ElaClo: A Framework for Optimizing Software Application Topology in the Cloud Environment

Nikola Tanković[a], Tihana Galinac Grbac[b], Mario Žagar[c]

[a]*Department of Information and Communication Technologies, Juraj Dobrila University of Pula, Croatia*
[b]*Faculty of Engineering, University of Rijeka, Croatia*
[c]*Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia*

Abstract

Application architectures in the cloud employ elastic components, and achieve lower operating costs without sacrificing quality. Software architects strive to provide efficient services by deciding on software topology: a set of structural architectural decisions. For a given application, there can be numerous software topology alternatives creating the need for automated optimization methods. Current optimization approaches rely on experts providing application performance models built upfront, based on their experience and the requirements provided. While such techniques are effective and valuable, they require additional maintenance effort as the software evolves.

This paper introduces ElaClo, a framework for optimizing application topologies in a cloud environment. ElaClo's main contribution is in providing optimization in the software assembly phase from automatically extracted application models. ElaClo provides workload generation, monitoring, topology management, elasticity mechanisms, and algorithms to support the optimization process. We have implemented ElaClo as an expert tool and evaluated it on a real-life cloud application from the retailing business domain. ElaClo was used to select optimal topologies with regards to service response time objectives and infrastructure costs. The efficiency of the optimization process and the quality of optimization results were validated quantitatively on a set of optimization runs. Results demonstrate the effectiveness of the

Email addresses: nikola.tankovic@unipu.hr (Nikola Tanković), tihana.galinac@riteh.hr (Tihana Galinac Grbac), mario.zagar@fer.hr (Mario Žagar)

suggested framework in yielding optimal topologies.

Keywords: software architecture, cloud computing, service-oriented computing, application topology, service deployment, evolutionary optimization

1. Introduction

Software architectures encompass one or many software structures that affect the ways systems achieve functional and non-functional requirements (Clements et al., 2010). Such structures reveal software elements, relations among them, and properties of both. Designing and maintaining software architecture is a critical software engineering process, and as such, it should be based on informed decisions. Relying only on end-result performance measurements tends to hide some fallacies conducted throughout the development. Decision making in designing architectures is an important research focus revealing many common mistakes that software architects make (van Vliet & Tang, 2016).

One important aspect of the software architecture decision process is addressing quality attributes (Ameller et al., 2015), which are shown to be at least as important as functional requirements in service-based systems (Cardellini et al., 2009). In this paper, we are especially concerned with topology decisions that influence the performance of service-based end applications provisioned in the cloud infrastructure.

Reasoning on optimal software deployment structures for applications in cloud environments with dynamic resource provisioning is a non-trivial task with several concerns: it is not easy to prepare a sufficiently large testing workload that will expose bottlenecks and non-elastic behavior, nor to arrange relations among components providing end services, especially on how to arrange deployment of such components to a cloud infrastructure. One encounters many questions like: Should software components be consolidated and deployed as a composite, or separated into multiple components? How to evaluate different benefits across such decision space (*consolidation* vs. *separation*) regarding operational infrastructure costs and performance? What topology bests suits my specific workload? Is there an optimal solution and how sensitive is it to workload volatility? How to design efficient scaling policies based on these decisions? To answer these questions, software architects usually rely on simple tests like maximum load testing, trial and error

iterations, and mostly intuition by choosing simplest, but not necessarily the optimal solution.

There are methods for simulating cloud applications (Calheiros et al., 2011) based on simplified application models, but require additional inspection, knowledge and effort for every iteration of the development process. Most architecture optimization techniques support architects by advocating design-time performance models (Koziolek et al., 2011; Mostinckx et al., 2009), which are very useful for early insights on future system behavior, but are quite challenging to build. They demand an experienced architect who can predict future performance metrics such as average CPU cycles for many different tasks throughout the system. A systematic overview of existing performance evaluation methods for distributed systems is given by Akinnuwesi et al. (2012).

Once the initial component implementation is achieved, early performance tests often expose deviations from early design-time models that undermine the whole integrity of the optimization process. New techniques that are based on measurements of targeted quality criteria are required to support optimizations after initial implementations.

In previous work (Tanković et al., 2015b) we demonstrated that a naïve structural solution can be sub-optimal even for simple web applications with a few components, so we made further advancements and developed a framework for evaluating different service-based application topologies in a cloud environment backed with evolutionary optimization algorithms and simulation models.

The main contribution of this paper is the design and implementation of the *ElaClo* framework for optimizing topologies of service-based applications that use cloud resources in an elastic manner under desired performance targets. Using ElaClo, software architects are provided with an expert system that gives insight into different application topologies and their effect on overall quality and cost of applications. Topology decisions can be evaluated after initial implementation of software components with the ability to determine optimal cloud topologies with respect to performance and infrastructure costs. Since performance models are automatically extracted by ElaClo from artifacts, topology optimization can be repeated throughout the process of software evolution with no need to constantly update application performance models.

We believe these contributions mitigate the gap in existing expert systems literature, providing a framework for achieving tactical decisions in crafting

elastic application architectures. Existing runtime optimization tools concentrate only on a single elastic component or elasticity that is achieved strictly by deployment tiers. These design-time tools are not backed by measured component performance but rather employ performance models with limited support for accommodating elastic behavior.

The remainder of this paper is organized as follows. Section 2 provides theoretical background. Section 3 defines the topology model used throughout the paper and enumerates challenges involved with making topology decisions. Design and implementation of ElaClo are described in Section 4. Section 5 describes the evolutionary optimization process applied by ElaClo using simulation models. Section 6 describes validation goals and methodology, which is further applied to a case study on the *CashRegister* application. Section 7 elucidates related work, and Section 8 concludes the paper providing future work incentives.

2. Background

Service-Oriented Computing has gained a lot of traction in the past decade due to the many benefits of developing software as a set of decentralized and loosely-coupled services enabling flexible software systems (Wulf et al., 2008; Papazoglou & Georgakopoulos, 2003). *Software Oriented Architecture* or SOA, when referred to as a paradigm, encompasses a set of guiding principles for building Service-Based Applications (SBAs) (Nitto & Meilander, 2012). A more recent trend, in the form of Microservice architecture reinstates the same principles in a more developer-friendly way easing the rigorous SOA principles and patterns (Thönes, 2015). The underlying concept remains: software construction should be based on a set of independent software units, where each unit can be constructed, maintained, deployed, and evolved independently.

The question whether SBAs are much easier to maintain and evolve than holistic approaches is largely influenced by application size. To avoid overdesign, novel software applications start small as monolithic applications composed of loosely-coupled components and evolve their way into a SBAs by incremental extraction of components into standalone artifacts (Newman, 2015). Extraction decisions should be based on application size, structure, workload characteristics, and service reuse potential. Thus, we will regard application components as base building blocks that can then be further exposed through different web services.

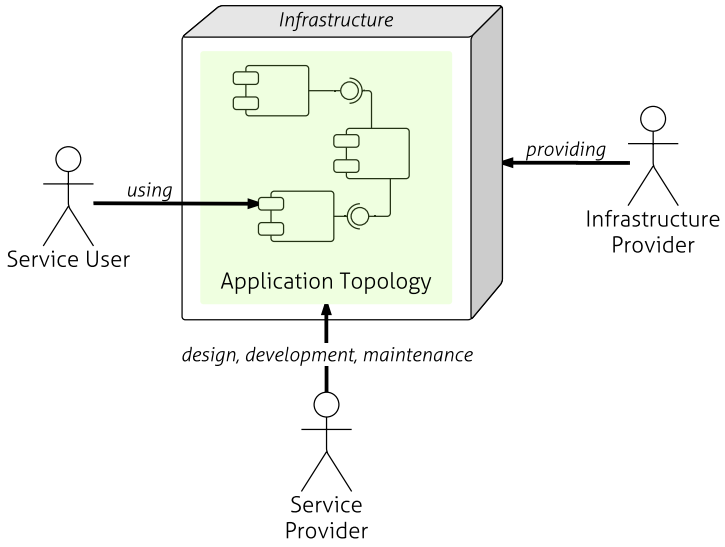


Figure 1: Concepts and roles involved around our research context

Cloud computing has emerged as a natural solution for on-demand resource provisioning, and is especially suited to provisioning SBAs with volatile workloads. Cloud and Service-Oriented Computing fit together well with each other (Dillon et al., 2010). Cloud platforms allow dynamic leasing of resources based on a *pay-as-you-go* model, a cost model that is desirable for service providers who exploit automatic control of resource consumption based on quality of service (QoS) attributes. Cloud infrastructure providers expose developer-friendly APIs for managing cloud resources, e.g. managing *Virtual Machine* (VM) lifecycles. However, crafting software applications amenable to such dynamic infrastructure capabilities is not a trivial task. The balance between infrastructure costs and service quality is hard to achieve, and involves additional concerns like specialized monitoring and scaling control algorithms.

This work will use terms *Service User*, *Service Provider* and *Infrastructure Provider* as displayed in Figure 1. The Service user is regarded as the consumer of the software service. Overall software service is represented as a composition of individual components arranged in some topology. It is designed, developed and maintained by service providers, and provisioned on infrastructure leased from infrastructure providers.

Cloud computing concepts are considered best utilized when service providers

tune their software applications to satisfy consumer's demands in an optimal cost-efficient way. Service providers guarantee performance levels to their consumers through Service-Level Agreements (SLAs). They employ various software controllers that dynamically lease infrastructure in order to maintain these levels (Wu et al., 2013), e.g. leasing additional resources in intensive workload periods.

There are also research techniques for integrating SLA goals into the software development phase (e.g. directly into the source code), enabling a much better control over application performance (Copil et al., 2013), which simplifies service-level performance specifications. But the main problem for service providers is still present in something we refer to as the *SLA sandwich problem*, where service providers must devise a way to design their software solutions so that performance levels are met. For that, success is heavily influenced by software architecture decisions. Cloud infrastructure providers typically offer SLAs in terms of infrastructure availability, or so-called *uptime*, while service providers are required to offer performance-related SLAs, often as response time percentiles (Xiong & Perros, 2009). These two measures are incomparable, e.g. better infrastructure availability will not lead to better application performance.

There is a large amount of research introducing mechanisms for optimal resource provisioning at infrastructure-, platform- and application-levels (Galante & De Bona, 2012). There is also a lot of research focused on studying runtime dynamics of an application such as the construction of *rule-* and *model-based* controllers for elasticity (Huang et al., 2014). However, there is little work present on how to structurally design software to be cost-effective in the cloud. In this paper, we emphasize the importance of *application topology*: a development and deployment aspect of software architecture concerning the structures of software components, together with their allocation of physical resources. Satisfying quality attributes through software architecture is recognized as important direction in a comprehensive study on software architectural decisions (Tofan et al., 2014). The term *application topology* was introduced in the TOSCA specification (Binz et al., 2014) referring to the structural and deployment view on software architecture by laying out components and corresponding deployment infrastructure. Since a given set of software components and infrastructure parts can yield numerous topologies, each of which affects costs and performance differently, we are interested in eliciting the most cost-effective one. This is especially important when the incoming workload is volatile where software elasticity

plays a crucial role (Tanković et al., 2015b).

3. Definitions and Challenges

In order to systematically explore different solutions from feasible topologies, we propose a *topology model* that combines application components into elastic services deployed on dynamic infrastructure. This model is represented by *Application Topology Graph* (ATG) based on the semantics of Unified Modeling Language (UML) Component and Deployment diagrams (OMG, 2015). In this section we describe ATG and present research challenges.

3.1. Application Topology Graph

Application topology is a structural description of applications components, their mutual relations, and mapping to infrastructure resources (Binz et al., 2014). In this work, we will focus on the computing resources in form of *virtual machines* (VMs) and software applications as composition of software components.

In previous work, we analyzed how different combinatorial solutions achieved by different mappings between components and resources (VMs) effect both application performance and total cost (Tanković et al., 2015b). The current efforts in choosing optimal deployment mappings are mostly based on trial and error or applying best practices (Newman, 2015).

ATG structure is used to represent all topological combinations of given components and their relationships. It is a scalable topology model that achieves elasticity by combining components into *elastic groups* (EGs), where each EG is a self-scalable compound component with its own scaling rules and mechanisms.

ATG is defined as a directed graph G_{ATG} which is composed of four subgraphs

$$G_{ATG} = G_{CS} \cup G_{EG} \cup G_{RT} \cup G_D$$

namely service structure graph G_{CS}, elastic grouping graph G_{EG}, resource type graph G_{RT} and deployment graph G_D.

The **component structure graph** is defined as

$$G_{CS} = (C, E_C)$$

where C represents a set of components, and $E_C = \{(c_i, c_j) | i \neq j, c_i, c_j \in C\}$ represents dependency relationships among them.

An **elastic grouping graph** is a connected bipartite graph connecting components to elastic groups:

$$G_{EG} = (C, P, E_{CP})$$

where each component from C belongs to some partition contained in P, a super-set of all possible *elastic groups* (EGs). Mapping from components in C to groups in P is represented by $E_{CP} = \{(c_i, p_j) | c_i \in S, p_j \in P, p_j = f_{C \to P}(c_i)\}$, where component-group mapping surjective function $f_{C \to P} : C \to P$ assigns each component to an elastic group so that no component can be in more than one group and there are no empty groups, that is $\forall p \in P, \exists c \in C, f_{C \to P}(c) = p$.

A **resource type graph** is a mapping that dedicates a certain resource type $w \in W$ to every elastic group:

$$G_{RT} = (P, W, E_{PW})$$

where each group from P is mapped to a resource type from W represented by $E_{PW} = \{(p_i, w_j) | p_i \in P, w_j \in W, w_j = f_{P \to W}(p_i)\}$. Typically, each type represents different resources that service providers offer through a different set of virtual machine characteristics (e.g. amount of CPUs or RAM). A resource type is dedicated to each elastic group, which will bind components to specific VM types in the deployment and elastic processes.

Finally, a **deployment graph** is also a bipartite graph connecting elastic groups to resources:

$$G_D = (P, R, E_{PR})$$

where each group from P is connected to a resource from R by deployment mapping $E_{PR} = \{(p_i, r_j) | p_i \in P, r_j \in R, p_i = f_{P \to R}(r_j)\}$. Again, $f_{P \to R}$ is a surjective function meaning that every reserved resource must be mapped to EG. An example of both grouping and deployment functions is schematically represented in Figure 2.

An example of ATG instance is given in Figure 3 displaying dependency relationships between *Invoice*, *Reporting* and *Resource* components together with mappings to elastic groups and deployment infrastructure. For better understanding, Figure 4 shows an example of the transformation of ATG to UML component and deployment models.

The task of the ElaClo tool will be to explore different mapping functions $f_{C \to P}$, $f_{P \to W}$, and $f_{P \to R}$ for optimizing quality criteria. The following

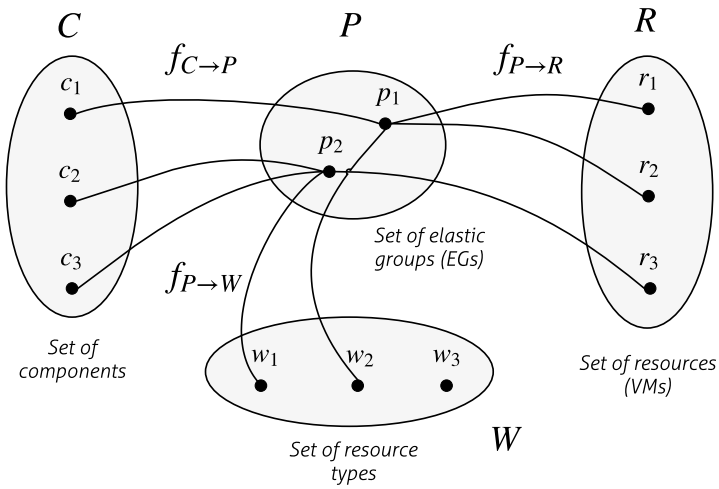


Figure 2: Schematic representation of mappings functions: $f_{P\to R}$ and $f_{C\to P}$

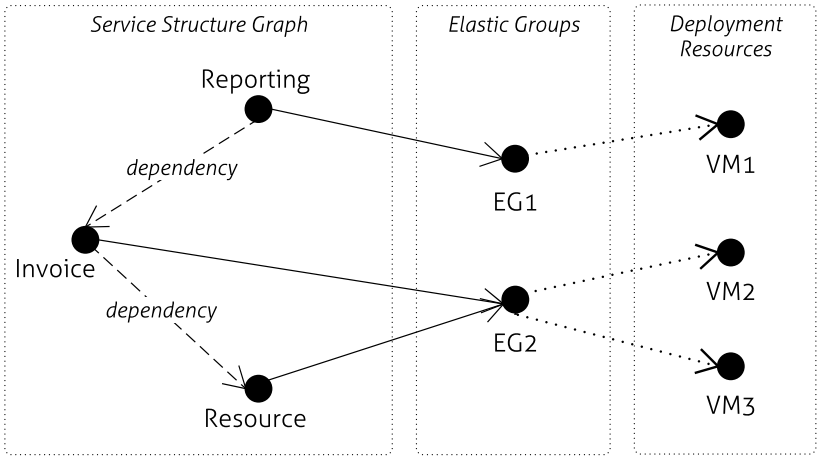


Figure 3: An example of Application Topology Graph (ATG)

subsections will lay out the optimization problem and disseminate the challenges involved in selecting appropriate topologies by explaining the effects of different topology related decisions to the performance of the application.

