniques are not as broadly applicable as *questioning* techniques [22].

Both types of techniques (*questioning* and *measuring*) can be seen as conventional methods for software architecture evaluation. For example, the Software Architecture Analysis Method [23] evaluates design quality attributes by ranking candidate architectures on how well they support representative task scenarios. However, ones the architecture is implemented no further evaluation process can be performed to ensure its alignment with the quality attributes. Another example is the evaluation proposed by Morasca and Lavazza [18], where authors estimate the faultiness of a software module at any point during development by using measures collected up to that time. They built faultiness models combining code measures and design measures in order to improve software quality by faultiness estimation models used at design time. In this case, the evaluation method can be used in several development phases but its only concern is the measurement of an exclusive attribute.

All the conventional evaluation methods are useful to evaluate CCA software architectures. However, the evolution of cloud computing environments requires more flexible quality evaluations methods that adjust to the expected dynamic of the applications [24, 25]. Evaluating the performance of cloud provisioning policies, application workload models, and resources performance models in a repeatable manner under varying system and user configurations and requirements is difficult to achieve [26]. Hence, new quality evaluation methods arise to be used as complement of the conventional ones.

For example, in [27] authors propose fuzzy logic control to evaluate the quality of cloud services. The results demonstrate that this approach performs an accurate evaluation of quality in service-oriented cloud computing. This type of evaluation is focused on cloud services as holistic perspective of CCA. A similar type of evaluation is given by Garg et al. [28] as a framework that measures the quality and prioritizes cloud services. Such framework helps to create healthy competition among cloud providers to satisfy their service level agreement by improving their quality.

A different solution is developed in [26] where the authors propose an extensible simulation toolkit that enables modeling and simulation of cloud computing systems and application provisioning environments. The toolkit supports both system and behavior modeling of cloud system components such as data centers, virtual machines and resource provisioning policies. In this sense, the simulation appears to be a feasible solution for quality estimation as an alternative to conventional approaches. However, the toolkit proposed by the authors is devoted

Table 1 Comparison of quality evaluation methods for software products

	Novelty		Fixed to a quality model	Number of properties evaluated by the method		Development phase where is applicable	
	Conventional method	New method (complementary)		One property	Multiple properties	Before implementation	After implementation
Jagli et al. [14]	X		X				
Lee et al. [8]	X		X		X		X
Morasca and Lavazza [18]	X			X		X	X
Olsina and Rossi [16]	X		X		X		X
Lew et al. [17]	X		X		X		X
Bardsiri and Hashemi [15]	X				X		X
Questioning techniques	X				X	X	
Measuring techniques	X			X		X (e.g. metrics)	X (e.g. prototypes and experiences)
Wang et al. [27]		X			X		X
Garg et al. [28]		X	X		X		X
Calheiros et al. [26]		X			X	X	X
M&S framework (this paper)		X	X		X	X	X

to physical components. Therefore, it is not useful for the modeling and simulation of software application components.

This paper proposes a simulation framework that allows measuring quality attributes of CCA using a discrete event approach. The simulation models are obtained by translating the composition designed at architectural level into a set of models (devoted to the behavior and structure of these components). Then, the developers do not need to build the simulation model. Moreover, both functional and non-functional characteristics must be taken into consideration in the development of a quality software system [29]. That is way most evaluation methods need an executable version of the software product prior to estimate quality attributes. The addition of functional implementation as part of the evaluation justifies that models available in later phases are more accurate than those available in earlier phases. The M&S framework developed allows replacing the software product implementation in early phases of development by the simulation of well-known architectural component functionalities. Therefore this solution provides an evaluation method that presumes functionality without using its implementation. Given that the architectural representation is relevant in quality prediction and effort estimation [22], the architectural components included in the framework are controlled by a metamodel that defines CCA architectural patterns. The quality properties selected for the evaluation process are constrained by the ISO/IEC 25010 quality model [7]. Then, the framework allows measuring the behavior of CCA at design phase with aims to estimate the final product quality. Moreover, it can be used in late phases of development to assess the impact of architectural changes over actual quality (not necessarily obtained by the framework).

Table 1 resumes main differences among the quality evaluation approaches presented in this section following the criteria: novelty, use of quality models, number of properties evaluated by the method, and development phase where is applicable.

3 Cloud computing applications (CCA)

The objectives of the cloud computing paradigm are to increase capacity and capabilities at runtime without investing in new infrastructure, licensing new software, and training new recruits [30]. To accomplish these objectives, the cloud computing model delivers information technologies as customizable services. The model provides a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services that are delivered on demand to external customers over the Internet [24].

Commonly, cloud computing architectures are designed as layered structures [2, 31–33]. Layered architectures are a well-known architectural pattern in which a

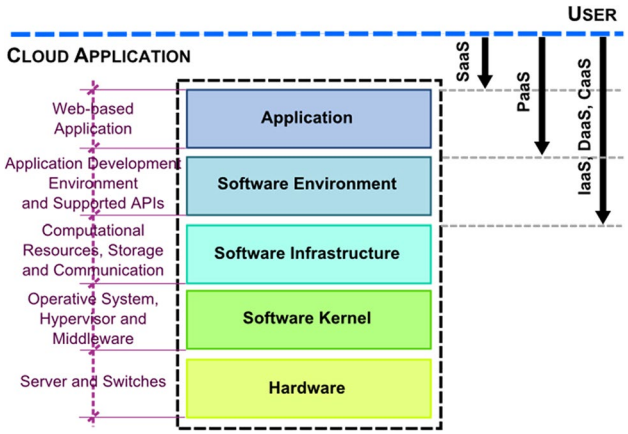


Fig. 1 Cloud computing layered architecture

layer is conceived as a coherent set of related functionality [21]. In a strictly layered structure, layer n may only use the services of layer n − 1. However, strictly layer structures are not often used in real architectures. Figure 1 depicts a generic representation of cloud computing environments as five-layer architecture [30, 32].

Each layer should be interpreted as follows:

- The *hardware layer* (layer 1) is responsible for managing the physical resources of the cloud, such as physical servers, routers, switches, power and cooling systems.
- The *software kernel layer* (layer 2) is responsible to allocate hardware resources (set at layer 1) to cloud users in an efficient, quick and smooth way.
- The *software infrastructure layer* (layer 3) creates a pool of storage and computing resources by partitioning the physical resources using virtualization technologies. This layer is commonly divided in three categories: *Infrastructure-as-a-Service (IaaS)* that refers to on-demand provisioning of computational resources; *Data-as-a-Service* (*DaaS*) that allows users to store an enormous amount of data on remote servers; and *Communication-as-a-Service* (*CaaS*) that provides secure, reliable, and fast communication services for users.
- The *software environment layer* (layer 4) provides an application development platform and a set of Application Programming Interfaces (APIs) to minimize the burden of deploying applications directly into virtual machine containers. Developers use these APIs as *Platform-as-a-Service* (*PaaS*) in order to program their applications without worry about hardware-software interactions.
- The *application layer* (layer 5) is the one that hosts the applications. Users can easily access such applications through a web client, even with limited processing and storage capabilities. This type of access is denoted as *Software-as-a-Service* (*SaaS*). The SaaS service provides the modeling of software deployment where users can run their applications without installing it in their computers [3].

Following the architectural structure described in Fig. 1, the CCAs are deployed at the top level. Then, their software architecture must be designed using services of layers 1–4. However, given that services from underlying layers are devoted to managing distinct aspects of the CCA execution, the set of elements used to design the architecture needs to be arranged considering the functional and non-functional requirements.

Non-functional requirements, as opposed to functional ones, do not express any functionality to be implemented [34]. Given that systems qualities are often expressed as non-functional requirements -also called quality attributes-, such requirements are recognized as very important factor to the success of software project [35–37]. Then, the ability to specify the quality properties required in a CCA is an important issue for consumers and service providers. Quality models are powerful tools that allow identifying the main quality properties to be evaluated in CCA. Moreover, the non-functional requirements are pervasive in descriptions of design patterns [38]. Software architects use design patterns to maintain balance between functional and non-functional requirements. Therefore, the M&S framework proposed in this paper uses architectural patterns to define the set of elements to be used in the CCA architectural description.

The following subsections describe the quality properties and design patterns of CCAs that were used as baselines for building the M&S framework.

3.1 Quality properties

Different from traditional applications, the CCA can leverage the automatic-scaling feature to achieve better performance, availability and lower operating cost [32]. Therefore, traditional quality models need to be modified to include quality properties of CCA.

In [39] authors use ontologies as model descriptor of quality schemes. A quality scheme is defined as "a set of triplets over a software product definition where each element is composed by a software attribute, the software metric that should be used to its measurement and the quality subcharacteristic that should be evaluated over it". The ISO/IEC 25010 quality model is used to bind the properties included in the scheme.

Quality schemes are generic documentation mechanisms designed to describe the quality properties required in a software product and, therefore, their structure can be used to support quality documentation. Given that CCA

quality evaluation implies the definition of quality properties, a CCA quality scheme can be defined to support the non-functional specification of this software product type. Using quality schemes as documentation strategy for CCA provides the quality agreement required for building the quality evaluation required as part of the M&S framework.

In recent years, several authors have studied the quality features of CCA [6, 8, 40–43]. Hence, in order to define a baseline for the CCA quality scheme, a set of twelve properties was selected: *reusability*, *availability*, *scalability*, *service customizability*, *functional feature commonality*, *non-functional feature commonality*, *infrastructure utilization*, *invocation time*, *service stability*, *service accuracy*, *resource coverage* and *robustness of service*. Then, each quality property was defined using the quality scheme concepts to get a feasible abstraction of the CCA quality domain. For example, *invocation time* was included in the CCA quality scheme as an instance of *software attribute* because it refers to a measurable property –and the *software attribute* is defined as "an entity that can be verified or measured in the software product" [44].

Ones all quality properties were defined as elements of the quality scheme, each software attribute was related to a metric that defines its measurement method. These metrics were obtained from literature [8]. The ontology concepts allow defining properly the metrics documentation. It also allows studying metrics relationship with aims to obtain information related to their application. In order to complete the definition, each pair (*attribute*, *metric*) was related to a quality subcharacteristic of the ISO/IEC 25010 quality model.