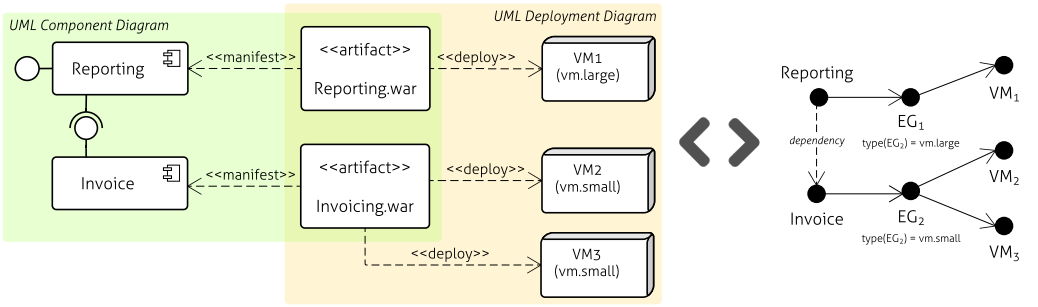


Figure 4: Application Topology Graph is similar to combined UML Component and Deployment diagrams

3.2. Optimization problem

The problem of selecting optimal topologies is a combinatorial one where the search space of all possible topologies $\mathbb{T}$ is defined by the available components C and available resource types W. The optimization goal is to find a set of optimal topologies $T \in \mathbb{T}$ defined by one or many objective functions $\Phi_q : \mathbb{T} \rightarrow V_q$ where V_q is a set of all possible quality measures for a given quality criteria q. Each chosen q must have associated set V_q that is totally ordered with an antisymmetric, transitive and total binary relation on V_q so that any two values in V_q can be compared.

The unique characteristic of this optimization problem is that it is depending on run-time optimization of the number of resources used (Figure 5): optimal allocation to elastic groups and resource types is dependent on run-time optimization of number of resources used which means new solving techniques must be developed which can take into consideration optimization behavior of application topologies in run-time, known as application elasticity.

The search space T represents all possible combinations of mapping functions $f_{C \rightarrow P}$ and $f_{P \rightarrow W}$. The size of T can be expressed using the *Stirling number of the second kind* (Sharp, 1968) $S(n, k)$, which gives the total count of combinations for allocating n elements into k sets (Equation 1).

$$S(n,k) = \frac{1}{k!} \sum_{j=0}^{k} (-1)^j \binom{k}{j} (k-j)^n \tag{1}$$

The possible elastic groups number for an application can vary from one (all components in single group) to $n = |C|$ (each component in an isolated

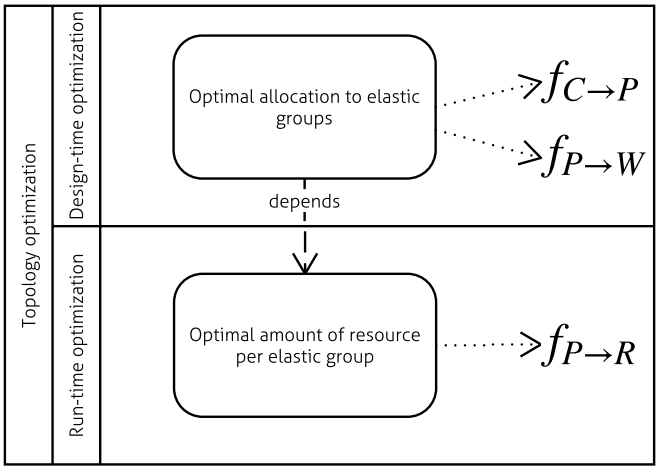


Figure 5: Dependency between $f^*_{C\to P}$, $f^*_{P\to W}$ and $f^*_{P\to R}$

elastic group), which gives the total number of possible topologies |T|:

$$\left|\mathrm{K}^{\mathrm{M}}\right| = \sum_{k=1}^{n} S(n,k) \cdot k^{|W|} \quad , \quad n = |C| \,. \tag{2}$$

.

Even for a small number of components, it is evident that search space grows quickly which makes sequential candidate evaluations unfeasible, and dictates the need for the application of search-based optimization methods (Bianchi et al., 2009).

3.3. Infrastructure cost model

ElaClo calculates total infrastructure costs based on a provided cost model. The most basic version provided by ElaClo is a cost model for leasing virtual machines. It involves different types of resources r in a form of virtual machines that can have distinct characteristics specified by virtual machine type $type(r) = w \in W$ where every type w has a corresponding cost $U_{COST}(w)$. Users are billed for virtual machine usage by predefined intervals of usage t_{cost}, which is typically a 1-hour interval. Total usage cost of a resource r during time t is expressed as

$$COST(r,t) = \lceil \frac{t}{t_{cost}} \rceil * U_{COST}(type(r)).$$

The provided cost model is used by almost every cloud infrastructure provider. There are also more advanced cost models available by some of the providers (e.g. auction-based models) but these are not included in current ElaClo version.

3.4. Topology effects on inter-component communication

Service-based architectures emphasize communication and collaboration that is often taking place on the network. One of key performance issues that software architect must consider is communication latency. A careful balance of local and remote communication among components is essential. Table 1 shows typical latencies involved in software engineering (Gregg, 2013). Latencies and overall communication cost penalties between remote components makes topology decisions in service-based applications very important.

For example, if we place *Reporting* and *Invoice* components from the *CashRegister* case on the same virtual machines we can gain a 1000-10000

Event	Duration
One 2GHz CPU cycle	$\sim 0.5\ ns$
Main memory access (DRAM from CPU)	$\sim 100\ ns$
Loopback TCP socket (same machine)	$\sim 10\ \mu s$
LAN	$\sim 1\ ms$
WAN (San Francisco to New York)	$\sim 40\ ms$
WAN (San Francisco to Australia)	$\sim 183\ ms$
Adding and booting a new VM	$\sim 2\ m$

Table 1: Example time scales of latencies.

fold reduction in round-trip latency between these two components by conducting communication within the same memory-space (e.g. method invocations) rather than through the invocation of remote communication protocols. Distributing applications across multiple services hinders end-to-end service latencies, so a careful balance on component distribution and consolidation decisions is required. It is obvious that topologies in which services are isolated have higher overall latencies than topologies that consolidate services on same resources. However, it is also preferred to isolate services from a maintenance and reliability point of view.

Challenge 1: How many services can we isolate without significantly affecting overall end-to-end latencies?

3.5. Topology effects on service-level objectives

There are different types of Service-Level Objectives (SLOs) in the literature (Wu & Buyya, 2010). In this work, we decided to design ElaClo around Service-Level Agreements composed of SLOs based on quite common quality attribute: service response time (latency). These types of SLOs are critical for real-time services targeting fast response times. We are especially interested in *percentiles of the response time* metric since average values do not address concerns of service users satisfaction properly (Xiong & Perros, 2009). Response time percentiles reveal statistical aspects of measured response times across usage. For instance, the service user can demand that 95% of all requests must be completed under a targeted response time. Under these assumptions, we see that the distribution of the response times becomes a very important factor. In this chapter, we provide a theoretical argument

for different impacts that topology choice can have on service response time.

If we represent service response times with a random variable T_S, and the distribution of T_S with f_{T_S}, then we can express required response time percentile $\tau_{\%}$ and targeted response time t_{max}:

$$viol\% = \int_{t_{max}}^{\infty} f_{T_S}(t)\mathrm{d}t \cdot 100\% \tag{3}$$

$$viol\% \leq \tau_{\%} \tag{4}$$

This imposes a preference on the distributions f_{T_S} that are characterized by smaller right tails. It can easily be seen that long tailed distributions with higher variability will require a larger upper bound t_{max}, and will thus have weaker SLAs. More specifically, by applying theoretical discoveries from queuing theory, a measure called *coefficient of variation* (C_S^2) is very important to explain delays in queuing systems (Harchol-Balter, 2013).

$$C_S^2 = \frac{Var(t)}{\mathbb{E}[t]^2} \tag{5}$$

C_S^2 is directly correlated with the variance of the response time variable. Therefore, a more stable service in terms of response time will achieve better response time percentiles. If we analyze services as queuing systems models, there are two factors causing variability in response time: (1) distribution of the inter-arrival times of incoming requests, and (2) the distribution of service times of the system. Since we can hardly effect the former because it is often based on many software usage scenarios, ElaClo will concentrate on latter by inspecting different topologies and their effects on response time distribution in the form of SLA percentiles violations. Based on that, we formulate our next challenge.

Challenge 2: By choosing topologies where highly variable services are isolated, we can lower the overall service coefficient of variation (C_S^2), and thus reduce the SLA violation rate. The challenge again remains in the careful choice over which services to isolate.

3.6. Practical challenges

Exploring different topologies by manual adaptation, packaging, deployment, and configuration of services is a tedious task (Durán & Salaün, 2015). Each topology additionally requires setting elastic capabilities and policies

in the form of different monitoring processes, controllers, and load balancers for each service.

In order to conduct a proper performance test, there is also an issue involving the choice and configuration of a load testing request generator that will realistically replicate usage scenarios. In previous studies we analyzed incoming workloads in business cloud applications (Tanković et al., 2015a) and derived a set of requirements for a synthetic workload generator:

- specifying a *workload mix*: a set of incoming service calls representing a combination of different user requests simultaneously;
- *variable intensity* for elasticity testing: at least a linear model for specifying the change in intensity measured in number of requests per time unit should be supported;
- a stochastic process of incoming requests arrivals: a Poisson process or similar model should be used to simulate the arrival of incoming requests.

Software services with only a few components can yield numerous topologies that require a systematic and time-consuming process for evaluating each one.

Challenge 3: How to alternate between different topologies in a fast and seamless way in order to evaluate different topologies efficiently? How to test each topology with a realistic workload?

4. The ElaClo Framework

In order to address the challenges presented in Section 3, we propose ElaClo: a cloud-based framework for optimizing application topologies based on service response times and operating cloud infrastructure costs. This section will elaborate on the usage of ElaClo, together with explanations of the details of ElaClo structure and implementation.

We propose ElaClo as a distributed system divided into:

- *ElaClo Application* components in form of an application framework for source code annotation and monitoring presented in our previous work (Tanković et al., 2015b).
- *ElaClo Development Environment* components represented by a web application for evaluating the performance of different topologies.

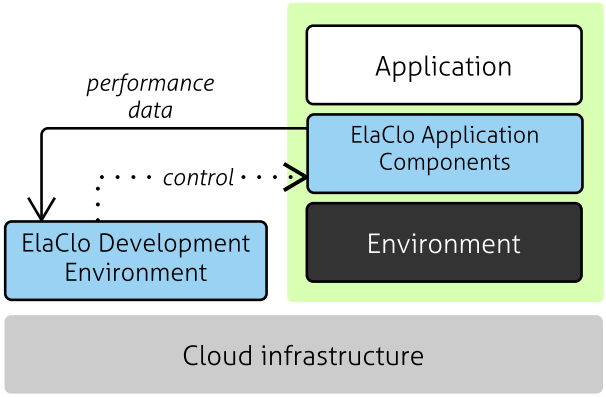


Figure 6: ElaClo framework overview

Figure 6 shows a high-level overview of the ElaClo framework. The topology for each test scenario is being configured in run-time by the ElaClo Development Environment components controlling the ElaClo Application components. Application components record the throughput and latencies of each application instance in order to deduce performance characteristics of each topology. We will provide more details on all the ElaClo components in Section 4.3.

4.1. Usage Methodology

Figure 7 shows a possible workflow when applying ElaClo in iterative software development. In order to conduct the test of different topologies, the developer must specify the workload for the tests consisting of a workload mix: multiple service requests (operations) $o \in O$ and intensity function $\lambda(t)$. Each service request consists of the URL, HTTP headers, and request payload. Since there can be many service request types in a defined workload, a probability of occurrence for each operation is also required with the constraint that $\sum_o p(o) = 1$. More details on modeling and generating workload is provided in Section 4.3.1.

Alongside the workload model, users must also provide accompanying response time SLOs for each service operation $t_{max}^{(o)}, \forall o \in O$ and required response time percentile $\tau_{\%}$.

Usually, there are many topologies to evaluate for a given application. Sequential evaluation of each topology in the cloud is time and resource demanding and often not feasible. In order to propose promising candidates

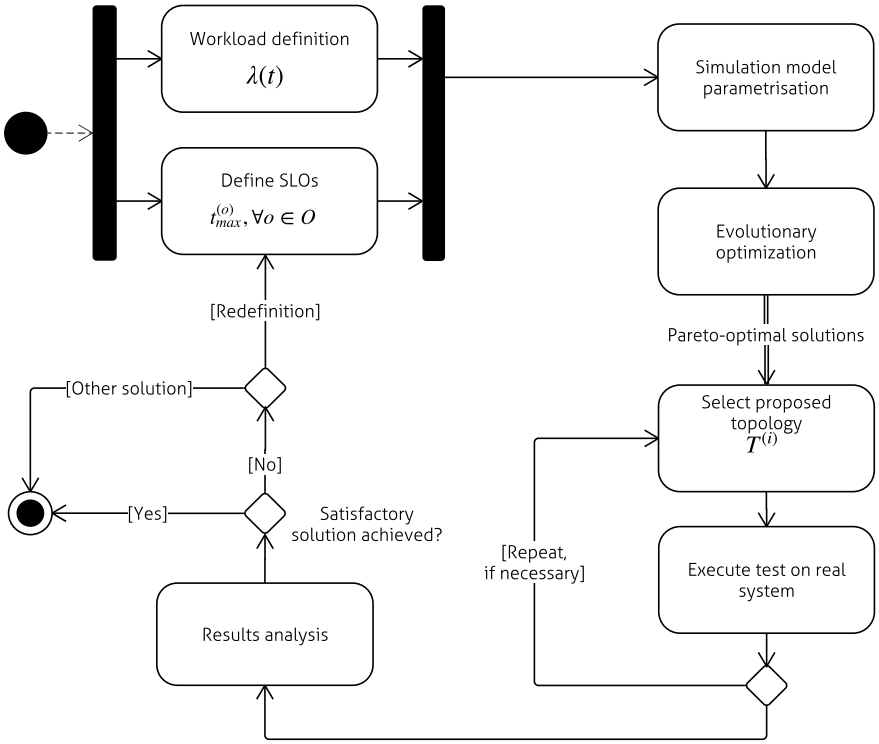


Figure 7: Overview on process of using ElaClo; variations are possible according to specific software development process

for evaluation in advance, ElaClo extracts a performance model from the application automatically that is used in the optimization process conducted in the simulation environment. This results in reducing the number of promising topologies that can be re-evaluated on cloud infrastructure. This process is described separately in Section 5.

If required, a custom topology can also be evaluated. For that, the application developer must provide the $f_{C\to P}$, $f_{P\to W}$ and $f_{P\to R}$ mapping functions, defined in Section 3. In ElaClo tool implementation we represented $f_{C\to P}$ and $f_{P\to W}$ mappings with a *drag-n-drop* user-interface for arranging available components into elastic groups. Figure 8 explains the user interaction elements provided by the topology section of the ElaClo user-interface. For convenience, a list of available topologies is also displayed. An example of the available topologies for the *CashRegister* case study application is displayed in Figure 9.