Table 2 resumes the CCA quality scheme used as evaluation goal of the M&S framework. The elements highlighted in italics depict the original quality properties to be evaluated in CCA.

3.2 Software architecture: design and patterns

Architectural design is a creative process where you design a system organization that will satisfy the functional and non-functional requirements of a system [45]. Several authors argue that architectural decisions are the core of software architectures [46–48]. Jansen y Bosch [49] define an *architectural design decision* as "a description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture". Software architectures have evolved from structural representations to decisions centered design models [50]. Therefore, the design patterns aid in documenting and communicating proven design solutions to recurring problems [38]. Given that non-functional requirements play a fundamental role when software architects need to make informed decisions [51], the architectural patterns show singular power in linking functional and non-functional consequences of an architectural approach together [52].

An architectural style defines a family of systems in terms of a pattern of structural organization. From this definition, architectural patterns solve regular problems by linking architectural elements into structural descriptions. Specifically, architectural styles determine the vocabulary of components and connectors that can be used in instances of the style, together with a set of constraints on how they can be combined.

Researches have study architectural patterns over traditional software products for years [53, 54]. However, the architectural styles useful in the CCA domain are not fully understood yet. A prominent approach in this field is the one proposed in [33], where authors describe architectural patterns that allow modeling CCA architectures using a restricted set of architectural styles. This approach involves both infrastructure and application design. Therefore, it gives a complete definition of the elements required for building CCA architectures. In this sense, CCA application patterns allow building web applications solutions by abstracting the infrastructure components into independent ideal scenarios (scenarios where infrastructure components execute software requirements without any constrain). Therefore, such patterns are useful to identify the set of components and connectors that provide an appropriate abstraction of the CCA software architecture.

With aims to define a representation of the CCA software architecture, the application patterns proposed in [33] were used as start point for the architectural design. Hence, a subset of patterns was analyzed in terms of their components and connectors to get a full definition of their structure. These architectural elements were used to build a conceptual metamodel that allows to validate the CCA patterns over architectural designs. Specifically, three types of components were identified:

(1) *Application components* Architectural elements used to define the functional requirements of the CCA. There are two types of application components: components used to define domain functionalities (domain-specific components), and components frequently used as standard templates in the development of CCA (e.g. load balancer and message queue).
(2) *Management components* Architectural elements used to automatically manage the behavior of the

Table 2 CCA quality scheme

Quality characteristic	Quality subcharacteristic	Software attribute	Metric	
			Name	Equation
Performance efficiency	Time behavior	*Invocation time*	Time behavior from user perspective	$TBU = \frac{ET}{TSIT}$ ET = *execution time*[a] TSIT = *total service invocation time*
	Resource utilization	*Infrastructure utilization*	Hardware resources utilization	$HRU = \frac{AR}{PR}$ AR = *amount of allocated resources* PR = *amount of pre-defined resources*
Reliability	Maturity	*Service accuracy*	Replies accuracy	$RA = \frac{CR}{TR}$ CR = *number of correct responses*[b] TR = *total number of requests*
	Availability	*Robustness of service*	Service robustness	$SR = \frac{AT}{TT}$ AT = *available time*[c] TT = *total time*
	Fault tolerance	*Service stability*	Coverage of fault tolerance	$CFT = \frac{FNF}{TF}$ FNF = *number of faults without becoming failures* TF = *total number of faults*
			Coverage of failure recovery	$CFR = \frac{RF}{Tf}$ RF = *number of failures remedied* Tf = *total number of failures*[d]
	Scalability	*Resource coverage*	Coverage of scalability	$COS = \frac{1}{k} \times \sum_{i=1}^{k} \frac{AR_i}{TRR_i}$ k = *number of allocation requests* ARi = *amount of allocated resources of the ith request* TRRi = *total amount of requested resources of ith request*
Maintainability	Reusability	Service customizability	Coverage of variability	$CV = \frac{VP_{CCA}}{VP_D}$ VP_{CCA} = *number of variation points realized in the application* VPD = *number of variation points in the domain*
		Functional feature commonality	Functional commonality	$FC = \frac{1}{n} \times \sum_{i=1}^{n} \frac{RFC_i}{Tr}$ n = *number of functional features* RFC_i = *number of requirements applying the ith functional feature* Tr = *total number of requirements analyzed in the domain*
		Non-functional feature commonality	Non-functional commonality	$NFC = \frac{1}{m} \times \sum_{i=1}^{m} \frac{RNFC_i}{Tr}$ m = *number of non-functional features* $RNFC_i$ = *number of requirements applying the ith non-functional feature* Tr = *total number of requirements analyzed in the domain*

[a] ET = TSIT – WT where WT = *waiting time*

[b] CR = TR – IR where IR = *number of incorrect responses*

[c] AT = TT – FT where FT = *service failure time*

[d] Tf = TF – FNF where TF = *total number of faults*

Table 3 Elements used for modeling the software architectures of CCA

Type	Architectural element		Description
	Type	Subtype	
Component	Application	Domain-specific	Its behavior is explicitly defined using a sequence of functional components
		Load balancer	Determines the number of synchronous accesses to the CCA
		Message queue	Determines the number of asynchronous accesses to the CCA
	Management	Elastic load balancer	The number of synchronous accesses to the application is used to determine the instances of the component
		Elastic queue	The number of asynchronous accesses to the application is used to determine the instances of the component
		Provider adapter	Wraps the provider interfaces into an abstract interface to be used within the scope of the distributed application
		Configuration manager	Application components should use a centrally stored configuration to provide a unified behavior
		Elastic manager	Monitors the utilization of cloud resources on which application component instances are deployed
	Functional	Processing	Processing functionality is split into separate function blocks and assigned to independent components
		Batch processing	Requests are delayed until environmental conditions make their processing feasible
		User interface	Bridge between the synchronous access of the human user and the asynchronous communication used with components
		Data access	Coordinates data manipulation if different storage offerings are used
		Data abstractor	The style of data representation is adjusted to allow data retrieved from storage offerings to be eventually consistent
		Idempotent processor	Ensures that duplicate messages and inconsistent data do not affect application functionality
Connector	External link	Unidirectional	Connection that links application components in a single direction
		Bidirectional	Connection that links application components in two opposite directions
		Special	Links among management and domain-specific components that handle special data, such as number of instances and resources allocation
	Internal link	Sequential	Links among functional components that determine the execution flow of domain-specific components

CCA by monitoring the behavior of the application components.

(3) *Functional components* Architectural elements that exhibit elemental responsibilities used to define complex functionalities in application components.

Table 3 resumes the main elements included in the UML description of the metamodel. Such UML description defines the set of classes, relationships and attributes used for modeling the CCA software architecture. In order to include the design constraints related to the architectural patterns, the UML description was complemented with OCL constraints. Such constrains help to verify the architecture designed from the UML description in terms of the CCA patterns.

Given that architectural components are defined in terms of their behavior as part of the pattern, the M&S framework uses the elements defined in the metamodel as functional sketch of the simulation model.

4 The simulation framework for cca architectures

Figure 2 depicts the structure of the M&S framework designed to estimate the quality of CCA products using the architectural design.

The infrastructure of the framework is defined in Java. Specifically, the Eclipse IDE[1] is used at the bottom layer to support the modeling and simulation tasks.

For modeling the CCA architecture, the framework employs a CCA plugin for Eclipse developed using the components and connectors defined in the CCA metamodel (described in Sect. 3.2). Such plugin was implemented using two projects from the Eclipse Modeling Project[2]: EMF/GMF [55, 56] and Sirius.[3] The Eclipse

[1] Available at http://www.eclipse.org/.

[2] Available at https://eclipse.org/modeling/.

[3] Available at https://eclipse.org/sirius/.

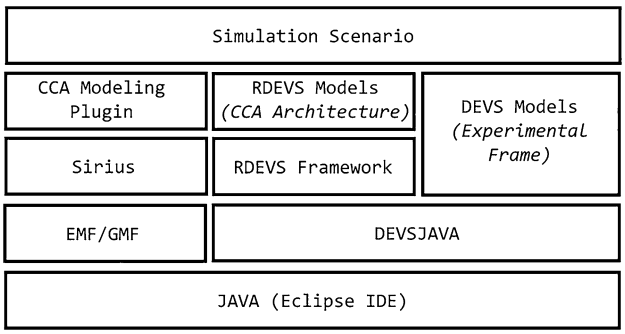


Fig. 2 Software modules that compose the M&S framework

Modeling Framework (EMF) was used to deploy the UML description of the metamodel in terms of Java classes, while the Graphical Modeling Framework (GMF) was used to define the graphical notation attached to architectural components. Additionally, the Sirius project was adopted to allow building a CCA architecture using the graphical elements defined for the metamodel. Section 4.1 presents this graphical modeling tool based on Eclipse.

For simulation purposes, the framework employs DEVSJAVA [57–59] which support the execution of models written in Java. Moreover, the CCA framework uses the Routed DEVS (RDEVS) framework detailed in [11] to structure the simulation models of the CCA architecture. The Routed Discrete Event System Specification formalism [11] is a subclass of DEVS [10, 60] that provides a solution for routing problems over discrete event models. The software architecture simulation can be studied using the routing problem perspective. Then, RDEVS models allow defining the behavior required for each architectural element in terms of their functionality (without worry about the interactions among components). In this case, the routing information manages the events flow. Then, a unique model can be used in several simulation instances surrounded with different routing information in order to get distinct CCA situations.

The CCA simulation scenarios combine simulation models specified using DEVS and RDEVS formalisms. While RDEVS models represent the CCA architecture to be evaluated, the DEVS models are used to explicitly define the experimental frame needed to control the quality estimation. Sections 4.2 and 4.3 describe how RDEVS formalism is used for building the CCA simulation model along with the structure designed for modeling the simulation scenarios.

4.1 Software modeling tool for designing CCA architectures

The software modeling tool was designed as a graphical representation of the CCA metamodel. The UML description was implemented as an Ecore model using the EMF plugin. Moreover, the OCL constrains were added to the Ecore model as a set of constrains that verifies the correctness of the metamodel instances (i.e. the CCA models). The graphical notation of the architectural elements was added with the GMF tool. Both specifications (Ecore model and graphical notation) were linked in a single software plugin for Eclipse using Sirius as intermediate.