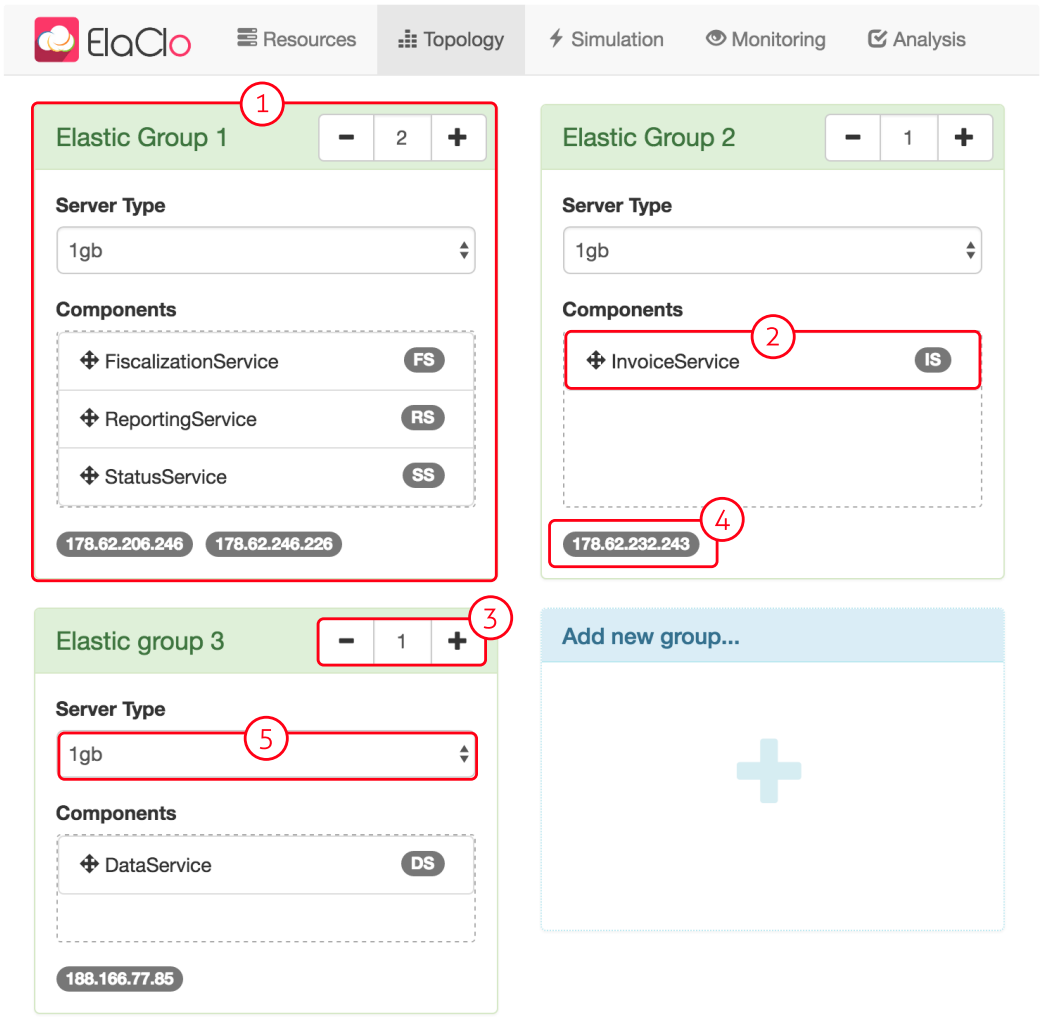


Figure 8: ElaClo user interface to define a custom topology based on our proposed topology model. Annotations: (1) Elastic groups that contain multiple components, (2) Every component in an elastic group can be dragged into another group, (3) The initial number of VMs assigned, (4) Public IP addresses of assigned VMs with deployed components, (5) Selection of VM type per group

For each topology $T^{(i)}$ there exists a corresponding $f^{(i)}_{C \to P}$ mapping function defined by application developers. The deployment mapping function f_{DPL} can be derived automatically by ElaClo, by assigning one available VM to each elastic group. Application developers can alter these starting assumptions by specifying the number of VM resources for each group. Note

that these are only the starting allocations, since there exists elasticity control that can alter the number of resources in run-time according to test workload needs, which makes f_{DPL} mapping dynamic.

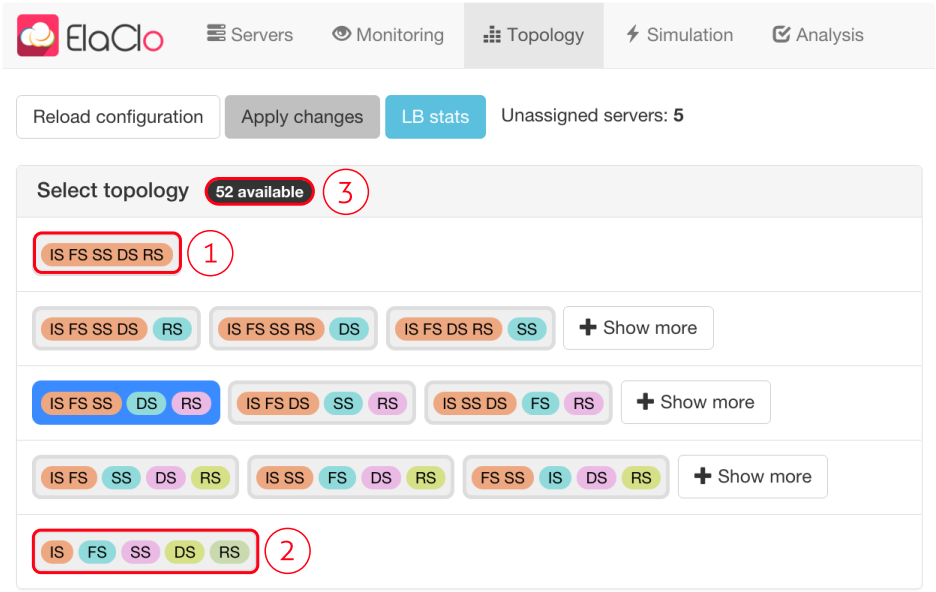


Figure 9: ElaClo interface for selecting topologies. Topologies are separated according to the number of elastic groups: (1) Topology with one group, (2) Topology with a maximal number of groups. The number of available topologies is also displayed (3).

Once the topology has been selected, and workload and SLA both defined, evaluation can start. ElaClo generates the desired workload and collects results from each topology evaluation run. Finally, a comparison among each topology performances is provided. We cover this in section 4.2.

4.2. Results analysis and SLO tuning

Upon evaluating candidate topologies, the following metrics are available for each candidate:

- Average throughput and response time for services provided by all components,
- SLA violations expressed as percentage of violations p_v. Ideally, p_v should be within limits defined in SLA, that is $p_v \leq (1 - \tau_{\%})$ where $\tau_{\%}$ is the required response time percentile defined in Section 3.5,

- Total infrastructure costs according to the cost model from Section 3.3.

On the *Analysis* part of the user interface, ElaClo displays and compares all topology results, flagging topologies that do not satisfy defined SLA, as displayed in Figure 10. If there are no topologies that completely satisfy SLA, it is possible that SLA is to strict or the application has a performance problem. The developers can observe performance metrics for each application component. If SLAs are being met by more than one topology, application developers can rank them depending on their preferences: cost-savings vs. improved performance leading to stricter SLAs. ElaClo can rank topologies according to these criteria.

If the resulting chosen topology $\mathrm{T}^{(final)}$ has some elastic groups that contain more than one component, the application designer can decide to create a composite and provide them through single web service. Ideally, the developer combines all services in every group $P^{(i)}$ having $|P^{(i)}| > 1$, where

$$P^{(i)} = \{s | (s, g_i) \in E_{SP}^{(final)}\}$$

and $E_{SP}^{(final)}$ are edges of subgraph G_{EG} contained in $\mathrm{T}^{(final)}$ (as defined in Section 2). For example, Figure 11 displays how *Invoice* and *Fiscalization* components from the *CashRegister* application can be combined into a single service and thus collocated at same hardware resources. This process in not automatized or conducted with ElaClo.

4.3. ElaClo Framework components

Figure 12 provides a detailed component level view of the ElaClo framework. We will describe each ElaClo component separately.

4.3.1. Workload Generator

The workload generator has been custom built to generate workload specified by application developers. ElaClo enables modeling workloads as a non-homogeneous Poisson Processes (NHPP) with intensity function $\lambda(t)$ defined as a piecewise linear function. We chose a Poisson process to achieve randomness of incoming arrivals to reveal potential congestion problems, and applying a workload model that provides changes in intensity will trigger elastic behavior through *scale-out* and *scale-in* actions.

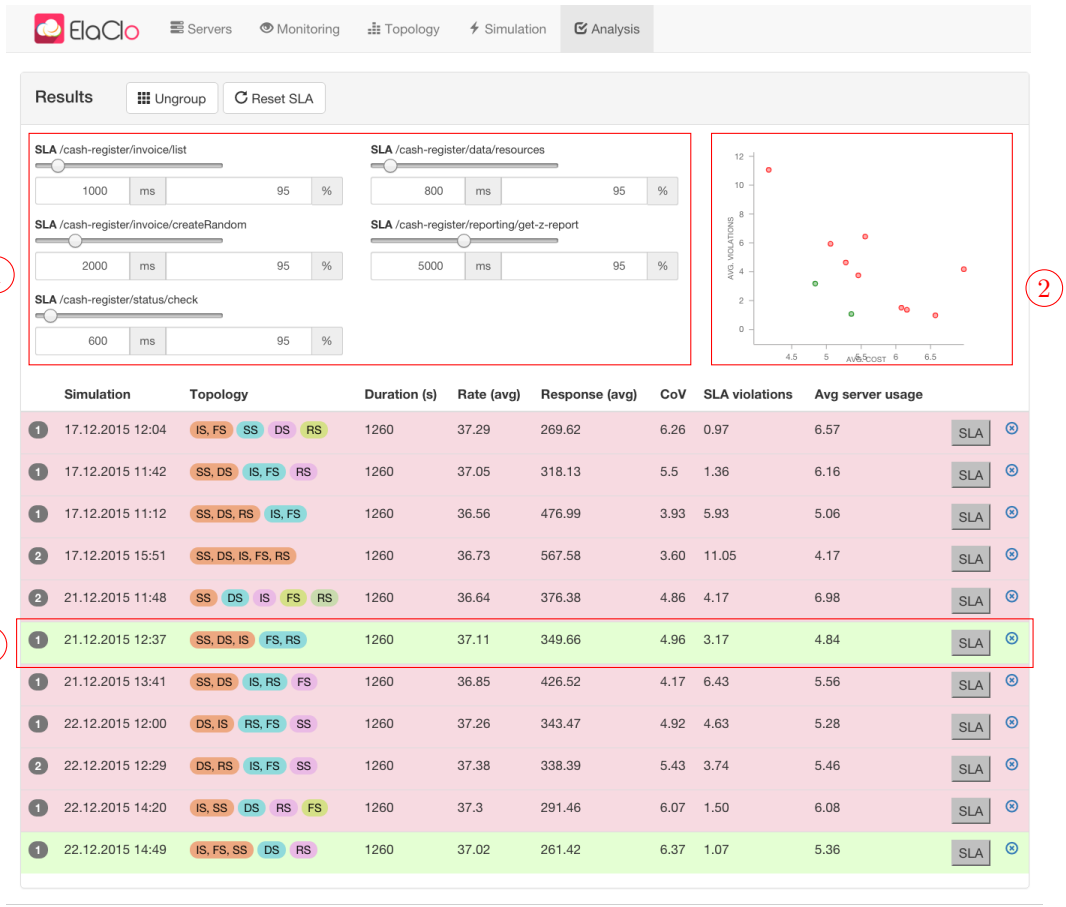


Figure 10: ElaClo interface for analyzing results. Annotations: (1) Tuning SLA parameters, (2) Graphical comparison of topologies (cost-quality), and (3) Topology results where topologies satisfying SLA are marked green.

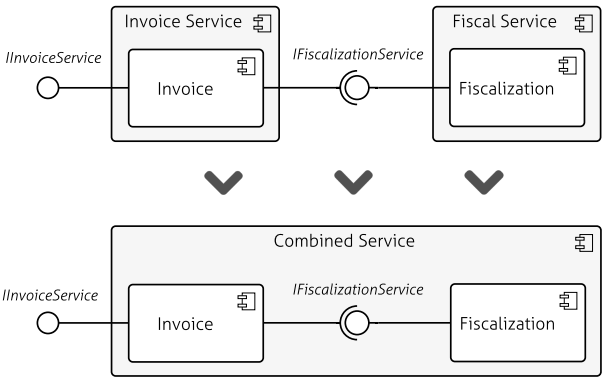


Figure 11: One possible decision resulting from ElaClo tests - to combine two services for better overall QoS

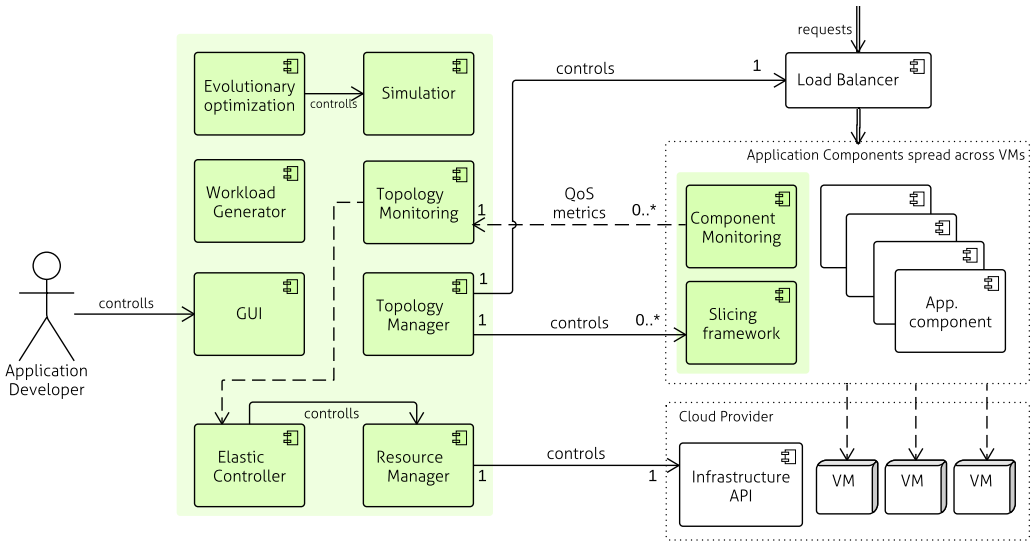


Figure 12: ElaClo framework components

For example, workload in Figure 13 can be expressed as:

$$\lambda(t) = \begin{cases} 5 & \text{if } t \leq 5 \\ t & \text{if } 5 < x < 20 \\ -2t + 60 & \text{if } 20 \leq x < 50 \end{cases} \tag{6}$$

The first part of the given example workload function from equation 6 serves as a *warm-up* period where we leave $\lambda(t)$ at a steady low-value; the

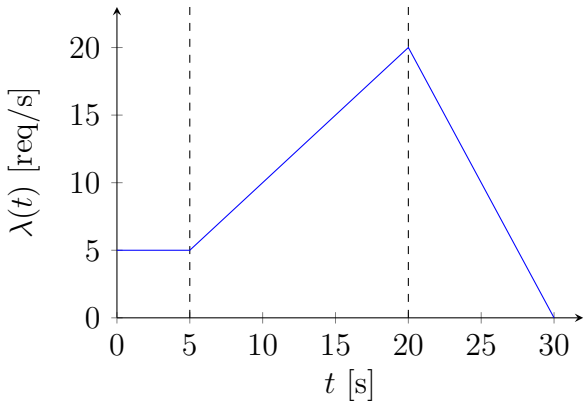


Figure 13: Example $\lambda(t)$ function for NHPP workload

second part corresponds to increasing workload, and final part to decreasing workload.

In order to specify a workload mix - the ratio between different request types, developers can provide probabilities of occurrence for each use case. For that, some insight is required in the characteristic workloads of the tested application or service.

4.3.2. Component Monitoring

The ElaClo framework applied in application components provides monitoring mechanism for several performance metrics:

- Response time and throughput for every operation on each component;
- System load average of allocated VM;
- Current CPU usage levels of allocated VM.

All metrics are calculated using a *moving average* method with a fixed configurable time window. This method is commonly used to reduce high oscillations in monitored data. Calculated values are continually serialized and sent to central *Topology Monitoring* component through a dedicated internal web service. The main metrics collected are the response time and throughput of each component operation, giving insight to the overall topology performance. CPU usage levels and system load averages are collected across VMs in application topology and used to support scaling decisions.

4.3.3. Topology Monitoring

Topology monitoring is responsible for collecting monitoring data from all application components and their underlying infrastructure resources (virtual machines). It aggregates collected metrics for components spread across multiple nodes. For example, if a component is deployed on multiple virtual machines, its average response time is calculated. This monitoring mechanism implements topological monitoring techniques presented in previous research (Moldovan et al., 2013) and (Trihinas et al., 2014). Once aggregated, topology monitoring data is available to be used by the *elastic controller* mechanism in supporting scaling decisions.

4.3.4. Elastic Controller

In order to test elasticity behavior and characteristics of each topology, ElaClo needs to provide elasticity attributes for tested services: means to elastically scale according to the current workload. For that purpose, ElaClo has a rule-based elastic controller that uses data from the *Topology Monitoring* module continually calculating moving load averages for all virtual machines assigned to application components. It then triggers scaling actions based on rules for these values, e.g. if CPU load reaches 70%. ElaClo enables configuration of several parameters:

- The time window for calculating moving average values.
- Thresholds for scaling *in* and *out*.
- Cool-off periods after scaling actions (separate for scaling *in* and *out*) to reduce scaling oscillations.

ElaClo provides defaults for these parameters. If necessary, they can be adequately set to correspond to the amount of time needed for a new VM to be fully operational. Additionally, if there is a need to optimize topologies with a more advanced elasticity controller policy, ElaClo supports turning off the provided controller, which leaves it up to the user to integrate a more sophisticated mechanism.

4.3.5. Topology Manager

Topology Manager (TM) is responsible for arranging the topology defined by the ATG graph model using provided application artifacts. The Topology Manager actuates mapping functions $f_{C\to P}$, $f_{P\to W}$ defined in Section 2

and deployment function $f_{P \to R}$ that is dynamically changed during topology evaluations due to run-time elasticity.

ElaClo achieves the effect of different topologies by deploying all components to each VM used, and then directing the usage of each component according to currently applied topology by configuring the load balancer and instructing each component via *Slicing Framework*. This way, there is no need for additional redeployment actions during topology changes, but rather, just a reconfiguration of each component, which results in a significant increase in speed.

4.3.6. Slicing Framework

The *Slicing Framework* together with the *Service Monitoring* module are the only two components integrated and used within the application that need to be optimized. They are controlled from a central *Topology Manager* module in configuring different application topologies. The *Topology Manager* controls which components should be available for each application instance deployed to a specified VM type. The *Slicing Framework* coordinates communication between components according to topology specifications. If certain operations from an application component invoke other operations that are not available locally, *Slicing framework* routes the request for this operation remotely through a load balancer which further routes the call to a remote component. A more detailed description on how topologies are managed and deployed is available in our previous work (Tanković et al., 2015b).

4.3.7. Resource Manager

The *Resource manager* is the actuator of decisions made in the *elastic controller*. It is responsible for transforming scaling intents to infrastructure-level actions using the infrastructure API provided by the cloud provider. This also includes reconfiguration of the load balancer and updating the internal ATG graph structure in the *Topology Manager* to represent modifications introduced by scaling *in* or *out*. Implementing elasticity is well researched so we will not cover this behavior here (Vaquero et al., 2011; Han et al., 2011).

4.3.8. Evolutionary Optimization and Simulator

These two components are responsible for extracting the simulation model using the application topology definition (ATG) and measuring the perfor-

mance (response times of each provided operation $o \in O$)) of the provided application components on a basic low load. The extracted simulation model is used as a cost function in the process of evolutionary optimization. Details on how evolutionary optimization is conducted are in Section 5.

5. Evolutionary optimization

Even a few application components can yield numerous topology variations. Given that a typical testing workload can be several hours long in order to properly test elasticity, evaluating each of the different topologies on a real infrastructure can be expensive and time-consuming even with ElaClo automation. To achieve topology optimization in an acceptable amount of time, ElaClo uses a surrogate simulation model based on queue networks from queue theory. Quality criteria such as cost and service response times are then determined much faster by running a discrete-time stochastic simulation upon a corresponding queue network for each topology variant. Searching for optimal candidates using simulation is achieved with an evolutionary algorithm.

Section 5.1 describes the surrogate model and simulation process that is used by ElaClo to enable faster candidate evaluation. Section 5.2 then elucidates the chosen optimization algorithm for searching the solution space with the described simulation as the cost function.

5.1. Topology evaluation

The application whose topology needs to optimized must integrate ElaClo Application components (as described in Section 4 and Figure 6) so that ElaClo can extract all the details on dependencies between application components together with simulation model parameters such as service-time distributions. This information is used to build surrogate models based on queue networks (QN) models.

Figure 14 displays a simulation meta-model consisting of a workload model (described in Section 4.3.1), a component model and an ATG model (from Section 3.1). Succinctly, the simulation model defines a variable intensity workload arranged from a mix of operation invocations from available components that are arranged in a certain topology. Service-time distribution for each operation is collected by ElaClo prior to simulation using real components and infrastructure. Measured response times are gathered by applying a minimal workload intensity.

The QN model used in ElaClo is a *multi-class open queue network* (MC-OQN) (Lazowska et al., 1984). Every MC-OQN node represents one or many application components deployed on a virtual machine. According to Kendal notation (Gross et al., 2008), nodes are modeled as $G/G/k$ with general arrival- and service-times distributions and k service nodes.

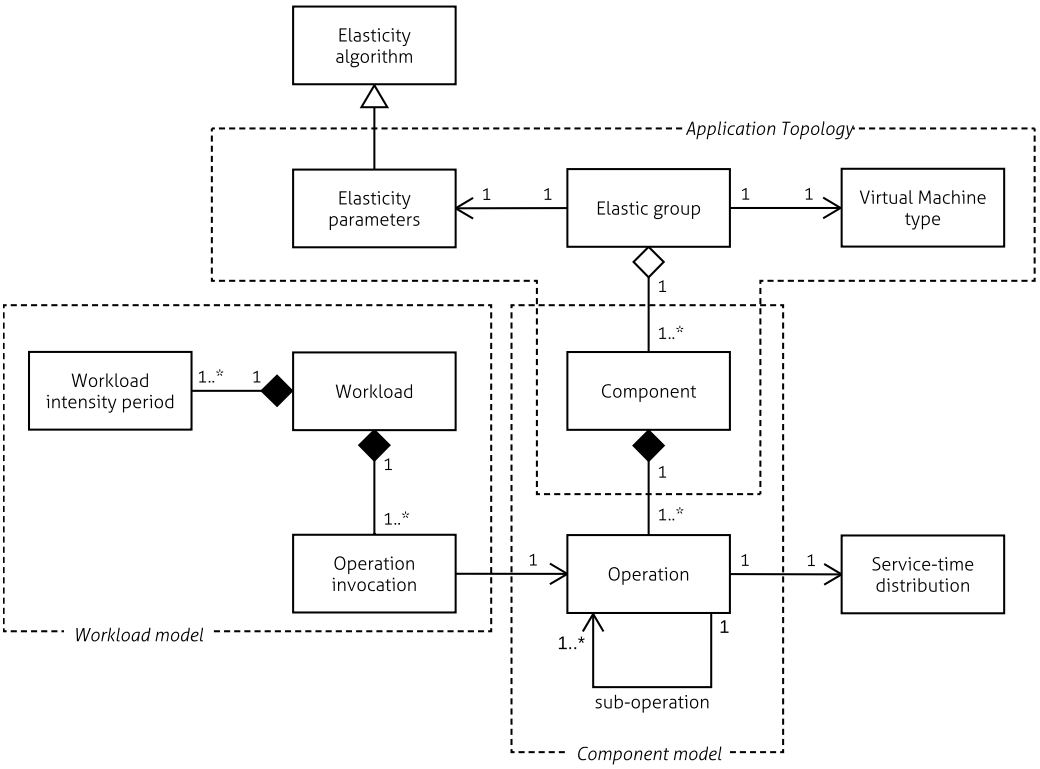


Figure 14: Simulation meta-model

The overall simulation process for determining performance of a single topology is presented in Figure 15. Aside from the aforementioned component model, the workload model, topology and service-time distributions, simulator additionally requires the cost model (from Section 3.3) and SLO parameters (from Section 3.4). Given all that, it can predict the infrastructure costs and amount of SLA violations. By default, the elasticity algorithm used is the rule-based algorithm (Section 4.3.4) with upper- and lower-bounds of node availability.

The simulation process yields several resulting metrics:

- total infrastructure cost for the given workload and topology,
- service-time distributions for all operations,
- a distribution of queue waiting times.

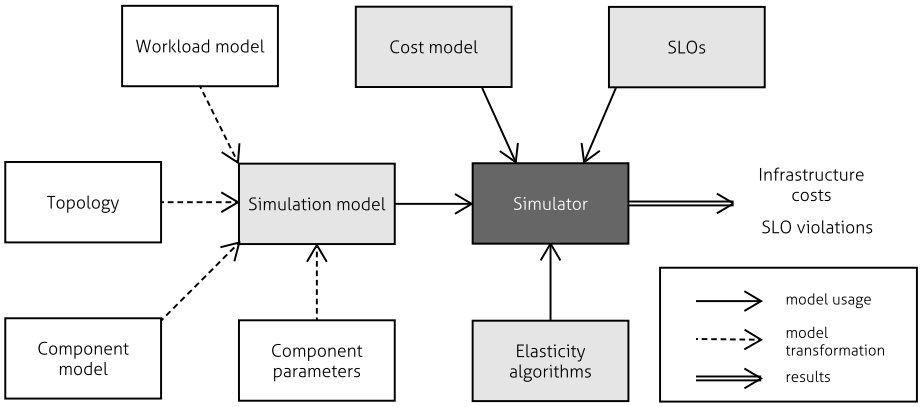


Figure 15: Complete process of simulating topology quality criteria

Provided data is used to calculate quality criteria in the form of response time percentiles and infrastructure costs required for the objective function in the optimization process.

5.1.1. Generating QN models

We will illustrate the transformation process from an application topology to a QN model on an illustrative example: an application M with two components $C^M = \{A, B\}$. Operations that component interfaces support are: $oper(A) = \{o_{A,1}, o_{A,2}\}$, $oper(B) = \{o_{B,1}, o_{B,2}\}$. A workload for M is defined with Equation 7 intensity function and request types according to Table 2 where $\tau\%$ represents targeted response time percentile that should be t_{SLA} or lower. M is to be deployed on a cloud infrastructure whose provider offers two virtual machine types $VM = \{VM_1, VM_2\}$ where vm_1 possesses one, and vm_2 two CPUs. Cost model follows the one presented in 3.3 with $cost(vm_1) = 6$ \$/h, and $cost(vm_2) = 12$ \$/h.

$$\lambda(t_{sim}) = \begin{cases} 5 & \text{if } t_{sim} \leq 5 \\ t_{sim} & \text{if } 5 < t_{sim} < 20 \\ -2t_{sim} + 60 & \text{if } 20 \leq t_{sim} < 50 \end{cases} \tag{7}$$

Every request type ($R = \{r_1, r_2, r_3\}$) is modeled as a single MC-OQN class. Workload is represented with a intensity vector

$$\vec{\lambda}(t_{sim}) = \lambda(t_{sim}) \cdot [p_1, p_2, p_3]^\top$$

Request type	Component	Operation	Dependency Ratio p_i (%)	t_{SLA}(ms)	$\tau_{\%}$
r_1	A	$o_{A,1}$	-	30	500	0.95
r_2	B	$o_{B,1}$	-	30	500	0.95
r_3	A	$o_{A,2}$	$o_{B,2}$	40	2000	0.95

Table 2: Workload for application in illustrative example

with p_1, p_2, p_3 being request ratios from Table 2. $\vec{\lambda}(t_{sim})$ defines incoming request intensity for every moment of simulation time (t_{sim}) for each request type. Request types are associated with a singe operation on the corresponding component that provides it, with a possibility that some operations depend on other operations.

Every node in MC-OQN represents the application of components deployed on virtual machines. The corresponding service-time (marked with random variable D) of each node is defined for all request classes separately:

$$D_{vm,r_i} = \sum_{o_{c,local}} D_{o_c,local} + \sum_{o_{c,remote}} D_{comm} \tag{8}$$

where $o_{c,local}$ represents all operations in which r_j is invoked on local components (deployed to current vm), and $o_{c,remote}$ are operations from components residing on different vms. If the dependent operation is located on another vm, service-time is increased for additional cost of network communication latency, represented with a random variable D_{comm}.

Resulting response time for a single request is determined by summing all participating service-times of nodes that service that particular request type:

$$D_{r_i} = \sum_{vm_j} D_{vm_j,r_i}. \tag{9}$$

Components from our example $C^M = \{A, B\}$ can be mapped to elastic groups in two ways. They can be in a single elastic group together, or separated into two elastic groups. Figure 16 displays the former topology where all request classes are serviced by single node (since it contains both components), whereas Figure 17a shows the latter variant. According to Table 2, request class r_1 with incoming intensity λ_1 targets operation $o_{A,1}$ on component A, thus it is associated with node $s_{1,1}$ from the first elastic group.

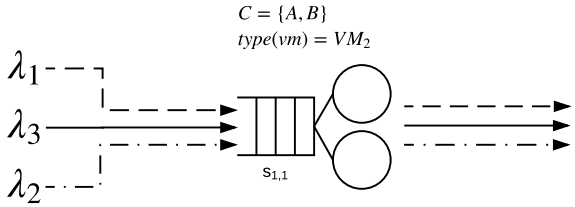


Figure 16: An example of QN with single elastic group whose virtual machine has two service nodes

Class r_2 with λ_2 intensity is targeting $o_{B,1}$ from component B. Class r_3 with λ_3 is linked to $o_{A,2}$ on component A which additionally invokes operation $o_{B,2}$ residing on component B.

According to the application topology graph from Section 3.1, the number of nodes in every elastic group is defined at run-time using elasticity mechanisms. This behavior is also implemented in the simulation process. During the simulation, the saturation of each node is determined as

$$\rho_{vm} = \sum_{i=1}^{R} \frac{\lambda_i}{\mu_i}$$

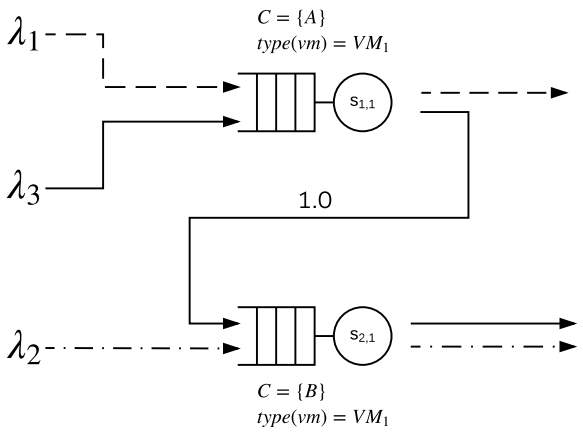
where $\mu_i \sim D_{vm,r_i}$ from Equation 8. ρ_{vm} is determined from every request class that is serviced on current node.

Elasticity controller applied is a rule-based scaling controller that induces scaling actions based on reaching lower and upper ρ_{vm} thresholds. Figure 17b shows an example where a second allocation group exhibits scaling to additional vm. It can be observed that load-balancer behavior is also simulated as incoming traffic is now equally distributed between two nodes.

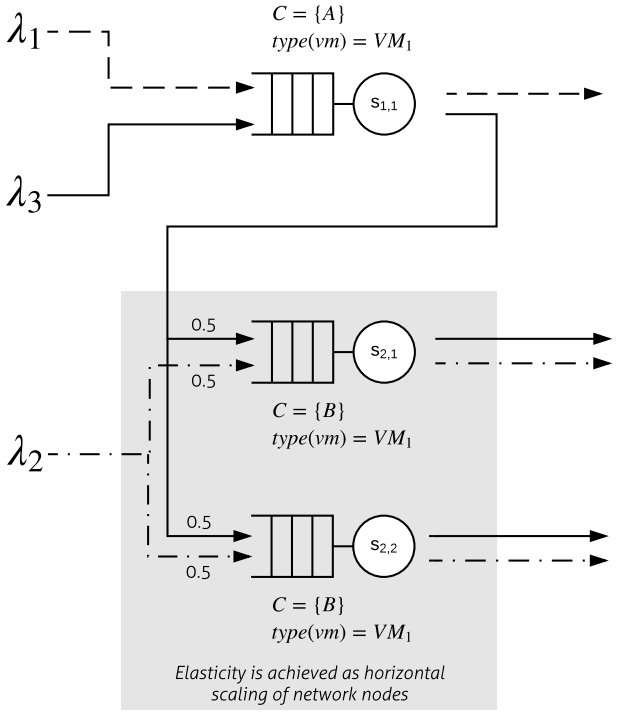
Virtual machine types ($VM = \{VM_1, VM_2\}$) are modeled with different k parameter that represent the number of service nodes on each network node. On Figure 17b node $s_{1,1}$ represents a vm of type VM_2 with two processors so $k = 2$ for that node.

5.2. Optimization algorithm

Topology optimization in this work is a combinatorial optimization problem (Korte & Vygen, 2012) with an exponentially growing search space, which does not necessarily induce NP-hardness Mann (2017). The presented topology optimization is a specialization of the deployment optimization problem present in the literature (Koziolek, 2011; Franceschelli et al., 2013)



(a) A QN representing topology with two elastic groups



(b) An example where a second elastic group scales horizontally to an additional vm instance, thus two nodes are used

Figure 17: MC-OQN that models possible topologies from application M

with a distinctive notion on components forming elastic groups and applying an evaluation function where such behavior is simulated. Given that it is an extension of a deployment optimization problem and that there are no closed-form solutions for its objective functions: assessing SLA violations and infrastructure costs, it is also considered to be NP-hard Canfora et al. (2005); Frey et al. (2013).

Additionally, it is also a multi-objective optimization problem since there is more than one quality criteria to be optimized (Donoso & Fabregat, 2016). Optimization goals in multi-objective optimization are often in mutual conflict, meaning there is no single solution that maximizes all objective functions. This is where the concept of Pareto optimality (Serafini, 2012) represents a set of solutions with optimal trade-offs among several objectives.

Assessing SLA violation rates and infrastructure costs from the given model in 3.3 has no known analytical solutions so we cannot formulate our problem as an integer programming model or similar exact technique. Instead, a black-box optimization approach is required, since our objective function does not provide gradient information. The best so far known approach is to apply population-based metaheuristics Feliot et al. (2017) such as genetic algorithms, with most popular proponents being NSGA-II and SPEA2 Eiben & Smith (2003); Zitzler et al. (2001).

Figure 18 shows optimization steps used in genetic algorithms using a (μ+ λ) approach (Back et al., 2000) with NSGA-II selection procedure (Deb et al., 2002). The ($\mu + \lambda$) approach generates λ new candidates in every iteration (generation) from he existing μ sized population. In doing so, mutation and crossing of candidates is used. As we continue, we will describe how each topology is represented as candidate in population, as well as the process of crossing and mutation operators applied.

5.2.1. Candidate representation

Every optimization candidate represents a single topology variant of our targeted system. It represents component mapping to elastic groups with the associated virtual machine types per group. An example candidate is given in Figure 19 with five components arranged into three elastic groups with every group having different virtual machine type.