Given that software architectures are defined as graphical designs, the plugin developed allows modeling CCA software architectures through the graphical notation of the architectural elements. Therefore, architects can use these elements to build their designs without carry out an explicit instantiation of the metamodel. Figure 3 shows a screenshot of the CCA plugin that highlights four sections. The project explorer (Sect. 1) details the Eclipse project where the architectural design is build. The properties of the project are defined by the architect in order to identify the CCA related to the design. Section 2 is the design area in which the architect must depict the CCA architecture. The available architectural elements are detailed in the tools palette (Sect. 3). This palette contains the graphical representation of all architectural elements defined in the metamodel (i.e. components and connectors) grouped by type and subtype. Finally, the table detailed in Sect. 4 allows defining properties for each architectural element included in the CCA design.

In order to define an instance of the metamodel, the architects build a CCA design using the tools palette and the properties table. The compliance of their designs can be verified employing the OCL constrains that ensure the correctness of the architectural patterns. This gives a structural evaluation of the design. Such evaluation provides an early assessment of the CCA architecture designed in terms of well-established patterns that ensure a good implementation of non-functional requirements. If the plugin detects pattern violations, it shows a warning message to the architect. This message does not prohibit continuing the quality evaluation. The architect should decide whether to maintain the actual design or change it before executing the quality evaluation.

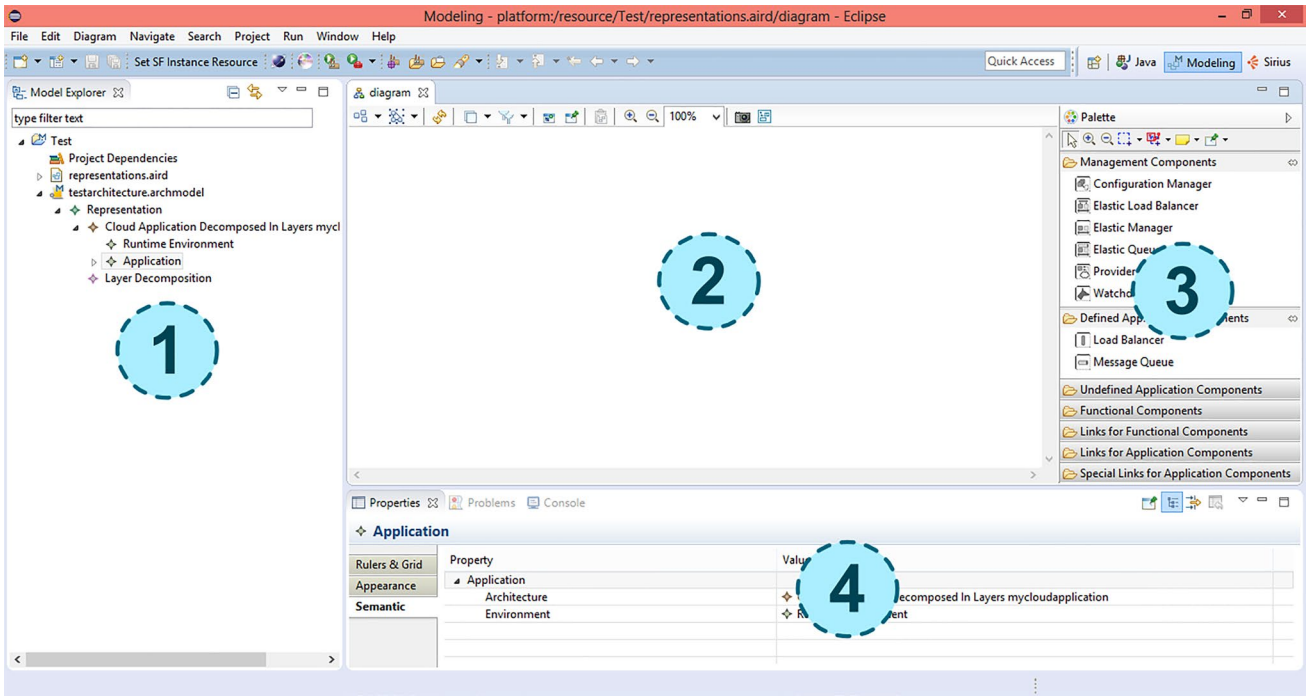


Fig. 3 CCA modeling software tool

4.2 RDEVS as modeling and simulation formalism for CCA software architecture

Software architectures can be seen as a set of components along with the connections among them. Such connections are defined in terms of interactions. That is, two architectural elements A and B are connected if and only if they need to interact with each other to solve some type of user request. Therefore, architectural connections represent all possible interactions among components.

When a user request needs to be solved, a subset of architectural interactions is used to navigate among components. Moreover, in a CCA scenario multiple replicas of the same component are distributed among all available resources. If each architectural component is reflected as a simulation model with its own behavior, then all its replicas must exhibit the same behavior. However, although there is a defined behavior for each type of component, the interactions among them depends on the user request. Then, the requests flow among components depends on the replicas available to execute the processing required for solving the user petition. That is, the same type of user request can take different routes (i.e. to be processed by different replicas) even when all the requests are solved in the same cloud computing environment. As a result, the route of each request (that is, the events flow) is given by the structural connections among components but, also, it is influenced by the available replicas. Therefore, the components behavior should consider routing information related to its replicas. Only the connections with feasible replicas for the request type will be valid.

In this context, the interactions are defined structurally at the architectural level but their effective use as routes is given at request level. These properties allow studying the simulation of CCA software architectures as a routing problem and, therefore, the RDEVS formalism proposed in [11] is applicable for building the simulation models.

In order to define a simulation model based on the architectural design, each simulation model required for the CCA simulation is obtained by mapping the components of the metamodel with a defined behavior. Such mapping allows building independent models that can be coupled following the structural definition (that is, the CCA software architecture) using the metamodel connections. Moreover, since the RDEVS models use routing information to manage the events flow, the simulation models can use the request type as support information to redirect events across couplings.

4.2.1 Getting RDEVS models from the CCA software architecture

The RDEVS formalism is based on three types of simulation models: *essential model*, *routing model* and *network model*. Each model helps to define a structure among elements that provides a simulation scenario for the modeling a routing problem.

The *RDEVS essential model* defines the behavior of routing components. Formally, its structure is equivalent to the DEVS atomic model definition. In order to represent the CCA simulation, the behavior of all architectural components defined in the CCA metamodel were explicitly formalized as *essential models*. Moreover, each type of architectural component was defined as an essential model that exhibits its own behavior according to its architectural definition. Since *management components* require an infrastructure deployment to be managed, an ideal scenario was used as a first attempt for quality evaluation. Each architectural element defined as *management component* in the metamodel was detailed as an essential model that helps to organize the instances of *application components*. Meanwhile three types of essential models were defined in order to represent *application components*. Two of them were used to represent the behavior of common components (*load balancer* and *message queue*). The third one was defined abstractly as the functional definition of *domain-specific components* (detailed by a sequence of functional components) in terms of the behavior expected by the interactions of its internal functions (including all possible internal states). From this perspective, each function (represented architecturally as *functional component*) can be executed as *processing*, *processing with faults* or *inactive by failure*. Then, the *essential model* attached to a domain-specific component is obtained by translating all the possible internal states that follows the sequence of functional components that compose it. Each *functional component* is defined considering its own behavior in order to maintain its responsibility. By following this approach, the *functional components* provide the behavior definition of *domain-specific components* without need to build new simulation models for specific CCA modules. Moreover, the translation proposed among *functional* and *domain-specific components* helps to build automatically the simulation model of any software functionality required in CCA architectures without need to provide any other information about its execution.

In RDEVS formalism, the *routing model* defines the basic simulation component used as part of the routing process. It embeds the routing component behavior as an operational description of its own behavior. Then, the *essential model* is embedded in the definition of the *routing model* as one of the main components required for its specification. Besides the *essential model*, the *routing model* includes information related to the input events acceptance and the output events destination. Such information is called *routing information*. Therefore, the *routing model* is defined as a simulation model that exhibits the *essential model* behavior only when input events should be accepted agreeing to its *routing information*. Furthermore, the output events are created following the output function of the *essential model* and including the *routing information* required to get their destination. Hence, several *routing models* can be defined using the same *essential model* with different *routing information*. As consequence of this combination, each *routing model* will exhibit different behavior in terms of the input event acceptance and the output events destination. Such flexibility allows building distinct simulation model configurations employing the same definition of the component behavior.

If the CCA simulation model is seen as the routing process to be solved in terms of a routing problem, each node composing the routing problem should be seen as the equivalent *RDEVS routing model*. Therefore, each replica of the architectural components available over the infrastructure resources should be interpreted as a *routing model*. Such *routing models* are composed by the *essential models* defined for the related architectural component.

Finally, the *RDEVS network model* describes a complex simulation model that provides a structure over the individual components in order to depict a routing process among them. Its definition includes a set of *routing models* and the couplings among them. Each *routing model* can be interpreted as a node. The couplings among nodes are defined as all-to-all connections in order to leave the routing task to the *routing information* defined in each node. In this sense, the simulation model that depicts the structure of the CCA software architecture was defined as a *network model* composed by the *routing models* obtained from the replicas designed by using the set of available components.

Table 4 summarizes the RDEVS models designed for each architectural component included in the CCA metamodel. By using the translation proposed in the table, the simulation models designed with RDEVS formalism provides a generic structure in terms of the architecture without consider the events flow as part of the defined behavior. Therefore, the routing information can be modified without need to adapt the behavior of the essential models that represents the architectural components.

4.3 CCA simulation scenario as a combination of DEVS and RDEVS models

A simulation scenario was defined to provide a solution for the quality estimation based on the CCA simulation model. Such scenario is depicted in Fig. 4. As figure shows, the simulation model to be evaluated is called *CCA Architecture*. Such model is obtained by automatically translating the CCA software architecture (that is designed using the CCA metamodel) to the equivalent simulation model defined as a *RDEVS network model*. However, given that the routing problem is attached only to the software architecture model, the *experimental frame* was designed

Table 4 RDEVS models that represent each type of element required for the CCA simulation

CCA software architecture			RDEVS model that represents it
Element	Type	Subtype	
Architectural component	Application	Domain-specific	An essential model built using the sequence of states defined from the functional components included in the component
		Load balancer	An essential model specifically defined to exhibit the load balancer behavior
		Message queue	An essential model specifically defined to exhibit the message queue behavior
	Management	Elastic load balancer	An essential model defined specifically to represent the behavior of each management component over an ideal infrastructure
		Elastic queue	
		Provider adapter	
		Configuration manager	
		Elastic manager	
	Functional	Processing	A state inside the essential model that represents the domain-specific component where the functional component is used as part of the defined sequence. Each component maintains its functional definition in order to provide a specific functionality
		Batch processing	
		User interface	
		Data access	
		Data abstractor	
		Idempotent processor	
Replica	Architectural component		A routing model composed by the essential model defined for the architectural component detailed in the replica along with its routing information
Software architecture	–		A network model composed by a set of architectural replicas defined as routing models

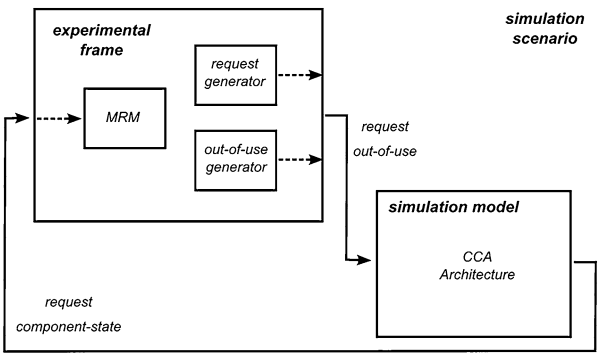


Fig. 4 Simulation scenario used in the framework

using DEVS formalism [10]. Both models are coupled in terms of three types of events: *request*, *out-of-use* and *component-state*.

The *request* event is used to test the *CCA Architecture* employing different types of user request. When this type of event is taken as input event in the *CCA Architecture*, the evaluation process is performed over such request. After finishing its processing, the *request* event is sent as output event of the *CCA Architecture*. This output event includes all the measures obtained during the processing. If the state of architectural components changes during the processing, the *CCA Architecture* sent a *component-state* event to notify this situation to the *experimental frame*.

In this context, the *experimental frame* was defined as a *DEVS coupled model* composed by three internal models: *request generator*, *out-of-use generator* and *MRM* (*metrics measurement model*). The *request generator* was defined as a *DEVS coupled model* that produces the user requests to be processed in the architecture. Five types of simulation models were defined in order to get distinct behaviors. Most user behaviors can be studied from the continuous perspective (as workloads). Then, the simulation models proposed in [61] combines continuous and discrete models with aims to describe common workloads performed by CCA users. These models were included in the M&S framework as available *request generator* models to be used in simulation scenarios.

The *out-of-use generator* and *MRM* were designed as *DEVS atomic models*. The *out-of-use generator* produces an *out-of-use* event that ends the simulation execution. This event is produced when the remaining time is zero. When the *CCA Architecture* receives an *out-of-use* event, it stops the processing and leaves the simulation models in inactive state. This mechanism ensures that the metrics measurement process is executed only when simulation models are active.

Table 5 Direct metrics obtained by the framework during the simulation

Variable	Description	Unit
ET	User request processing time	Time
TSIT	Total time to solve a user request	Time
TR	Number of requests solved	Requests
IR	Number of requests with incorrect response or solution[a]	Requests
FT	Inactive time[b]	Time
TT	Operative time	Time
FNF	Number of faults that are not failures[c]	Faults
TF	Number of faults	Faults
RF	Number of failures solved[d]	Failures

[a]A wrong solution is given when some architectural component exhibits the *processing with faults* state

[b]The CCA is inactive when all replicas of an architectural component exhibit the *fault* state

[c]A *fault* is a temporal issue that can be solved. A *failure* is a *fault* that cannot be solved and, therefore, the component needs to be replaced by a new replica

[d]Replicas created to solve a *fault*

Finally, the *MRM* model is used to calculate the set of metrics presented in Table 2. The events flow proposed between the *CCA Architecture* and *experimental frame* allows measuring direct metrics (Table 5). These metrics are grouped to get the final quality evaluation. Such quality evaluation is centered in the original quality properties modeled in the quality ontology [39]. Given that the *MRM* model uses a specific set of events to estimate multiple quality metrics; the M&S framework provides a solution that allows evaluating several quality properties using the software product architecture. The measures obtained during the simulation are stored in a CSV file to provide an analyzable output.

5 Case study: building the simulation model for two CCA software architectures

Most web applications use N-tier architecture style to arrange its components. In this context, two-tier and three-tier distributions are commonly used in CCA software architectures. Both distributions were studied using the M&S framework to ensure the accuracy of the simulation models.

A *two-tier cloud application* implements the CCA separating the data tier from the presentation and business logic tier. Meanwhile, a *three-tier cloud application* needs to be able to scale presentation, business logic, and data handling independently (because the requirements of these functions regarding the necessary number of application component instances to handle workload differ greatly). Figure 5 shows both distributions using the patterns proposed by Fehling et al. [33].

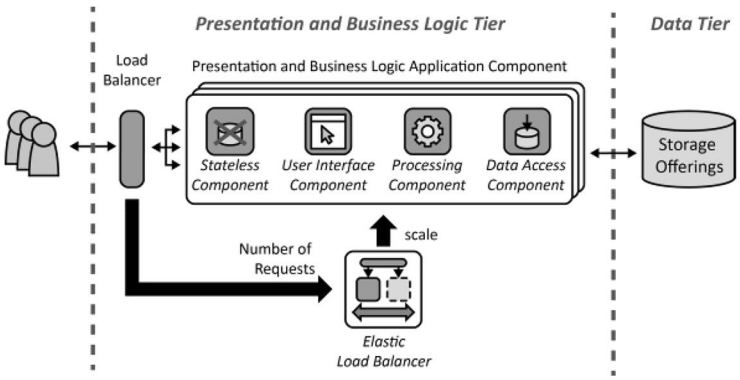
Even when both architectures use the same components, the communication paths that describe the expected execution must know the number of replicas available for the processing. In the *two-tier cloud application* (Fig. 5a) an elastic load balancer determines the number of required instances of application components and provisions and decommissions them as needed (to ensure an appropriate number of instances for the number of user accesses). These application components handle accesses by users, workload processing, and data access in a holistic fashion. Instead, in the *three-tier cloud application* (Fig. 5b) the elastic load balancer provisions or decommissions presentation application components using the number of requests sent to the presentation tier. After handling the request and validating the necessary user inputs the presentation component sends the request to the message queue for the business logic tier. When one of the business logic components is idle, it consumes the message from the queue and processes the message. After processing the message, it sends the data to be stored to the message queue for the data tier.

Then, the simulation models that denote both CCA software architectures need to represent multiple instances of some components in order to direct the events flow. However, the architectural components scale without modifying the architecture design. So, the simulation models should be tested with several numbers of replicas to study their performance in multiple simulation scenarios.

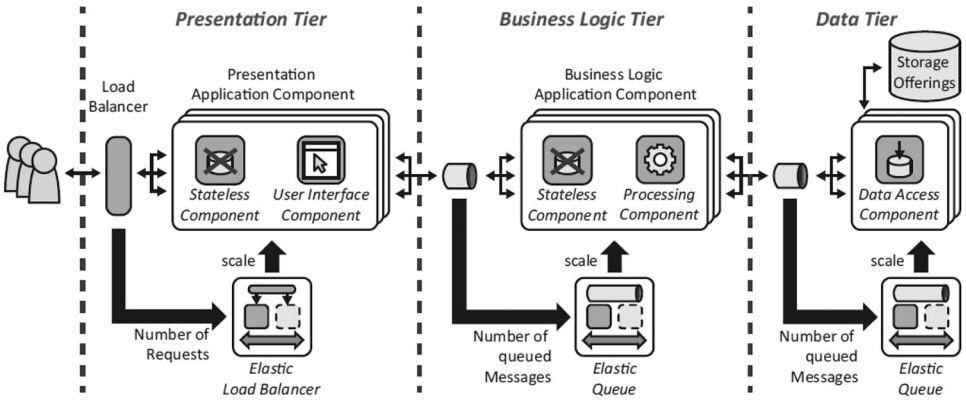
In order to evaluate the CCA software architectures, both structures were modeled with the M&S framework. Figures 6 and 7 shows the architectural models designed using the plugin described in Sect. 4.1. If the architecture is built from scratch, the modeling plugin can be used also as software support tool for the design task. Given that both CCA architectures are conceived as predefined patterns, the modeling was performed only to obtain the component structures required for building the simulation models.

After defining the architectures, the M&S framework is able to apply automatically the set of translation rules to get the equivalent RDEVS network model for each CCA architectural design. Table 6 resumes the number of RDEVS models obtained in each case by applying a defined number of replicas for application components.

As table shows, building the simulation models applying an automatic translation reduces the number of tasks assigned to the software architect. If the task of building the simulation models manually had assigned to the software architect, the time spend in the design phase will



(a) Two-tier architecture.



(b) Three-tier architecture.

Fig. 5 CCA software architectures commonly used in N-tier distribution modeled by Fehling et al. [33]

increase. By using the framework, the architect only needs to configure some parameters and the simulation models are automatically created from the architectural design according to the number of replicas desired for each application component to be scale out.

Then, the framework builds the simulation models using the translation rules designed for the architectural components involved in the architecture design. Such translation leads to simulation models that ensure the accuracy of the RDEVS formalism. Moreover, the translation process is hidden to architects in order to maintain independence between modeling and simulation tasks. This independence provides an appropriate separation of skills required for CCA software architects. That is, the software architects only need to develop the architecture using the available components without worry about how such design will be mapped to the RDEVS simulation model. Therefore, this abstraction allows both architects and developers to use the M&S framework without any knowledge of discrete event simulation.