5.2.2. Candidate operations

Genetic algorithm includes two operations: candidate mutation and crossing (Back et al., 2000). Both operations are carried out randomly with a

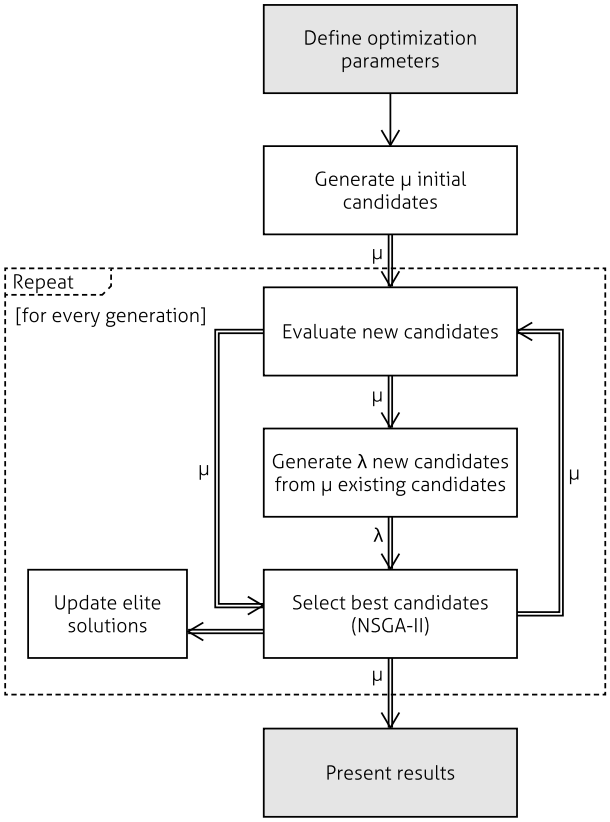


Figure 18: $\mu + \lambda$ genetic algorithm with NSGA-II selection method

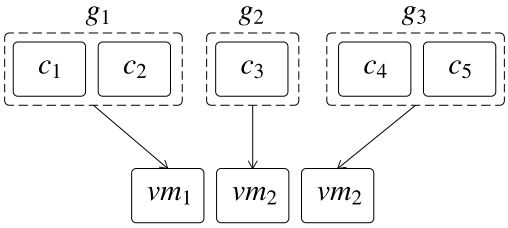


Figure 19: Example candidate with five components arranged into three elastic groups. First group uses a vm_1 virtual machine type, and other two groups use a vm_2 virtual machine type.

given probability, with the mutation probability set to $p_{mut} = 0.2$ and the reproduction probability set to $p_{rep} = 0.7$. The remaining probability of $p_{rand} = 0.1$ denotes randomly generating a new candidate. Crossing opera-

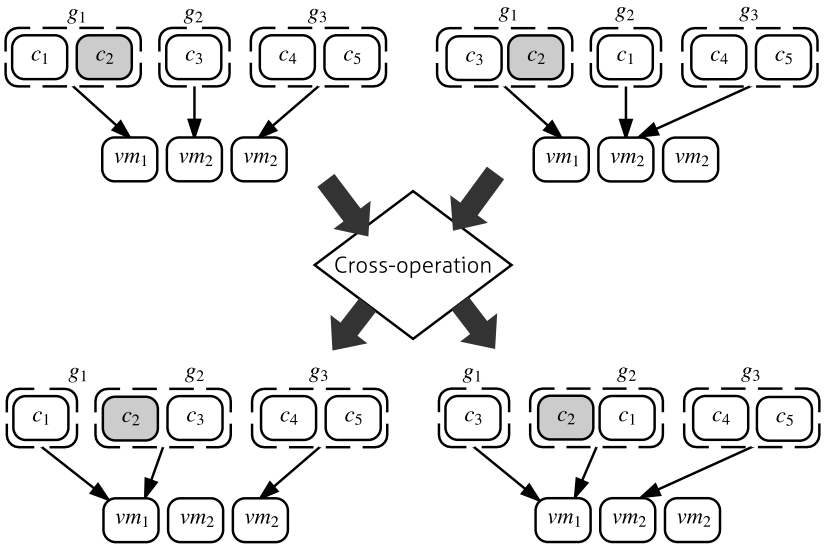


Figure 20: An example of candidate reproduction

tion is carried out on two parent candidates yielding two child candidates. In ElaClo, crossing is implemented by randomly choosing a component from the first parent and moving it to a group with the majority of its neighbors from a second parent. The virtual machine type for a resulting group is also dictated from the second parent. The mutation operator randomly selects a component among candidates and moves it to a randomly selected group where an empty group is also considered a possibility.

Figure 20 displays an example of reproduction between two parents where the randomly selected component for reallocation is c_2. The child on the left is derived from the first parent by moving c_2 to a group containing c_3, according to the second parent where c_2 resides alongside c_3. Analogously, the child on the right is derived from second parent by moving the c_2 component to a group with c_1 component. In the case where there is more than one possibility for choosing a resulting group, a group that is most similar to the parent group is chosen. Similarity is measured as the number matching components. Ties are resolved randomly.

5.3. Summary

This section disseminated a process for determining optimal topologies using a meta-heuristic and a queue-network simulation as the objective function. The goal for this process is to provide a cost and time effective process

for recommending topologies to be tested on the real cloud environment. The simulation model presented here has been devised to simulate response time and infrastructure costs quality criteria.

6. Validation

This section disseminates the procedure for validating the proposed optimization method and demonstrating the ElaClo tool on a real-life application. Validation goals include assessing both validity and performance of the proposed optimization process. Validation is conducted using a case study Software-as-a-Service (SaaS) application, *CashRegister* that enables invoicing and stock management for small and medium enterprises in retail business domain. We chose this single application as a *typical case* Easterbrook et al. (2008) of a smaller focused service-based application in order to gain more insight how ElaClo can be applied to similar applications.

Section 6.1 states the research questions for assessing the validity and the performance of the optimization process. Section 6.2 disseminates the methodology for answering them. Section 6.3 elucidates details in implementing ElaClo prototype. Section 6.4 describes the *CashRegister* application from the case study, and Section 6.5 disseminates the results of the statistical analysis. Finally, Section 6.6 enumerates possible limitations of the research.

6.1. Validity levels

The goal of the optimization process proposed is to give software architects an automated procedure for determining optimal elastic application topologies on a cloud infrastructure. According to Böhme & Reussner (2008) there are three levels of validation for predicting the performance of the system using models:

1. **Validating accuracy** states whether metrics obtained from models of system are accurate compared to metrics from a real system. In the context of this work, we are interested in the precision of obtaining optimal candidates from simulation models.
2. **Validation of applicability** assesses whether the user of the system is able to conduct the optimization procedure. For this work, as it is an automated process, the question remains to assess whether the user can obtain all necessary information in order to start the optimization process. Since ElaClo also automatically extract application models from artifacts, the only true precondition for using ElaClo is the difficulty of applying the ElaClo Application framework to the targeted application. The procedure for integrating the application with our framework is presented in our previous work (Tanković et al., 2015b).

3. **Cost/Benefit validation** is concerned with the associated costs and benefits of the optimization procedure. For this validation, we will assess the performance of the algorithm in converging on optimal topologies. We will also compare the results with other well-known heuristics and random search methods as suggested in Aleti et al. (2013).

For validating the accuracy of optimization results we will assess how close the resulting candidates are to optimal results. Since this work relies on an evolutionary algorithm using simulation models, we define the following research questions:

RQ1: How close is the average evolutionary algorithm result to a true Pareto set?

RQ2: To what extent are resulting topologies obtained in simulated environment relevant to real systems and environments?

The ElaClo tool is evaluating topologies in the cloud environment sequentially. Evaluation time for each topology is determined by the total length of the workload model. In order to test elastic behavior properly, workload models should be at least a couple of hours long since a single topology reconfiguration on the cloud infrastructure takes around 2-3 minutes. This means that the number of resulting topologies from the evolutionary algorithm must be relatively small. This can be achieved by setting the underlying genetic algorithm population size (μ parameter) to around 10 to 30 individuals. The total optimization length is primarily determined by the performance of the evolutionary algorithm and selected parameters. Due to the importance of the evolutionary optimization process, we define the final research question in assessing the performance of the algorithm.

RQ3: What are the time-related performance measures of the applied genetic algorithm?

Finally, we are also interested in assessing how the proposed genetic algorithm performs compared to other existing optimization algorithms. That forms the last question.

RQ4: How does the proposed genetic algorithm compare to other applicable optimization algorithms?

6.2. Methodology

In order to answer **RQ1** we use a quantifiable measure of optimization precision called *coverage*:

$$\mathcal{C}(\mathbb{T}^*_{result}, \mathbb{T}^*_{true})$$

that was introduced in (Zitzler & Thiele, 1999). The coverage measure will compare the achieved Pareto front of quality criteria PF_{result} of the respective optimal topologies $\mathbb{T}^*_{result}$ obtained from a single run of genetic algorithm to the true Pareto set PF_{true}, or the best possible quality criteria achieved by $\mathbb{T}^*_{true}$. The coverage measure describes the ratio of non-dominated solutions (topologies) between two sets. We say that topology T_A dominates topology T_B if all quality criteria achieved by T_A are better then quality criteria of T_B and mark that as

$$T_A \succ T_B.$$

Due to the stochastic nature of the applied simulation method, we propose a slight modification of the coverage metric for our experiment, where we shall state the fraction of topologies that yield PF_{result} that are contained in the set of topologies that yield PF_{true}:

$$\mathcal{C}^*(\mathbb{T}^*_{result}, \mathbb{T}^*_{true}) = \frac{|\mathbb{T}^*_{result} \cap \mathbb{T}^*_{true}|}{|\mathbb{T}^*_{true}|} \in [0, 1] \quad .$$

The $\mathcal{C}^*$ metric denotes the ratio of results from the single execution in the true solution $\mathbb{T}^*_{true}$ where $\mathcal{C}(\mathbb{T}^*_{result}, \mathbb{T}^*_{true}) = 1$ means $\mathbb{T}^*_{result} = \mathbb{T}^*_{true}$. This way, solutions are compared to statistically optimal topologies, since individual quality criteria measurements vary.

As a goal for this research we will set a minimal coverage $\mathcal{C}^*$ in average execution to 0.7, which states that in an average optimization process, a minimum of 70% of topologies will be discovered from true solution set $\mathbb{T}^*_{true}$. This will be tested with a one-way Student t-test with the null hypothesis stating that average value of $\mathcal{C}^*$ is 0.7, with the alternative hypothesis being that this value is larger than 0.7. We will set the upper p-value limit to 1%. A precondition to the Student t-test is a normal distribution of the measured value, which will be tested with a Kolmogorov–Smirnov test.

The true Pareto front PF_{true} and topologies $\mathbb{T}^*_{true}$ are not a priori known. Due to the large solution space, computational requirements and the stochastic nature of the simulation, it is not possible to determine exactly. For this reason, we approximate PF_{true} by conducting a series of optimization processes and determining the resulting Pareto set from all results combined. This is standard procedure in multi-objective optimization (Fonseca et al., 2005).

Obtaining the exact optimal topologies by evaluation in the cloud environment is also not practical since a single topology evaluation takes up to

several hours, depending on workload. In addition, due to elasticity variability, each evaluation should be repeated multiple times. Therefore, an alternative approach is required to answer **RQ2**: we conduct additional series of evolutionary optimization in simulation environment, but this time with inverted quality criteria goals that will yield additional set of solutions $\mathbb{T}^*_{inverse}$. Such solutions should represent topologies with higher costs and lower quality attributes. Then, both solution sets: $\mathbb{T}^*_{inverse}$ and $\mathbb{T}^*_{result}$, will be evaluated using ElaClo and a real application. This will result in quality criteria obtained in a real cloud environment: $f_{cloud}(\mathbb{T}^*_{result})$ and $f_{cloud}(\mathbb{T}^*_{inverse})$. We will then assess the number of dominated solutions from $\mathbb{T}^*_{inverse}$ by solutions in $\mathbb{T}^*_{result}$, which must be statistically larger than the number of dominated results from $\mathbb{T}^*_{result}$ by results in $\mathbb{T}^*_{inverse}$. If solution T_A dominates solution T_B with quality criteria measured in a real environment, we shall note this as:

$$T_A \overset{*}{\succ} T_B.$$

For quantifying the dominance count, we introduce a Dominance-Score (DS) metric for a topology T with respect to a topology set $\mathbb{T}$:

$$DS(T, \mathbb{T}) = \sum_{T_i \in \mathbb{T}} f_{DOM}(T, T_i)$$

where

$$f_{DOM}(T_A, T_B) = \begin{cases} 1 & \text{if } T_A \overset{*}{\succ} T_B \\ 0 & \text{otherwise} \end{cases} .$$

DS expresses how many topologies from a certain group are dominated by the topology from which a score is given. The average DS score of a topology in $\mathbb{T}^*_{result}$ in relation to $\mathbb{T}^*_{inverse}$ should be statistically larger then the opposite case. We will compare scores between groups with a Mann-Whitney U test (Mann & Whitney, 1947), also known as Wilcoxon rank-sum test for which a null-hypothesis will state that the DS score for a randomly chosen candidate from the first group will be less than or greater than a randomly selected DS score from the second group. Statistical significance for these tests will be set to $p < 0.01$. We will also express a set of *common-language effect-size* parameters (McGraw & Wong, 1992) for a more intuitive understanding of the given results.

In order to answer **RQ3** we will determine the minimum number of generations in the genetic algorithm required for a target coverage of $\mathcal{C}^* = 0.7$

under a statistical significance of $p < 0.01$. We will also graphically display the average $\mathcal{C}^*$ for each GA generation with the corresponding population quality criteria (average infrastructure cost and response time violation percentage).

Ideally, to answer **RQ4**, the optimization problem should be solved with an exact algorithm. Since that is not possible and the only possible option is applying population-based meta-heuristics (as stated in Section 5), we will separately compare the selection and the local-search parts of the GA with similar heuristics. In comparing the selection process we will compare the applied NSGA-II method with the well known SPEA2 algorithm (Zitzler et al., 2001). For the local-search, which is implemented in cross and mutate operations, we will compare them with a simple Monte Carlo search method (MCS), which will perform a random global search over the candidates. Such comparison was also recommended in a systematic study on architecture optimization provided by Aleti et al. (2013).

6.3. ElaClo reference implementation

In order to evaluate our proposed optimization framework, we implemented ElaClo for evaluating service-based applications built on top of the *JavaEE* component-based development platform. *Workload Generator* has been implemented in Python using the *Tornado*[1] web framework for asynchronous networking. A technique described in (Ross, 2012) was used to generate an NHPP Poisson process according to a user-defined $\lambda(t)$ function. *Topology Monitoring*, *Resource Manager*, *Elastic Controller* and *Topology Manager* have also been implemented as Python web applications. Monitoring and topology data are kept in-memory, since only recent results from all services are needed for scaling decisions. A user interface is created in *HTML* and *JavaScript*, exploiting various open-source components for graphing and charting. HAProxy has been used as a load-balancer for services in different topologies. To provide dynamic switches between different topologies, we have implemented a web service around HAProxy to dynamically reconfigure and reload it according to the current ATG model. More technical details on building load-balancing environment to represent different topologies is provided in (Tanković et al., 2015b). Our ElaClo implementation operates over *DigitalOcean* [2] VPS servers but it can easily be extended to use differ-

[1]Tornado Web Server, available at `http://www.tornadoweb.org/en/stable/`

[2]DigitalOcean: Cloud Hosting company, `www.digitalocean.com`

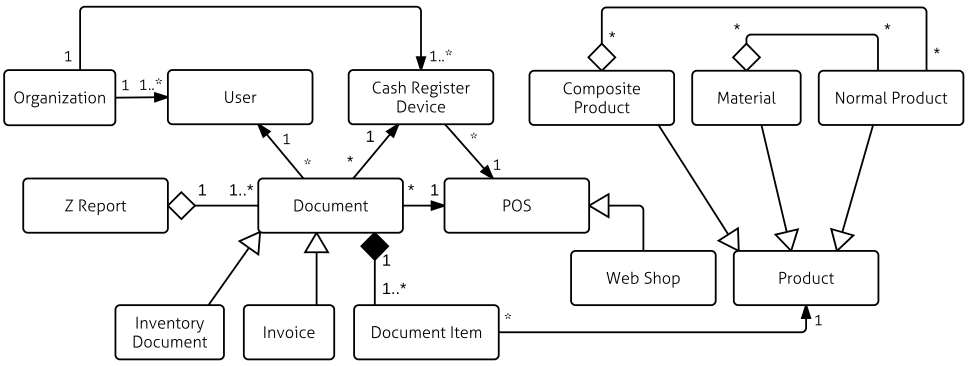


Figure 21: Cash Register Java application domain class model

ent providers (e.g. *Amazon Web Services*) as well. This can enable testing topologies involving multiple cloud providers.

6.4. CashRegister Case Study

A case study used to evaluate ElaClo is from a real-life cloud application *CashRegister* provided in a Software-as-a-Service (SaaS) model. *CashRegister* was developed by the organization *Superius d.o.o.* [3] and currently available in the Croatian, Slovenian and Czech Republic market with more than six thousand daily active users. *CashRegister* is designed for retail services: issuing quotes, invoices, tracking product stock levels, retail reporting, and satisfying *fiscalization legislations* stating that each issued invoice should be electronically verified with a local government dedicated web services.