The RDEVS network models obtained from the translation process executed from the architecture design were used in several simulation scenarios to evaluate their performance. Multiple simulations were carried out with aim to validate the accuracy of the simulation models developed for representing the CCA architectures. Thus, the behavior of the proposed simulation models is related to the benefits of the architectural design patterns used as basis while building the architecture.

In this context, the main characteristics derived from a greater number of tiers used as part of an architectural design include: (1) the possibility of replicating and scaling independently each of the defined application components; and (2) the impact of the faults and failures over the replicas scaled out at application level.

In a two-tier architecture, the only way to process a request is using an instance of the application component (in Fig. 5a such component is depicted by the *presentation and business logic tier*). Once the available resources are full (therefore, they cannot allocate new replicas), new requests cannot be processed until some component is released from its assigned task. On the other hand, the three-tier architecture (Fig. 5b) can scale out each tier as independently. Such distribution allows release application components while different functions are executed.

Furthermore, in the two-tier architecture each replica built for an application component has the responsibility

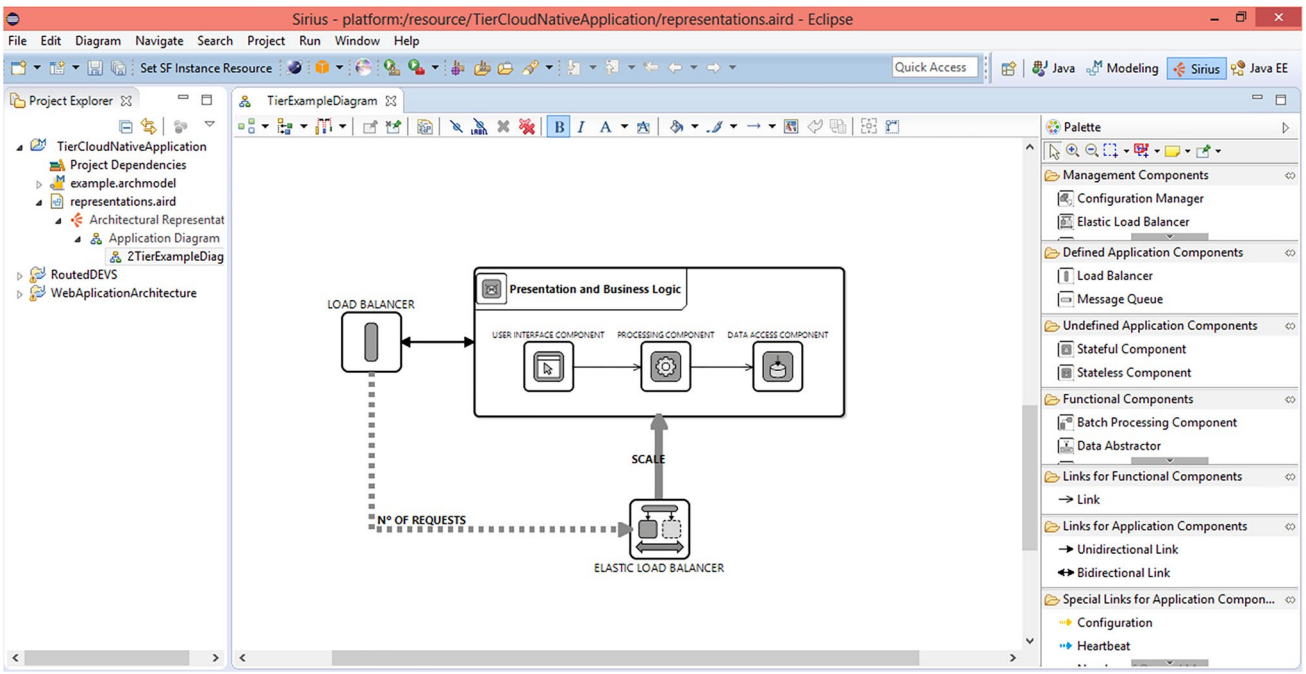


Fig. 6 Screenshot of the two-tier architecture modeled with the framework

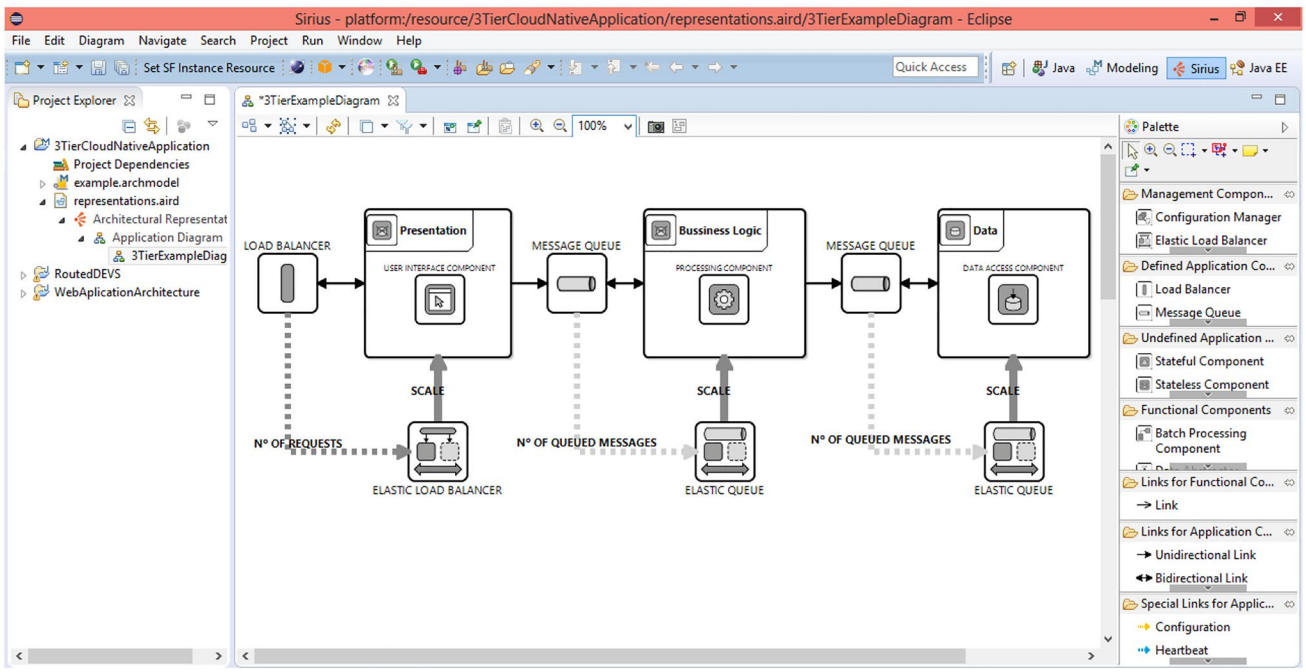


Fig. 7 Screenshot of the three-tier architecture modeled with the framework

of executing all the functions included in its behavior. The presence of some fault or failure during its execution leads to an application error that must be solved because it interferes with a high-level component. However, given that in the three-tier architecture the functions are broken down (giving a set of application components that execute a restricted set of functionalities), the presence of some fault or failure does not necessarily entail an error at application level.

Table 6 Number of simulation models created automatically by the M&S framework with aims to represent two and three tiers architectures

Model	Number of RDEVS models[a]		
	Essential models	Routing models	Network models
Two-tiers application	3	2 + N	1
Three-tiers application	7	6 + 3N	1

[a]Number of RDEVS models required to build a single network model composed by N instances (replicas) of the application components

In order to exploit both scenarios, several simulations were executed using different availability for the maximum number of replicas to be allocated. Four scenarios were carried out with *numberOfReplicas* = 5, *numberOfReplicas* = 10, *numberOfReplicas* = 20 and *numberOfReplicas* = 50. In all scenarios executed, the *request-generator* model was configured with the same type of user behavior in order to allow results comparison.

First, the number of requests attended by the CCA was studied. Figure 8 presents the number of user requests processed inside the CCA during the simulation. As can be seen, in the case of Fig. 8a there is a steep slope that disappears as the number of replicas available increases (Fig. 8b–d). This can be due to two reasons: (1) lack of application components availability; or (2) failure states in undefined application components. In the first case, incoming requests are discarded as a consequence of the lack of replicas that must process their content in order to get a user response. These requests remain in the simulation model, making the slope of the curve grow (Fig. 8a). Another possibility is that some application component goes into a failure state. Therefore, the requests assigned to such component are discarded (because the component is not able to process them). Then, these requests are lost within the application, making the slope of the curve grow (Fig. 8a). By analyzing the metric *Tf* (total number of failures) in different executions with the same configuration it is observed that the existence of this slope is not only derived from the presence/absence of failure states. Even, the slope is present in the absence of faults. Furthermore, as it is illustrated in Figs. 8b–d, if the number of replicas is increased the slope disappears and the number of requests in the application is limited.

A similar behavior is observed for the three-tier architecture (Fig. 9). However, the slope in Fig. 9a is smaller than the one presented in Fig. 8a. The number of requests that

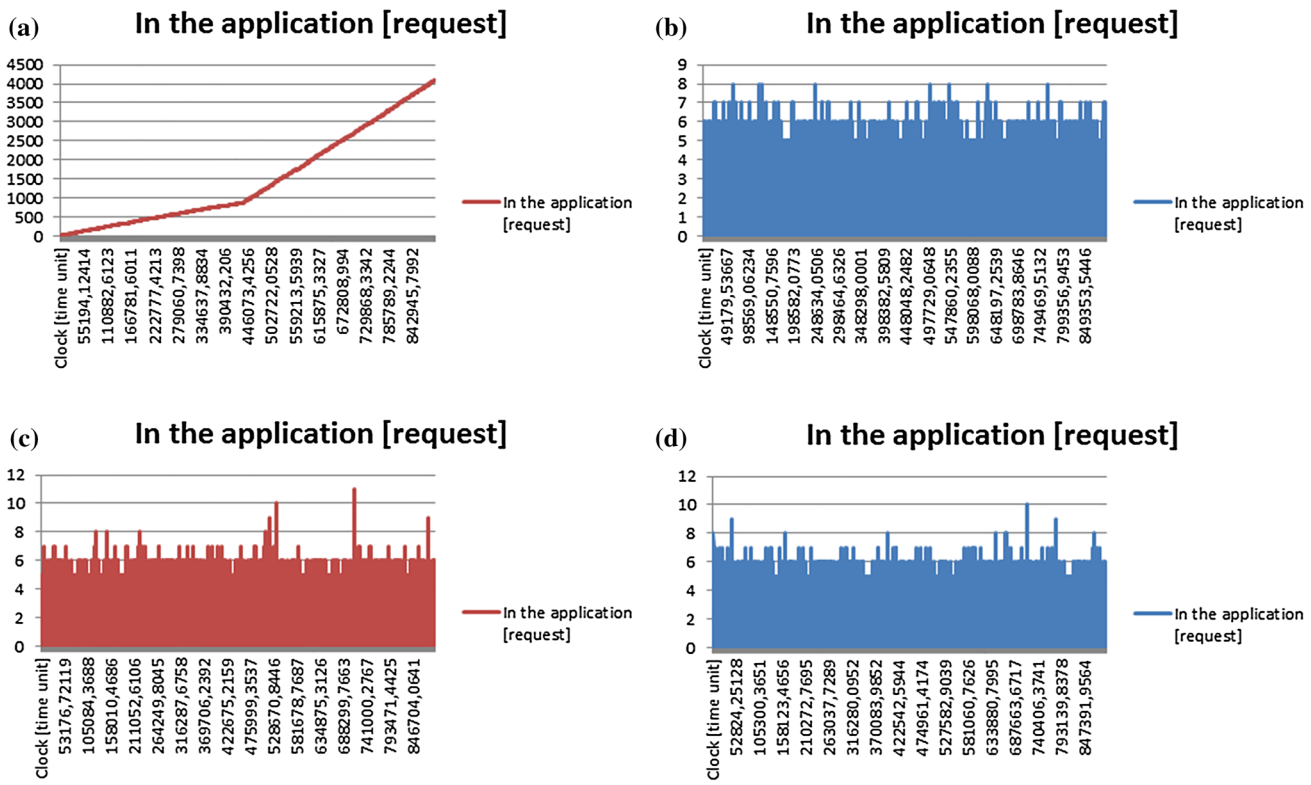


Fig. 8 Requests being process by the two-tiers architecture. **a** *numberOfReplicas* = 5, **b** *numberOfReplicas* = 10, **c** *numberOfReplicas* = 20, **d** *numberOfReplicas* = 50

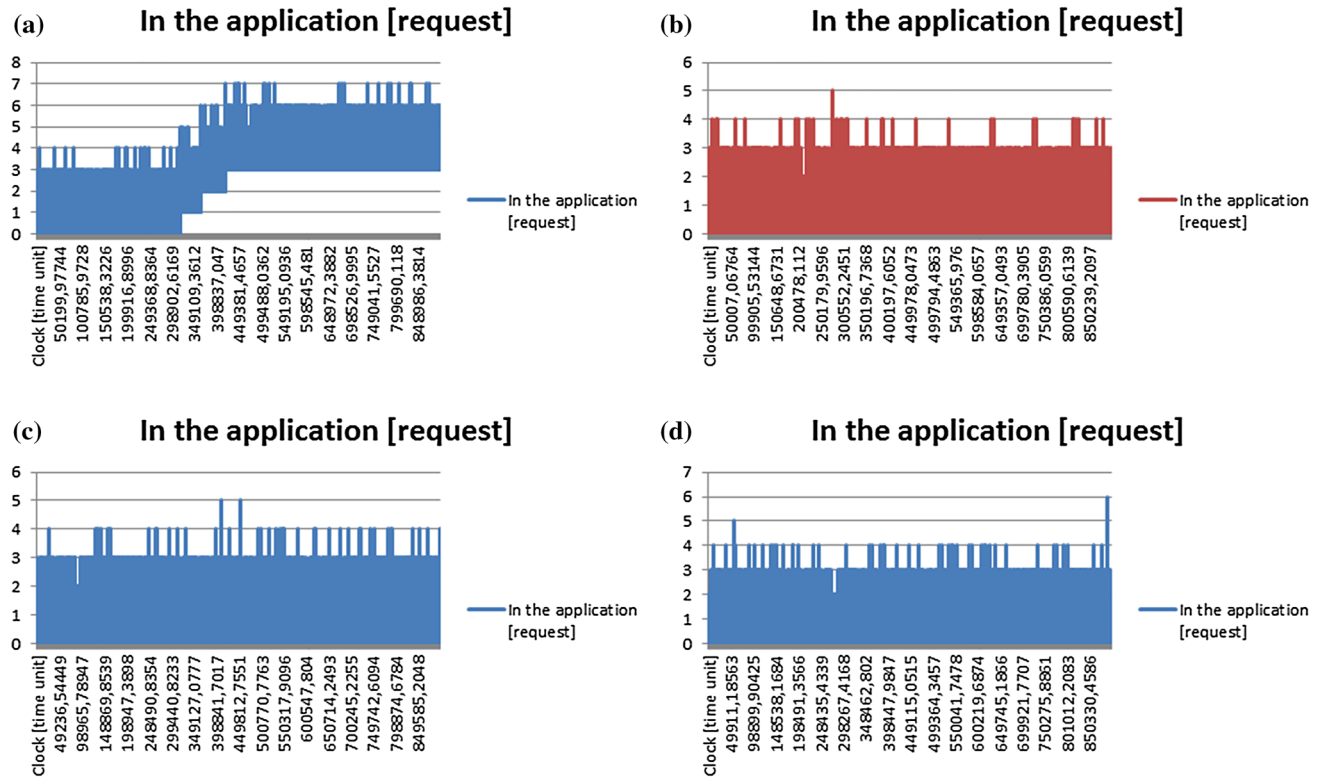


Fig. 9 Requests being process by the three-tiers architecture. **a** *numberOfReplicas* = 5, **b** *numberOfReplicas* = 10, **c** *numberOfReplicas* = 20, **d** *numberOfReplicas* = 50

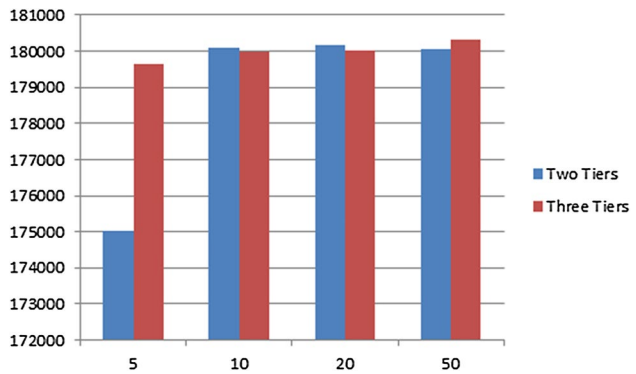


Fig. 10 Comparison between the average requests solved by both CCA architectures

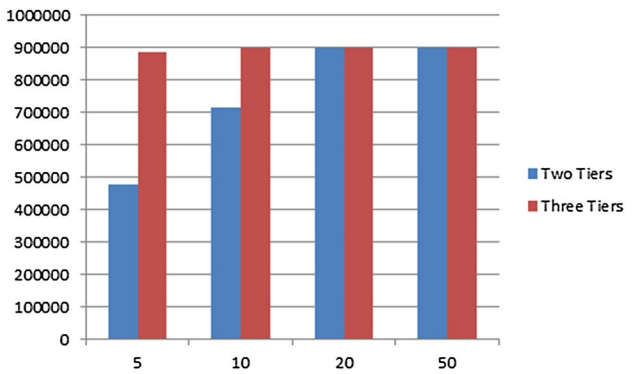


Fig. 11 Comparison between the average requests solved by both CCA architectures in a new configuration of the *request-generator* model

are discarded in the three-tier architecture is smaller than in the two-tier architecture. Hence, the performance of the three-tier architecture is better than the architecture of two-tiers.

Figure 10 shows the average number of requests answered *TR* (total number of requests) in the cases analyzed over an increasing number of available replicas. As can be seen, when the number of replicas increases from 5 to 10, there is a significant difference in the value of TR for the two-tier architecture. This is because some of the discarded requests in a configuration with *numberOfReplicas* = 5 are processed correctly in a configuration with *numberOfReplicas* = 10.

In the case of five available replicas it is evident that the model of the three-tier architecture has a better performance for the general processing of the requests (Fig. 10). This is consistent with the analysis performed in terms of the requests processed/discarded in both design