A reduced class diagram of domain model classes present in *CashRegister* is presented in Figure 21. Organizations, represented by *Organization* class, using *CashRegister* are small and medium businesses selling their products and services in retail shops (*POS*) or at their websites (*WebShop*). Offered *Products* can be composed of *Materials* and *Normal Products* and are sold by creating *Invoice documents*. At the end of each selling cycle, which is typically one day, a *Z-Report* is created containing recapitulation of every issued invoice. *Inventory Documents* are created for replenishing stocks, as *CashRegister* also keeps track of product inventory.

Currently, *CashRegister* is deployed on a private infrastructure and is engineered as a monolithic three-tier application (Figure 22). There is an ongo-

[3] Superius d.o.o. - a SaaS company from Croatia available at `http://www.superius.co`

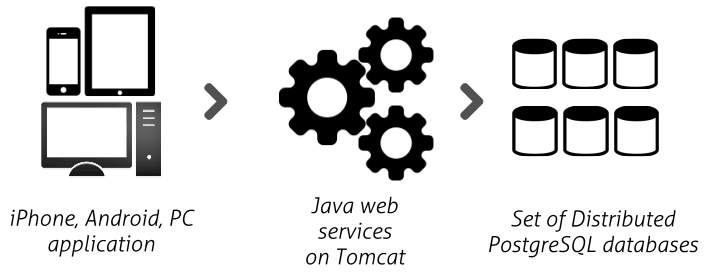


Figure 22: Overview on Cash Register architecture

ing initiative from *CashRegister* service providers to re-engineer *CashRegister* as application exploiting several web services following a service-oriented architecture guidelines and deploy the whole service to cloud infrastructure to achieve elastic capacity due to volatile workload demands.

For our case study, we extracted five major *CashRegister* application components and made them available for communication with the ElaClo environment using the ElaClo Application framework. These components are displayed in Figure 23 and all have distinct functionalities:

- *Reporting* component enables detailed sales reports like daily and monthly recapitulations,
- *Invoice* component is responsible for issuing invoices and visibility to issued invoices,
- *Status* components enables services for data synchronizations between client and server,
- *Fiscalization* components verify issued invoices with government web services,
- *Resource* component enables management over products available for sale.

Since *CashRegister* is a critical business application, consumer SLOs are defined based on response time of critical services like issuing invoices, and ElaClo will be used to determine a set of Pareto optimal topologies w.r.t. infrastructure costs and SLO violations. Table 3 displays predefines SLOs from CashRegister system. Every response time SLO is defined on a 95th

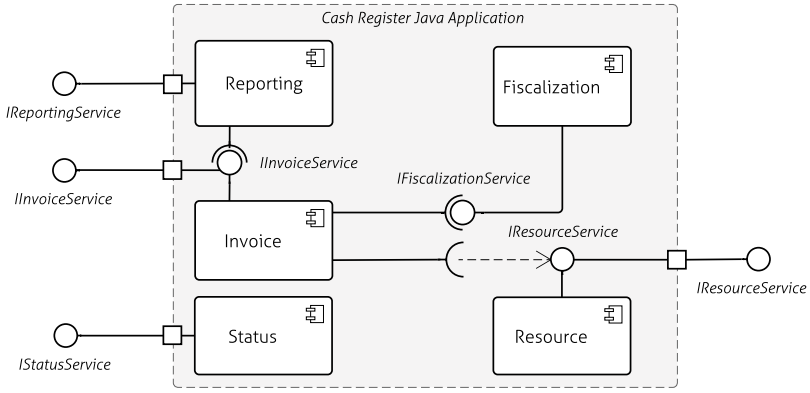


Figure 23: Cash Register Java application components

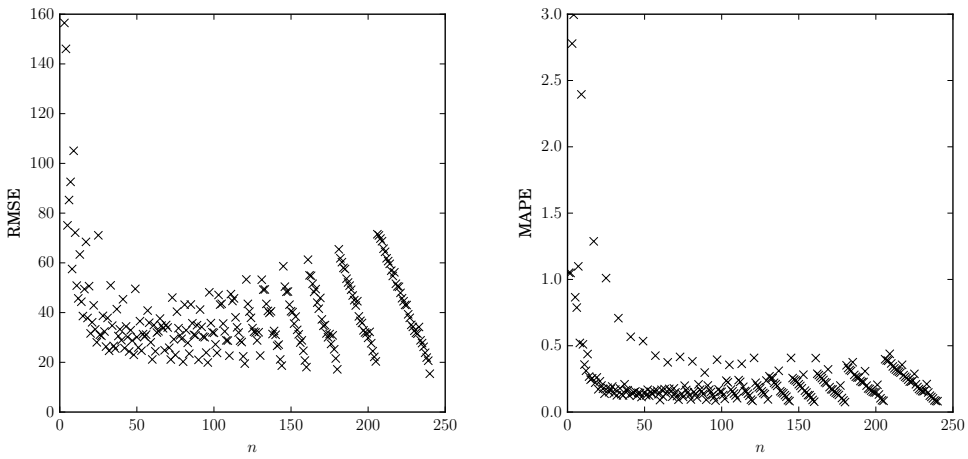
percentile ($\tau_{\%} = 0.95$), which means that response time should be lower than t_{SLA} 95% of the time.

Table 3: SLOs for CashRegister system

Scenario	**Component**	**Description**	t_{SLA}(ms)	$\tau_{\%}$
UC_1	Invoice	Fetch details about invoice	500	0.95
UC_2	Resource	Fetch products and partners data	500	0.95
UC_3	Invoice	Create a new invoice	2500	0.98
UC_4	Report	Fetch daily sale report	4000	0.95
UC_5	Status	Details about system state	300	0.95

To obtain a workload model, we analyzed production logs of the *CashRegister* system, which contained more than 13 million transactions over 40 days of usage. Using these data we constructed a time series of average daily usage intensity (number of requests per unit of time) calculated over one minute periods (1440 entries for 24-hour usage).

We then applied a piece-wise linear interpolation to obtain a workload intensity function $\lambda(t)$ with multiple linear segments $s_1, s_2, ..., s_n$ for a total of n segments. Figure 24 displays the correlation between the number of segments n and both Root Mean Square Error (RMSE) and Mean Absolute Percentage Error (MAPE) between $\lambda(t)$ and the original time-series. We used $n = 240$ segments for $\lambda(t)$. Figure 25 displays the $\lambda(t)$ workload model with $n = 240$ compared to the original workload data.



(a) RMSE (b) MAPE

Figure 24: RMSE and MAPE errors of workload model against real workload depending on number of segments n

For elasticity strategy we used a common rule-based approach with lower and upper CPU usage thresholds that trigger scale-up and -down action.

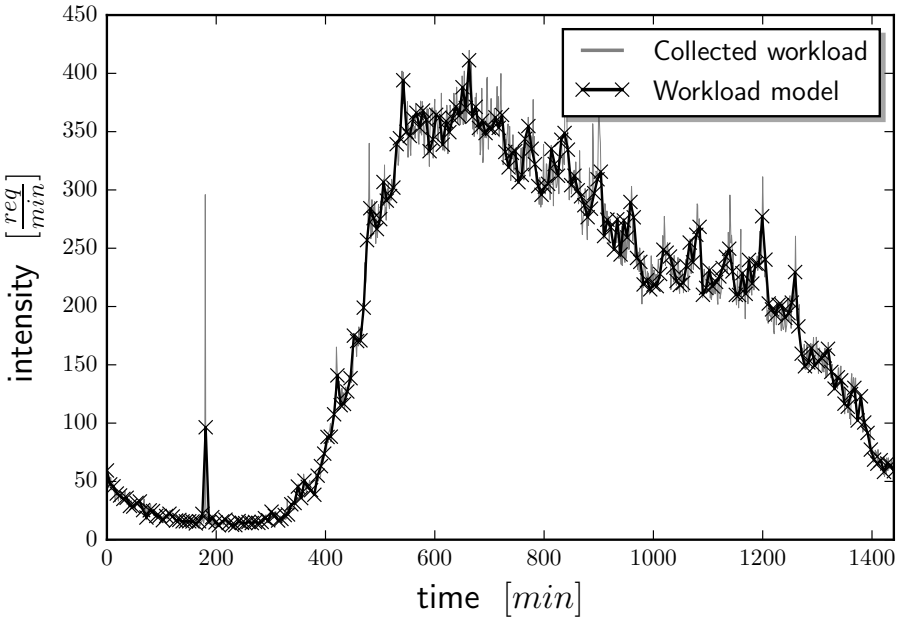


Figure 25: Comparison between real workload and piece-wise linear model

6.5. Validation of the optimization process

CashRegister application has five components, and the optimization process will differentiate among three virtual machine types (Table 4), which produces a total of 1561 available topologies according to Equation 2.

Table 4: Virtual machine types considered in the optimization process

Type	**Name**	**CPUs**	**RAM [GB]**	**Price/h**
VM_1	Small	1	1	0.015
VM_2	Medium	2	2	0.03
VM_4	Large	4	4	0.06

In order to formulate the simulation model we must extract the response time distributions for every operation provided by every component. These distribution are collected by ElaClo automatically by running the application on a minimal single user workload. Obtained measurements are fitted as a *log-normal* distribution, which is a better fit to response times than the commonly used exponential distribution. Figure 26 shows a comparison between a lognormal and an exponential distribution for modeling response time. This especially holds for lower end values of response time (below 1000 ms for given example). Lognormal distributions were parameterized using the first two moments of real distributions.

Table 5 gives the parameterization values of the lognormal response time distributions for all operations, where lognormal distributions are defined by two parameters:

$$\ln \mathcal{N}(\mu,\, \sigma^2)\,,$$

a location parameter $\mu \in \mathbb{R}$ and a scale parameter $\sigma > 0$, which are computed against the first two moments of real distribution according to:

$$\mu = \ln\left(\frac{m^2}{\sqrt{v+m^2}}\right), \sigma = \sqrt{\ln\left(1+\frac{v}{m^2}\right)},$$

where m is the measured first moment (mean), and v the second moment (variance).

With parameters needed for the simulation model defined, we can proceed to the process of search-based optimization using the genetic algorithm with a queue network simulation model. With the algorithm presented in Section

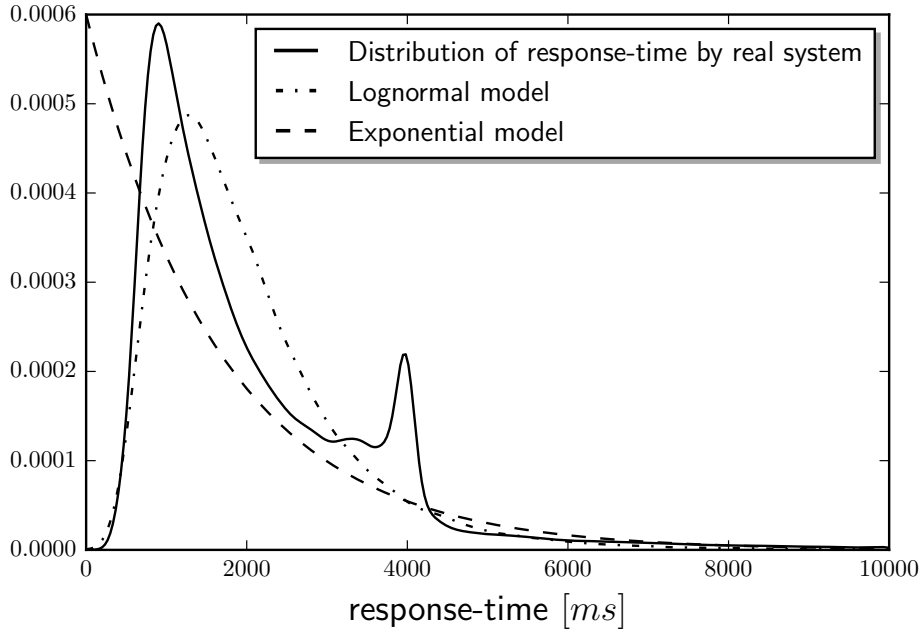


Figure 26: Comparison between response time distribution for *SaveInvoice* operation on our *CashRegister* case study between real system, and models using lognormal and exponential distribution

Table 5: Parameters of lognormal distribution of response time for individual operations

Operation	**Component**	**Description**	μ	σ
CreateInvoice	*Invoice*	Creating invoice	-0.39	0.76
Fiscalize	*Fiscalization*	Fiscalization of invoice	-1.12	0.64
GetStatus	*Status*	System status report	-2.58	1.39
GetReport	*Report*	Generating sales report	0.49	0.71
InvoiceList	*Invoice*	Fetching invoice details	-1.13	0.94
GetResources	*Resource*	List products and customers	-1.99	1.46

5 we have narrowed the search space $\mathbb{T}$ to recommended 8 candidates optimal in the simulation environment. Parameters used by the genetic algorithm, including population size μ, new candidates per generation λ, and the total number of generations n, were set to:

$$\mu = 30,\ \lambda = 12,\ n = 100\ .$$

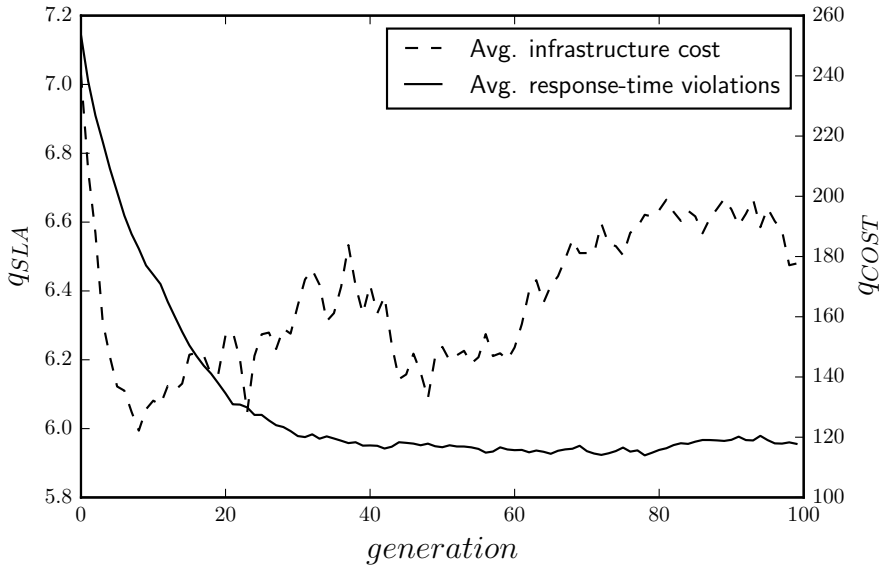


Figure 27: Average infrastructure cost and response time violations for each generation in GA

The average genetic algorithm execution time on a virtual machine with a single core was about 12.5 hours. Due to the possibility of parallelizing evaluations in the genetic algorithm, a 4-core virtual machine was used, which shortened average execution time to four hours. The process can be further optimized by using up to λ cores, which is the maximum number of new evaluations per genetic algorithm iteration (population). Figure 27 gives average values for simulated quality criteria for each population so we can observe the progress of genetic algorithm. We observe an increasing trend in average costs towards the larger iterations, due to large variance of costs in a set of Pareto-optimal solutions.

Table 6 enumerates the obtained solution set after running GA with q_{SLA} being the amount of requests exceeded t_{SLA} given in Table 3, and q_{COST} the total infrastructure leasing cost during simulation according to the cost model introduced in 3.3 and the prices set in Table 4. Obtained solution set can now be further analyzed using real topologies in cloud environment with ElaClo.

Simulated results indicate that it is better to use smaller virtual machine types. Small machines lower the infrastructure costs of low intensity workload

Table 6: Pareto-set candidates obtained with genetic algorithm in simulation environment

Topology T		q_{COST}	q_{SLA}
$G_1 = \{Fiscalization, Invoice, Report, Resource, Status\}$	$type(G_1) = VM_1$	84.00	21.52
$G_1 = \{Fiscalization, Report, Resource\}$ $G_2 = \{Invoice, Status\}$	$type(G_1) = VM_1$ $type(G_2) = VM_1$	85.00	9.59
$G_1 = \{Fiscalization, Report, Resource, Status\}$ $G_2 = \{Invoice\}$	$type(G_1) = VM_1$ $type(G_2) = VM_1$	86.00	8.64
$G_1 = \{Fiscalization, Report, Status\}$ $G_2 = \{Invoice, Resource\}$	$type(G_1) = VM_1$ $type(G_2) = VM_1$	87.00	5.99
$G_1 = \{Fiscalization, Report, Status\}$ $G_2 = \{Resource\}$ $G_3 = \{Invoice\}$	$type(G_1) = VM_1$ $type(G_2) = VM_1$ $type(G_3) = VM_1$	95.00	4.90
$G_1 = \{Fiscalization, Report\}$ $G_2 = \{Resource\}$ $G_3 = \{Invoice\}$ $G_4 = \{Status\}$	$type(G_1) = VM_2$ $type(G_2) = VM_1$ $type(G_3) = VM_1$ $type(G_4) = VM_1$	147.00	3.01
$G_1 = \{Report, Resource\}$ $G_2 = \{Status\}$ $G_3 = \{Invoice\}$ $G_4 = \{Fiscalization\}$	$type(G_1) = VM_1$ $type(G_2) = VM_1$ $type(G_3) = VM_1$ $type(G_4) = VM_2$	154.00	2.84
$G_1 = \{Fiscalization\}$ $G_2 = \{Report\}$ $G_3 = \{Resource\}$ $G_4 = \{Invoice\}$ $G_5 = \{Status\}$	$type(G_1) = VM_1$ $type(G_2) = VM_4$ $type(G_3) = VM_1$ $type(G_4) = VM_1$ $type(G_5) = VM_2$	259.00	1.09

periods. Smaller virtual machines also enable a more fine-grained elasticity realization where current capacity can follow current workload demands more precisely. The benefit of having larger virtual machines is their ability to handle high frequency workload changes where elasticity mechanism are too slow to react (adding additional virtual machines takes at least two minutes on most cloud providers).