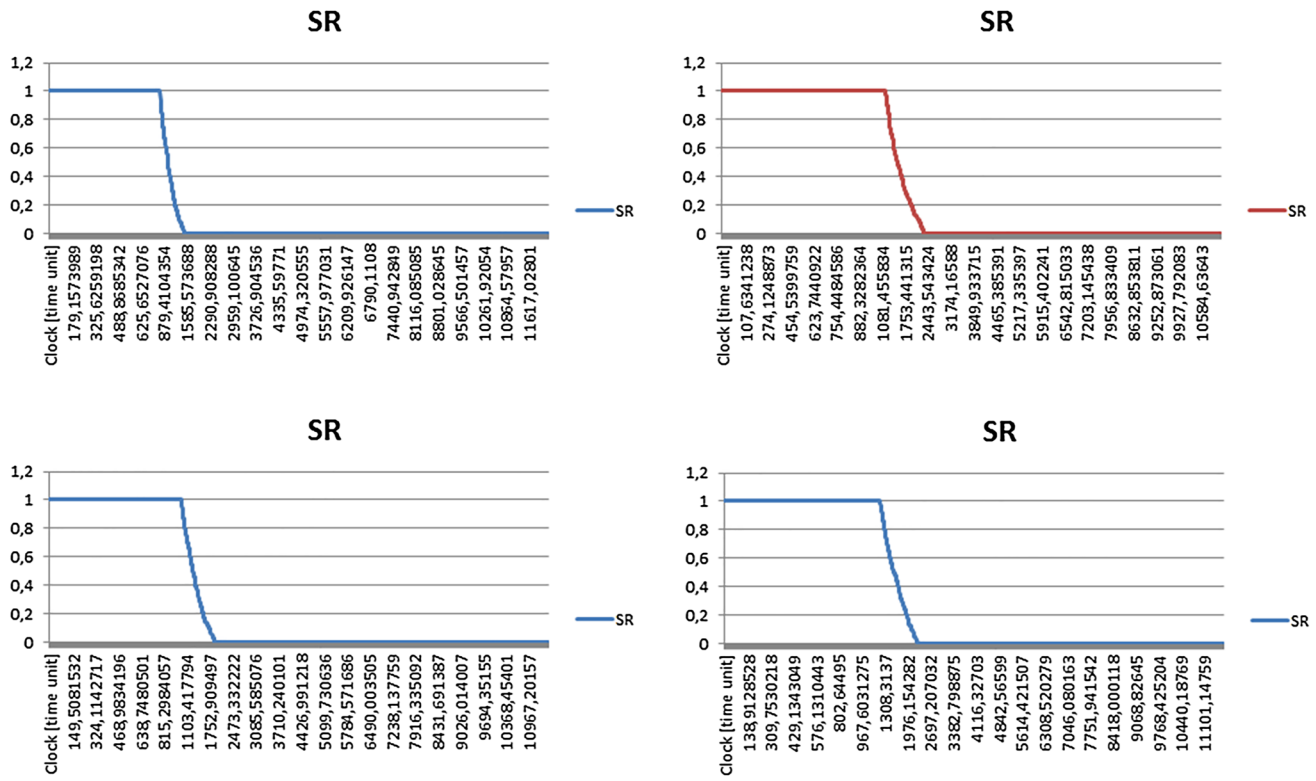


Fig. 12 SR in the two-tier architecture with *failureProbability* = 0.5

structures. However, in cases of 10, 20 and 50 replicas the value of *TR* is similar in both architectures. In order to verify that this similarity is due to the fact that the configuration of the models does not allow a greater scalability of the requests, alternative scenarios were executed modifying the behavior used in the *request-generator* model. Figure 11 presents the same analysis using a new configuration in the *request-generator* model. As figure shows, the two-tier architecture model presents significant differences with respect to the three-tier model when there are 5 and 10 replicas of the application components. In cases of 20 and 50 replicas, the number of instances implemented seems to be enough to process all the requests. Then, the difference between the *TR* values is not visualized.

Then, the analysis performed in order to study the behavior of both architectures from the number of replicas required for application components is reliable in comparison with the expected behavior of the design patterns they represent.

In order to test the handling of failure states, several simulations were also performed. The simulations were executed with different values of the *failureProbability* parameter related to functional components included in the architecture (*failureProbability* = 0.5 and *failureProbability* = 1). In all cases, the number of available replicas remained fixed in *numberOfReplicas* = 25.

As an example, Figs. 12 and 13 show the results obtained for the *SR* (*service robustness*) metric in four executions using *failureProbability* = 0.5. The SR metric refers to the relationship between the total operation time and available time of the CCA service. When all the replicas of the same application component go into failure state, the system is not available and, therefore, it will not be able to process new requests (SR = 0). Then, a value near to 1 indicates that it is highly probable that, at a given time, the system is in service. As can be seen, the three-tier architecture maintains a high SR level (SR = 1) for a period of time greater than the one exhibit in the two-tier architecture. Making the same comparison with *failureProbability* = 1, the three-tier architecture presents better performance than the two-tier model. Such behavior is due to the separation of the responsibilities in the three-tier pattern that provides a smaller impact of failure states at functional component over the CCA. In the case of two tiers, a failure in any of the functional components involved in the design of an application component will result in the failure of the all application. However, in the three-tier architecture the independence among application components provide also independence among functional components. Then, failure states at functional component level will only

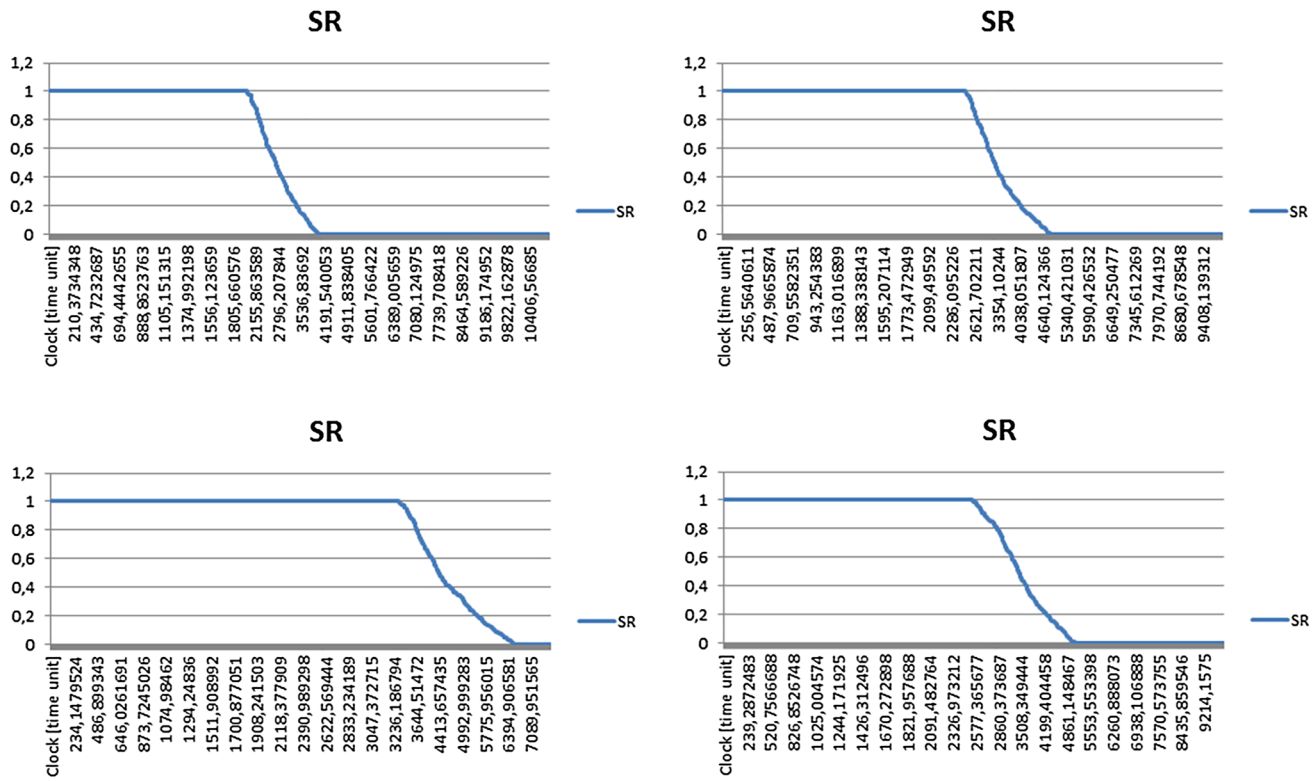


Fig. 13 SR in the three-tier architecture with *failureProbability* = 0.5

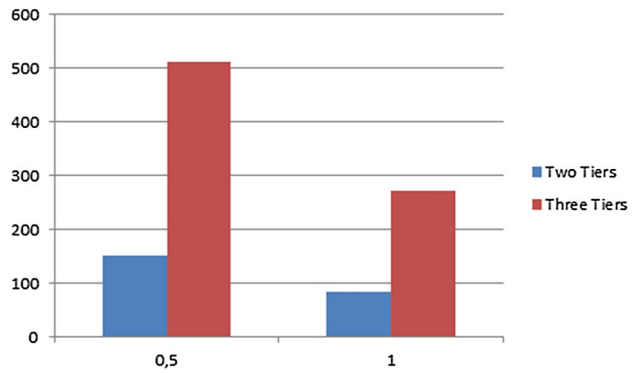


Fig. 14 Average requests processed with failure states

impact on the application component related to it (not to all the application as in the two-tier model).

Moreover, it is observed that when increasing the probability of failure, the number of processed requests is reduced in both cases (Fig. 14). Then, when the model is in failure state the system solves a lower number of user requests. Again, the designed simulation models have reflected a behavior consistent with what is expected in the design patterns they represent.

6 Conclusions and future research

In this paper we present a modeling and simulation framework that provides an evaluation model for quality estimation in early stages of development for cloud computing applications. By using the software architecture as sketch, the framework provides an engineering solution that helps to study the performance of cloud application designs by measuring a defined set of quality properties. As software architectures are defined from architectural patterns, architectural components included in the framework were obtained from the study of real architectural patterns. The translation between architectural components and simulation models is performed as a hidden process that maps predefined simulation models to the architectural structure outlined by the design. Further, the same design can be tested using different number of instances of the application components (defined as component replicas) to study their deployment over the resources.

The framework can be used as support tool in software engineering processes to improve the quality of the final cloud application using an early estimation across the development. One of the main advantages of the framework is the possibility to study the performance of several designs without implementation effort. The software

architect can experiment with the CCA design to improve its final proposal with aims to fulfil the quality expectations. Moreover, the framework can also be used to study how architectural changes can improve the performance of already existing architectures. On the other hand, the limitations are related to the predefined set of architectural components. Even when new components could be added to the metamodel, such addition will always require a new mapping with a new simulation model. Also, although the framework is limited to a defined set of quality measures, since the *CCA Architecture* model is built considering all possible states of architectural components and the events flow is managed by the simulation formalism, new quality properties can be added. Such addition will probably involve changes over the *MRM* model included in the *experimental frame* in order to calculate new metrics.

The authors are currently using the framework to evaluate a variety of cloud applications in order to obtain new quality properties to be included as part of the framework. Thereby, the addition of new metrics combining new simulation models with the existent ones is a current research. Thus, given that the simulation performed by the framework is only attached to software components, the addition of infrastructure resources is another research area to be exploited as future work. Actually, authors are studying the possibility of linking the modeling and simulation framework as a software level description of infrastructure models developed using CloudSim [26].

Funding This work was supported by the Universidad Tecnológica Nacional [Grant Numbers EIUTIFE0003803TC, SIUTIFE0005273TC]; and the National Scientific and Technical Research Council - Argentina [Grant Numbers PIP 112-20170101131CO; PUE 22920160100132CO].