Before answering the given research questions, we evaluated whether the evaluations of topologies in simulated environment are consistent. For that purpose we conducted a Kruskal-Wallis H non-parametric statistical test (Kruskal & Wallis, 1952) with a null-hypothesis that all results come from the same distribution regardless of simulated topology. Inability to reject that hypothesis would show that different topologies have no impact on selected

Table 7: Results of Kruskal-Wallisovog H non-parametric test

Quality criteria	Test result	
	p-value	% of compared topologies
q_{COST}	<0.05	93.5%
	<0.01	91.3%
q_{SLA}	<0.05	76.2%
	<0.01	69.6%

quality criteria. We ran 58 different topologies through the Kruskal-Wallis H test. These 58 topologies had at least ten simulation evaluations conducted through the process of optimizations. Table 7 gives the results of this test: we can reject the null-hypothesis with $p < 0.01$ for more that 91% of compared topologies for the criteria of infrastructure cost, and more than 69% of topologies for the response time criteria. Results support the assumption that our surrogate model produces converging evaluation results.

6.5.1. Performance and accuracy

To validate the performance aspects of the genetic algorithm we conducted a series of 40 optimization runs. Figure 28 displays a heat map of all evaluated topologies compared to the quality of candidates in a true Pareto-set forming a Pareto frontier PF_{true}. As noted, for answering **RQ1** and **RQ3**, we used the coverage metric. Optimization runs were configured to perform 100 generations ($n = 100$), which resulted in 100 datasets with a population of 40 samples, with each sample coming from single run. We performed a Student t-test for the data from each generation to assert whether the coverage goal was achieved. Results indicate that all generations before the 88th failed to reject the null-hypothesis. For the 88th iteration, the null-hypothesis was rejected with $p - value < 0.01$, which went down to $p - value < 0.001$ by the 100th iteration. We can conclude that on average it takes 88 generations to reach our coverage goal of $\mathcal{C}^* = 0.7$. By achieving targeted coverage we conclude that the precision of the algorithm is satisfactory (**RQ1**) and that it takes less then 100 generations of the algorithm to achieve it (**RQ3**) Figure 29 displays the convergence of coverage metric collected through algorithm generations: average coverage from 40 runs together with the standard deviation.

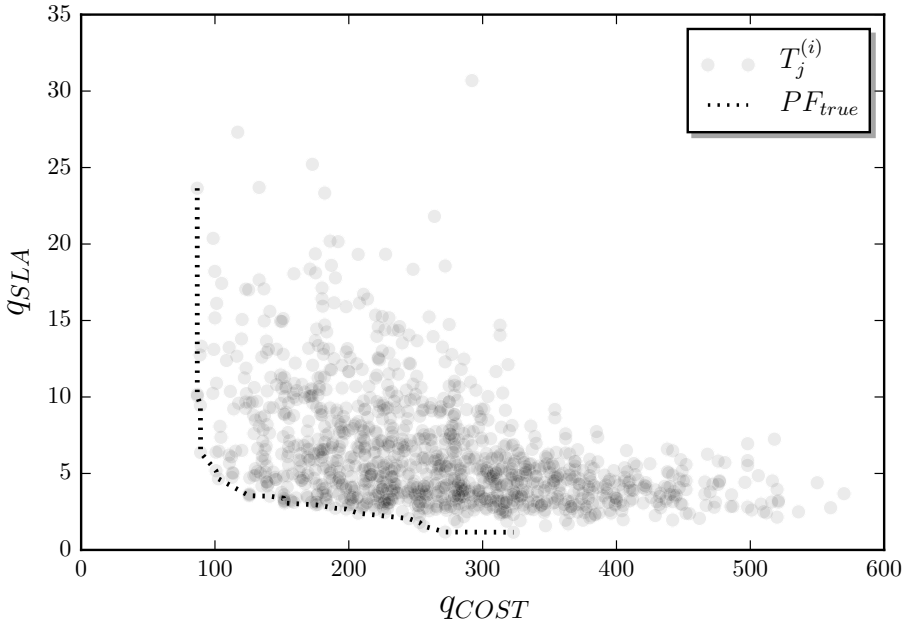


Figure 28: All evaluated topologies from 40 optimization runs ($i \in [1, 40]$) compared to approximated real Pareto set PF_{real}

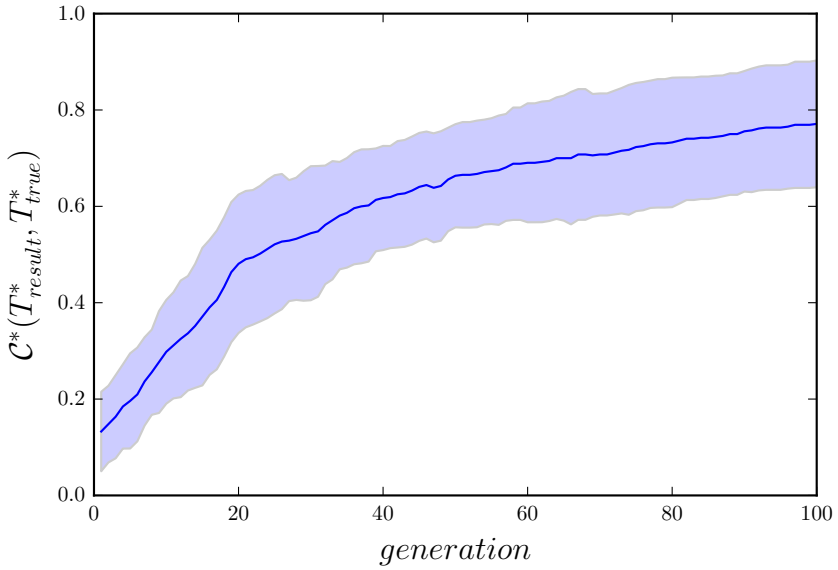


Figure 29: Average coverage metric $\mathcal{C}^*(\mathbb{T}^*_{result}, \mathbb{T}^*_{true})$ through generations of GA

In order to answer **RQ2** and test if optimal candidates from a simulated environment selected by GA (Table 6) also achieved optimal properties in a real environment, we evaluated their quality criteria using cloud resources with ElaClo. All suggested optimal candidates from GA were iteratively evaluated. Figure 30 shows the achieved quality attributes on the cloud environment for optimal topologies suggested by GA. We observe that the majority of candidates are non-dominated, which means that the results from GA are indeed indicative. Table 8 displays the result of the evaluation in a real environment with the final list of optimal topologies.

The second step was to obtain a set of least optimal candidate topologies in a simulated environment by the inverting evaluation goals, displayed in Figure 31. Their quality criteria was again evaluated by ElaClo on a real cloud infrastructure. The quality criteria of the best and worst topologies from GA are displayed in Figure 32. We observe that the best candidates from the simulation retain the same result in a real environment.

To show the similarities in performance between real and simulated environments, we computed the DS score (defined in Section 6.2) between both sets of solutions with a Mann-Whitney U test and successfully rejected the

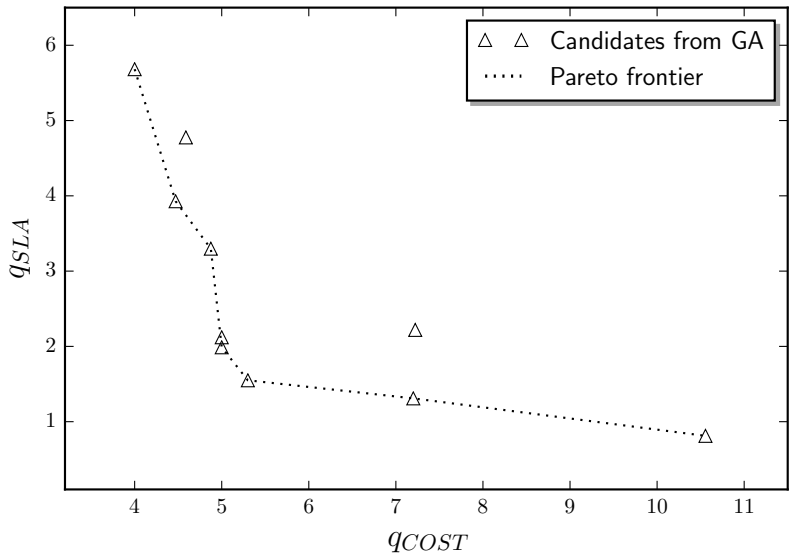


Figure 30: Real cloud infrastructure quality criteria of candidates selected by GA

null-hypothesis in favor of the alternative hypothesis ($p < 1 \cdot 10^{-4}$) with a U-statistic of 100.0. Therefore, quality relations achieved with GA a in simulated environment hold also for real cloud environments.

Common-language effect-sizes are also obtained: dominance ratio between two groups of solution sets is 38%, meaning that when comparing each two solutions between both groups, in 38% cases, the topology from first group yields both lower costs and lower amount of response time violations. Further, in 97% of cases, the first group yields lower infrastructure costs, and finally, in 41% of cases yields a lower amount of response time violations.

For the given case-study, we observe that the optimization process is efficient at selecting topologies that yield lower infrastructure costs. At the same time, it is able to locate topologies that achieve similar or better results in terms of response time violations. From the resulting effect-sizes, we conclude that our automatically generated MC-OQN performance model is better at assessing cost than response time violations. This is expected since it is very hard to model response time precisely without carefully manually designed performance models like LQN-s. However, the goal of optimiza-

Topology T		q_{COST}	q_{SLA}
$G_1 = \{Fiscalization, Invoice, Report, Resource, Status\}$	$type(G_1) = VM_1$	4.00	5.68
$G_1 = \{Fiscalization, Report, Resource, Status\}$ $G_2 = \{Invoice\}$	$type(G_1) = VM_1$ $type(G_2) = VM_1$	4.88	3.30
$G_1 = \{Fiscalization, Report, Status\}$ $G_2 = \{Invoice, Resource\}$	$type(G_1) = VM_1$ $type(G_2) = VM_1$	5.00	1.99
$G_1 = \{Resource\}$ $G_2 = \{Invoice\}$ $G_3 = \{Fiscalization, Report, Status\}$	$type(G_1) = VM_1$ $type(G_2) = VM_1$ $type(G_3) = VM_1$	5.30	1.55
$G_1 = \{Status\}$ $G_2 = \{Report, Resource\}$ $G_3 = \{Fiscalization\}$ $G_4 = \{Invoice\}$	$type(G_1) = VM_1$ $type(G_2) = VM_1$ $type(G_3) = VM_2$ $type(G_4) = VM_1$	7.20	1.31
$G_1 = \{Fiscalization\}$ $G_2 = \{Report\}$ $G_3 = \{Resource\}$ $G_4 = \{Invoice\}$ $G_5 = \{Status\}$	$type(G_1) = VM_1$ $type(G_2) = VM_4$ $type(G_3) = VM_1$ $type(G_4) = VM_1$ $type(G_5) = VM_2$	10.56	0.81

Table 8: Final set of optimal topologies after evaluation in the cloud

tion in a simulation environment is to suggest optimal candidates and since the final evaluation is performed on real infrastructure, we regard the model informative and expressive enough for that task.

The optimization results reveal that there are non-trivial efficient topologies besides the commonly applied topologies that either deploy components separately or in a single composite. Results suggest that for our case study, costs can be significantly reduced by combining certain components. Such topologies induce only a minor penalty in overall response time, which does not result in SLA violation.

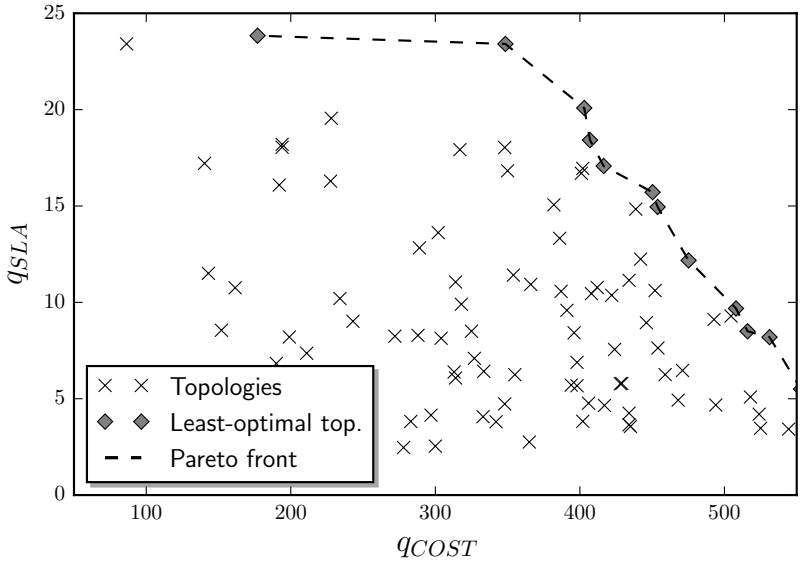


Figure 31: Pareto-front achieved in simulation environment under inverted evaluation goals

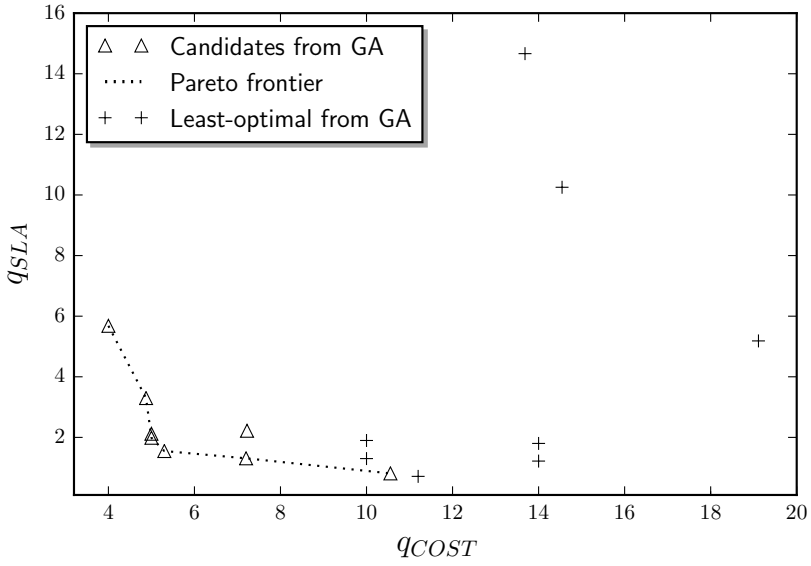


Figure 32: A comparison of quality criteria between best and worst candidates from GA evaluated on a real cloud environment

6.5.2. Comparison

We performed a series of 20 optimization runs for each algorithm variant: NSGA-II, SPEA2 and MCS. We computed the coverage metrics and stored the resulting populations with their respecting QoS values. Table 9 displays the results. NSGA-II and SPEA2 selection algorithms are very similar, which corroborates the previous comparisons of these algorithms on other optimization problems in noisy environments (Bui et al., 2004). NSGA-II tends to conduct more evaluations, and is thus more expensive than SPEA2, but it results in somewhat better coverage. The proposed local search methods in GA enable better convergence that is observable by the degraded performance of MCS. NSGA-II and SPEA2 with local searches achieve a 37% better coverage on average for 25% fewer evaluations. This suggests that applying local search through crossing and mutation operations accelerates algorithm convergence. Figure 33 shows an example result from a single optimization run performed with all three variants. MCS failed to sufficiently reach the Pareto frontier. We also supported these claims by performing a Mann-Whitney rank test on the coverage values achieved by all algorithms. Table 10 displays the comparison among all algorithm pairs. NSGA-II and SPEA2, while there is no statistically significant difference between them, significantly outperform MCS, which empirically supports our proposed methods for conducting local search in GA.