Compliance with ethical standards

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Mell P, Grance T (2011) The NIST definition of cloud computing
2. Hu F, Qiu M, Li J, Grant T, Taylor D, McCaleb S, Hamner R (2011) A review on cloud computing: design challenges in architecture and security. J Comput Inf Technol 19(1):25–55
3. Bera S, Misra S, Rodrigues JJ (2014) Cloud computing applications for smart grid: a survey. IEEE Trans Parallel Distrib Syst 26(5):1477–1494
4. Bouchenak S (2010) Automated control for SLA-aware elastic clouds. In: Proceedings of the fifth international workshop on feedback control implementation and design in computing systems and networks (pp 27–28). ACM
5. Offutt J (2002) Quality attributes of web software applications. IEEE Softw 19(2):25–32
6. Breu R, Kuntzmann-Combelles A, Felderer M (2014) New perspectives on software quality. IEEE Softw 31(1):32–38
7. Institute of Electrical Electronic Engineering (2011) ISO/IEC 25010: systems and software engineering—systems and software quality requirements and evaluation (SQuaRE)—system and software quality models
8. Lee JY, Lee JW, Kim SD (2009) A quality model for evaluating software-as-a-service in cloud computing. In 7º ACIS international conference on software engineering research, management and applications (pp 261–266). IEEE
9. Roy J, Suryn W, Eftekhar SM, Terfas H (2018) Towards a quality evaluation framework for cloud-based applications. SQM XXVI 111
10. Zeigler BP, Muzy A, Kofman E (2018) Theory of modeling and simulation: discrete event & iterative system computational foundations. Academic Press, London
11. Blas MJ, Gonnet S, Leone H (2017) Routing structure over discrete event system specification: a DEVS adaptation to develop smart routing in simulation models. In: 2017 Winter simulation conference (WSC) (pp 774–785). IEEE
12. Samoladas I, Gousios G, Spinellis D, Stamelos I (2008) The SQO-OSS quality model: measurement based open source software evaluation. In: IFIP international conference on open source systems (pp 237–248). Springer, Boston, MA
13. Deissenboeck F, Juergens E, Lochmann K, Wagner S (2009) Software quality models: purposes, usage scenarios and requirements. In: 2009 ICSE workshop on software quality (pp 9–14). IEEE
14. Jagli D, Purohit S, Chandra NS (2017) SaaS CloudQual: a quality model for evaluating software as a service on the cloud computing environment. In: Innovations in computer science and engineering (pp 73–80). Springer, Singapore
15. Bardsiri AK, Hashemi SM (2014) Qos metrics for cloud computing services evaluation. Int J Intell Syst Appl 6(12):27
16. Olsina L, Rossi G (2002) Measuring Web application quality with WebQEM. IEEE Multimedia 9(4):20–29
17. Lew P, Olsina L, Zhang L (2010) Quality, quality in use, actual usability and user experience as key drivers for web application evaluation. In: International conference on web engineering (pp 218–232). Springer, Berlin, Heidelberg
18. Morasca S, Lavazza L (2019) Comparing the effectiveness of using design and code measures in software faultiness estimation. In: Proceedings of the evaluation and assessment on software engineering (pp 112–121). ACM
19. Abdelmaboud A, Jawawi DN, Ghani I, Elsafi A, Kitchenham B (2015) Quality of service approaches in cloud computing: a systematic mapping study. J Syst Softw 101(1):159–179
20. Wnukiewicz KK (2006) The role of quality requirements in software architecture design. Master Thesis, Blekinge Institute of Technology, Sweden
21. Bass L, Clements P, Kazman R (2012) Software architecture in practice. Addison-Wesley Professional, Boston
22. Dobrica L, Niemela E (2002) A survey on software architecture analysis methods. IEEE Trans Softw Eng 28(7):638–653
23. Clements P, Bass L, Kazman R, Abowd G (1995) Predicting software quality by architecture-level evaluation. In: Proceedings of the fifth international conference on software quality (vol 5, pp 485–497)
24. Foster I, Zhao Y, Raicu I, Lu S (2008) Cloud computing and grid computing 360-degree compared. In: 2008 Grid computing environments workshop (pp 1–10). IEEE

25. Ardagna D, Casale G, Ciavotta M, Pérez JF, Wang W (2014) Quality-of-service in cloud computing: modeling techniques and their applications. J Internet Serv Appl 5(1):11
26. Calheiros RN, Ranjan R, Beloglazov A, De Rose CA, Buyya R (2011) CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw Pract Exp 41(1):23–50
27. Wang S, Liu Z, Sun Q, Zou H, Yang F (2014) Towards an accurate evaluation of quality of cloud service in service-oriented cloud computing. J Intell Manuf 25(2):283–291
28. Garg SK, Versteeg S, Buyya R (2013) A framework for ranking of cloud computing services. Future Gener Comput Syst 29(4):1012–1023
29. Chung L, Do Prado Leite JCS (2009) On non-functional requirements in software engineering. In: Conceptual modeling: foundations and applications (pp 363–379). Springer, Berlin, Heidelberg
30. Khan AN, Kiah MM, Khan SU, Madani SA (2013) Towards secure mobile cloud computing: a survey. Future Gener Comput Syst 29(5):1278–1299
31. Rimal BP, Choi E, Lumb I (2009) A taxonomy and survey of cloud computing systems. In: 2009 Fifth international joint conference on INC, IMS and IDC (pp 44–51). IEEE
32. Zhang Q, Cheng L, Boutaba R (2010) Cloud computing: state-of-the-art and research challenges. J Internet Serv Appl 1(1):7–18
33. Fehling C, Leymann F, Retter R, Schupeck W, Arbitter P (2014) Cloud computing patterns: fundamentals to design, build, and manage cloud applications. Springer, Heidelberg
34. Cysneiros LM, Do Prado Leite JCS, Neto JDMS (2001) A framework for integrating non-functional requirements into conceptual models. Requir Eng 6(2):97–115
35. Chung L, Nixon BA, Yu E, Mylopoulos J (2012) Non-functional requirements in software engineering, vol 5. Springer, New York
36. Firesmith D (2003) Using quality models to engineer quality requirements. J Object Technol 2(5):67–75
37. Mairiza D, Zowghi D, Nurmuliani N (2010) An investigation into the notion of non-functional requirements. In: Proceedings of the 2010 ACM symposium on applied computing (pp 311–317). ACM
38. Gross D, Yu E (2001) From non-functional requirements to design through patterns. Requir Eng 6(1):18–36
39. Blas MJ, Gonnet S, Leone H (2017) An ontology to document a quality scheme specification of a software product. Expert Syst 34(5):e12213
40. Gao J, Pattabhiraman P, Bai X, Tsai WT (2011) SaaS performance and scalability evaluation in clouds. In: Proceedings of 2011 IEEE 6th international symposium on service oriented system (SOSE) (pp 61–71). IEEE
41. Khan IA, Singh R (2012) Quality assurance and integration testing aspects in web based applications. Int J Comput Sci Eng Appl 2(3):109
42. Wen PX, Dong L (2013) Quality model for evaluating SaaS service. In: 2013 Fourth international conference on emerging intelligent data and web technologies (pp 83–87). IEEE
43. Zhao B, Zhu Y (2014) Formalizing and validating the web quality model for web source quality evaluation. Expert Syst Appl 41(7):3306–3312
44. Institute of Electrical Electronic Engineering (2001) ISO/IEC 9126-1: software engineering—product quality—part 1: quality model
45. Sommerville I (2011) Software engineering, 9th edn. Addison-Wesley, Boston
46. Kruchten P (2004) An ontology of architectural design decisions in software intensive systems. In: 2nd Groningen workshop on software variability (pp 54–61)
47. Tyree J, Akerman A (2005) Architecture decisions: demystifying architecture. IEEE Softw 22(2):19–27
48. De Boer RC, Lago P, Telea A, Van Vliet H (2009) Ontology-driven visualization of architectural design decisions. In: 2009 Joint working IEEE/IFIP conference on software architecture & European conference on software architecture (pp 51–60). IEEE
49. Jansen A, Bosch J (2005) Software architecture as a set of architectural design decisions. In: 5th working IEEE/IFIP conference on software architecture (pp 109–120). IEEE
50. Kruchten P, Capilla R, Dueñas JC (2009) The decision view's role in software architecture practice. IEEE Softw 26(2):36–42
51. Ameller D, Franch X (2010) How do software architects consider non-functional requirements: a survey. In: International working conference on requirements engineering: foundation for software quality (pp 276–277). Springer, Berlin, Heidelberg
52. Harrison N, Avgeriou P (2007) Pattern-driven architectural partitioning: balancing functional and non-functional requirements. In: 2007 Second international conference on digital telecommunications (pp 21–21). IEEE
53. Myllymäki T, Koskimies K, Mikkonen T (2002) Structuring product-lines: a layered architectural style. In: International conference on object-oriented information systems (pp 482–487). Springer, Berlin, Heidelberg
54. Pahl C, Giesecke S, Hasselbring W (2009) Ontology-based modelling of architectural styles. Inf Softw Technol 51(12):1739–1749
55. Steinberg D, Budinsky F, Merks E, Paternostro M (2008) EMF: eclipse modeling framework. Pearson Education, Upper Saddle River
56. Gronback RC (2009) Eclipse modeling project: a domain-specific language (DSL) toolkit. Pearson Education, Boston
57. Arizona Center for Integrative Modeling and Simulation (2004) http://www.acims.arizona.edu/
58. Sarjoughian HS, Singh R (2004) Building simulation modeling environments using systems theory and software architecture principles. In: Proceedings of the advanced simulation technology conference (pp 99–104)
59. Zeigler BP, Sarjoughian HS (2003) Introduction to DEVS modeling and simulation with java: developing component-based simulation models. Arizona State University, Tucson
60. Blas MJ, Gonnet SM, Leone HP, Zeigler BP (2018) A conceptual framework to classify the extensions of DEVS formalism as variants and subclasses. In: Proceedings of the 2018 winter simulation conference (pp 560–571). IEEE Press
61. Blas M, Gonnet S, Leone H (2017) Modeling user temporal behaviors using hybrid simulation models. IEEE Latin America Trans 15(2):341–348