Table 9: A comparison of coverage, number of evaluations, average resulting topology costs and SLA violations within different search algorithms

	Coverage $\mathcal{C}^*$	# of evaluations	q_{COST}	q_{SLA}
MCS	0.565 ± 0.09	128.7 ± 15	181.1 ± 102.8	5.9 ± 4.8
SPEA2	0.773 ± 0.16 (+36.8%)	90.3 ± 6 (-29.8%)	121.1 ± 32.9	6.3 ± 5.3
NSGA-II	0.781 ± 0.13 (+38.2%)	100.7 ± 13 (-21.8%)	119.7 ± 36.6	6.7 ± 5.5

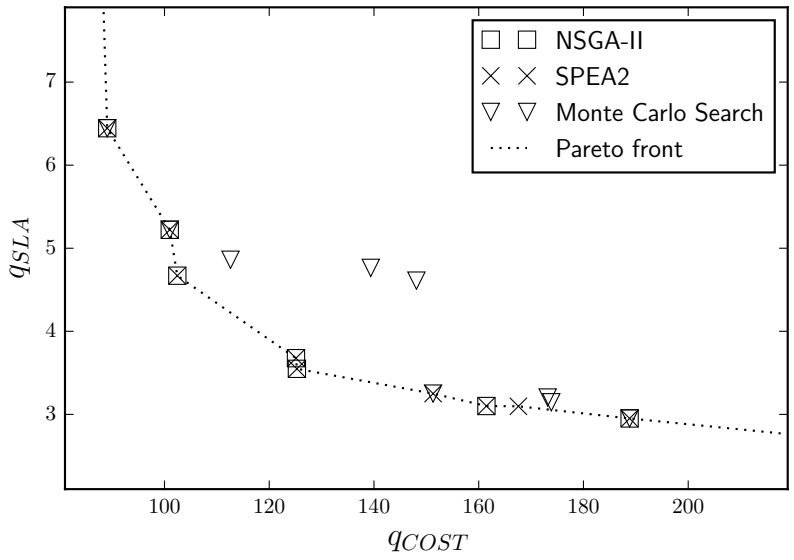


Figure 33: A comparison of resulting Pareto sets from different algorithms

Table 10: Results of the Mann-Whitney rank test among coverages obtained by running different optimization algorithms; the alternative hypothesis states that algorithms listed in rows have better coverage against algorithms in columns

	SPEA2		**MCS**	
	H	*p-value*	*H*	*p-value*
NSGA-II	890.0	0.190	1436.5	$< 10^{-9}$
SPEA2	-		1402.0	$< 10^{-9}$

6.6. Limitations

Our research demonstrates a solution for optimizing application topologies of the implemented software components in a cloud environment exploiting elasticity capabilities. Since there are many different architectures from which a software application can be implemented, we will share the known limitations of the current framework together with guidelines for future research.

The ElaClo framework is intended to be used with a component-based application where components are provided as web services to front-end components. Large enterprise solutions where there are multiple levels of composed components, possibly organized around an enterprise service-bus (ESB) are currently not supported. Additionally, ElaClo can optimize topologies from components that are stateless, which is also a precondition for achieving elasticity with horizontal scaling. As such, it cannot be employed on optimizing configurations of components for data persistence like databases and cashing mechanisms. This means that such components should not be included in the topology optimization process. Application developers should take care to provide alternative components that can *mimic* their behavior or ensure that these components are deployed independently of the ElaClo cloud framework. This should not be an issue if cloud-based storage services are used, since they already provide elastic storage capabilities.

The surrogate model applied could also be improved with ability to model I/O congestions that components experience towards media such as harddisks or outer web services that are not part of the application topology. Currently, the simulation model is simple, but nonetheless, proves to be effective in yielding optimal topology suggestions.

There are also threats to the validity of this research that should be considered (Jedlitschka et al., 2008):

- **Construct validity**. The assumption of the ATG model is that elasticity can be achieved by simple horizontal scaling of components. There could be cases where a more complex method of scaling is required based on the possibility that some functionality could be hard to implement in a *stateless* way.

- **External validity**. ElaClo is evaluated on a single case study. Although we consider the *CashRegister* representative, and realistic case used by thousands of users, additional case studies could reveal untested difficulties in applying ElaClo.

7. Related Work

The development of effective software topologies and deployments in the cloud has not been thoroughly studied. Most previous research focused on elasticity attribute of software processes (Dustdar et al., 2011) or on underlying infrastructure (Dhingra, 2014). An extensive review on both types of this research is available at (Ardagna et al., 2014). When researching impact of the software topologies, most work has concentrated on the deployment of application components divided by architectural layers (e.g. database, load-balancing, application servers) (Leymann et al., 2011), or large-scale placement of independent software applications to cloud computing centers (Li et al., 2011a) on an Infrastructure-As-A-Service level where optimization is carried for the benefit of the infrastructure provider. A similar problem has been studied in the form of deducing optimal service compositions where a pool of existing services has been provided with different quality attributes (Teixeira et al., 2015). Web services in this context are mostly used in *Programming in the large* models (e.g. BPEL) Jula et al. (2014). The benefits of such optimization is achieved by service users that strive to use the most efficient services.

Our work considers optimization that benefits service providers in providing cost-effective services (*Programming in the small*). End services should offer competitive SLAs by optimizing topology of internal software components through their optimal integration with infrastructure resources. Research with similar optimization goals is mostly oriented at embedded systems (Martínez-Álvarez et al., 2013; Li et al., 2011b; Bhuvaneswari, 2015; Aleti et al., 2009), or specialized sub-systems like databases (Zhao et al., 2016), but cloud provisioning environments introduce dynamic deployment capabilities (e.g. through elasticity), which adds additional complexity to available optimization techniques.

Huang & Shen (2015) studied service deployment in the SaaS and SOA environments, similar to the context of our study. They proposed an integrated approach to service deployment for reducing service execution times. They took into consideration inter-service communication costs and potential parallelism among services. They rested their solution on solving graph problems. Their study was primarily targeted at applications with predefined workflows. Our study additionally takes into consideration an important property for SaaS applications: workload mix and intensity variation, which are the main drivers of elasticity attributes in cloud applications.

Hadded et al. (2015) proposed an algorithm for deciding the number of elasticity controllers in service-based applications, grouping services in elastic groups similar to ElaClo. The input for their algorithm was the service dependency graph, as the algorithm took into considerations only the *dependency* relationships among services. ElaClo executes on real infrastructure, and while it takes more time to execute tests, it gives the ability to specify test workload and SLA criteria, and compare end-results among topologies.

Another work targeting optimal resource allocation is the ROAR modeling framework (Sun et al., 2015). ROAR simplifies and optimizes cloud resource allocation to meet QoS standards. It is best used for layered architectural designs where ROAR enables the modeling of incoming workloads and uses cloud infrastructure to determine minimum VM number to satisfy QoS goals. ElaClo's goal is similar to ROAR, in that goal but with some distinctive differences: ElaClo is oriented at optimizing topology at the component level, while ROAR targets multi-tier architectures optimizing resource amounts for each tier. Additionally, ROAR does not include testing of elasticity behavior in dynamically provisioning underlying infrastructure according to current demand levels. Optimizing topology that exhibits elasticity is crucial due to effects that different topologies have on scaling.

Król & Kitowski (2016) explored service self-scalability in large-scale software platforms. They proposed grouping functionality in self-scalable services and applied their findings in a data-farming domain. A similar concept is used in ElaClo, which finds optimal topologies by proposing self-scalable elastic groups of components with further evaluation of each proposed topology.

The S-IDE tool (Celik & Tekinerdogan, 2013) is also used for infrastructure allocation decisions, but in simulation systems domain. S-IDE effectively allocates resources at the deployment phase in simulation systems where simulation modules can be deployed to physical resources in many different ways. S-IDE automatically derives feasible deployment alternatives, and then uses CTAP (Capacitated Task Allocation Problem) (Zheng et al., 2007) to evaluate each solution.

Andrikopoulos et al. (2014) have been studying optimal application topologies in the cloud as well. Their research was oriented at providing custom utility functions for optimality selection. Their research does not yet address variable workloads, so no elasticity is considered.

Lloyd et al. (2013) conducted an empirical study on the performance of all deployment variants for an multi-tier applications consisting of four

components, resulting in 15 different topologies. They displayed an performance variation of ĩ-2% due to isolating application components to separate VMs. The main difference between their research and ours is that they applied horizontal decomposition based on layers (e.g. application server, database server, file server, proxy server) whereas ElaClo works with components differentiated by functional requirements (e.g. invoicing, reporting, system services). This is an important difference since we displayed much larger performance variations. Lloyd et al. indicated that service isolation entails additional resources but could be beneficial for enabling fine-grained elasticity. This claim was also proven by the results we obtained, since workload we applied was non-deterministic, using a non-homogeneous Poisson process to simulate large variation in the number of clients. Emphasizing variability in a cloud environment is crucial, and has been recognized as an important research factor (Schwarzkopf et al., 2012).

There are also many research efforts to optimize overall application architectures. They primarily focus on optimization at design-time based on software models (Franceschelli et al., 2013; Etemaadi & Chaudron, 2015; Ashraf et al., 2015). Design-time optimization methods rely on detailed system models (e.g. Markov chains, Petri-nets, Queuing network models) and thus require an experienced architect (Harchol-Balter, 2013). Evolutionary algorithms are exploited to evaluate large design-spaces. Precision of the results of design-time optimizations are inherently limited by the accuracy of the developed software performance model.

The following sections give a more detailed description of existing architecture optimization tools.

7.1. ArcheOpterix

ArcheOpterix (Aleti et al., 2009) is a tool developed for optimizing embedded component-based systems. It is based on providing AADL specifications [4] and quality criteria requirements for targeted systems. The current ArcheOpterix tool version can optimize deployments of components to hardware nodes of different type w.r.t. minimal communication between hardware nodes. ArcheOpterix uses evolutionary algorithms and is implemented as an Eclipse plug-in.

[4] Architecture Analysis and Design Language (AADL), available at `http://www.aadl.info/`

7.2. PerOpteryx

PerOpteryx (Koziolek et al., 2011) is a tool used for optimizing component-based system models, not necessarily in embedded form. PerOpterix is based on defining different degrees of freedom that systems configurations can achieve such as selection of alternative components, server types and deployment options. The optimization phase emphasizes evolutionary algorithms with the evaluation functions based on simulation using Layered Queue Networks. Similar to ArcheOpterix, PerOpterix is implemented as Eclipse plug-in.

7.3. SPACE4CLOUD

SPACE4CLOUD (Franceschelli et al., 2013) provides Palladio component model (PCM) (Reussner et al., 2011) extension that enables design-time performance and cost estimation for cloud systems. SPACE4CLOUD defines a cloud information system meta-model that enables multi-cloud setups, and it supports modeling of volatile workloads and performance analysis for a given cloud PCM model.

A new tool combining SPACE4CLOUD and PerOpteryx has been announced (Ciavotta et al., 2015) that enables searching the design space for optimal deployment configuration of software components. The system is envisioned in two stages: (1) applying PerOpteryx to find the best configuration under peak workload, then (2) applying SPACE4CLOUD to optimize virtual machine types for a complete 24-hour workload given a configuration from the first step.

The ElaClo tool combines these two steps in a single simulation model in order to explore the effects of volatile workload and elasticity mechanisms to quality and costs under different configurations. We believe that this process cannot be separated into two independent problems because of the combined effects among topologies and run-time elasticity efficiency on different quality criteria.

7.4. Discussion

Based on the analysis of the related work in architecture optimization, the biggest novelty of the ElaClo framework is the hybrid approach to architecture optimization: combining optimization techniques from design-time modeling with performance evaluation tools based on measurement. ElaClo's intention is to be complementary to existing design-time tools with its main application in the assembly phase, where software components are

ready to be deployed and evaluated. Performance models can then be extracted automatically from given application components with more accurate results and used to automatically suggest promising topologies. ElaClo also provides all the automation required to evaluate such topologies on real cloud infrastructure which significantly reduces expert user effort.

In ElaClo, the extracted performance model is based on multi-class queuing networks with general response-time distributions. Similar to design-time optimization approaches, ElaClo also employs genetic algorithms coupled with queue network simulations to evaluate each candidate. ElaClo is unique in arranging components into elastic groups and simulating elasticity behavior. Elasticity is simulated through automatic addition and removal of computing nodes based on node utilization. Finally, recommended topologies from the simulation environment are re-evaluated on real cloud infrastructure to obtain the final results.

The downside of our framework compared to existing design-time approaches is the application of simpler performance models. Layered Queue Networks employed in previous works can express more sophisticated interactions between software components. However, these interactions are not trivial to extract automatically so we have compensated this loss of precision by applying measured service time distributions and the ability to model the elasticity within the simulator.

8. Conclusions and Future Work

This paper introduced the ElaClo framework for optimizing topologies of service-based applications according to response-time based service-level objectives and cloud infrastructure operation costs. ElaClo provides these capabilities by providing a development framework to be used in targeted application together with a cloud based test-bed service with an integrated simulation environment to accelerate optimization process. By using ElaClo, developers are empowered with meaningful information about topology evaluation and optimization processes and so they can target and offer feasible service-level objectives.

We demonstrated the capabilities of ElaClo on a real-life application and displayed that there are non-trivial topologies available that perform better than using trivial deployments. Since the space of all possible topologies is vast, ElaClo determines optimal topology candidates using a simulation environment by automatically extracting simulation models based on measurement from the real application components. Suggested topologies were re-evaluated on a real infrastructure under same volatile workload model, exploiting ElaClo to automatically manage the cloud topology, generate workload, monitor and enable service elasticity. We applied several statistical tests to evaluate the accuracy and efficiency of the optimization process in simulated environment and concluded that applying genetic algorithms on simple queue network performance models can differentiate topologies that tend to be optimal when tested in the real environment.

The largest challenge in applying ElaClo is the process of integrating ElaClo's application framework into applications that needs to optimized. This requires additional effort from service providers and limits ElaClo applicability in different component framework implementations. For that reason, future work will involve integrating application containers (e.g. Docker) to facilitate managing incremental changes to component artifacts throughout development process, enabling comparison of topologies across different application versions and consisting of components with different technological background (e.g. application servers, implementation languages). The possibility to generate TOSCA (Katsaros et al., 2014) models based on chosen topologies will also be provided to ease up further deployment in production environments. Additionally, integration with OpenStack infrastructure APIs will be implemented to enable testing topologies spanning across multiple data centers (Katsaros et al., 2014).

The most important area for future improvement includes enabling even faster techniques for searching the possible solution-space based on analytical models and solutions. We are currently researching automatic extraction of Layered Queue Network (LQN) models from application artifacts and applying analytical solvers like LQNS (Franks et al., 2009). There is also a need to establish an automated evaluation framework that can validate surrogate model speed and precision. By knowing that, the final optimization framework would utilize multiple cost functions: (1) analytical models, (2) simulation models, and (3) measurement on real systems. The most appropriate technique should be automatically selected based on higher-level goals (e.g. optimization duration or precision) set by the application's architects.

Acknowledgment

This work has been supported in part by Croatian Science Foundation's funding of the project 860 UIP-2014-09-7945 and by the University of Rijeka Research Grant 13.09.2.2.16.

References

Akinnuwesi, B. a., Uzoka, F.-M. E., Olabiyisi, S. O., & Omidiora, E. O. (2012). A framework for user-centric model for evaluating the performance of distributed software system architecture. *Expert Systems with Applications*, *39*, 9323–9339. doi:10.1016/j.eswa.2012.02.067.

Aleti, A., Björnander, S., Grunske, L., & Meedeniya, I. (2009). ArcheOpterix: An extendable tool for architecture optimization of AADL models. *Proceedings of the 2009 ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software, MOMPES 2009*, (pp. 61–71). doi:10.1109/MOMPES.2009.5069138.

Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., & Meedeniya, I. (2013). Software Architecture Optimization Methods: A Systematic Literature Review. *IEEE Transactions on Software Engineering*, *39*, 658–683. doi:10.1109/TSE.2012.64.

Ameller, D., Galster, M., Avgeriou, P., & Franch, X. (2015). A survey on quality attributes in service-based systems. *Software Quality Journal*, (pp. 1–29). doi:10.1007/s11219-015-9268-4.

Andrikopoulos, V., Gómez Sáez, S., Leymann, F., & Wettinger, J. (2014). Optimal Distribution of Applications in the Cloud. In M. Jarke, J. Mylopoulos, C. Quix, C. Rolland, Y. Manolopoulos, H. Mouratidis, & J. Horkoff (Eds.), *Advanced Information Systems Engineering SE - 6* (pp. 75–90). Springer International Publishing volume 8484 of *Lecture Notes in Computer Science*. doi:10.1007/978-3-319-07881-6_6.

Ardagna, D., Casale, G., Ciavotta, M., Pérez, J. F., & Wang, W. (2014). Quality-of-service in cloud computing: modeling techniques and their applications. *Journal of Internet Services and Applications*, *5*, 11. doi:10.1186/s13174-014-0011-3.

Ashraf, A., Byholm, B., & Porres, I. (2015). A Multi-Objective ACS Algorithm to Optimize Cost, Performance, and Reliability in the Cloud. In O. Rana, & M. Parashar (Eds.), *8th IEEE/ACM International Conference on Utility and Cloud Computing* (pp. 341–347). IEEE. doi:10.1109/UCC.2015.54.

Back, T., Fogel, D., & Michalewicz, Z. (2000). Evolutionary Computation 1: Basic Algorithms and Operators. *Evolutionary Computation*, (p. 378).

Bhuvaneswari, M. C. (2015). *Application of evolutionary algorithms for multi-objective optimization in VLSI and embedded systems*. Springer. doi:10.1007/978-81-322-1958-3.

Bianchi, L., Dorigo, M., Gambardella, L. M., & Gutjahr, W. J. (2009). A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, *8*, 239–287. doi:10.1007/s11047-008-9098-4.

Binz, T., Breitenbücher, U., Kopp, O., Leymann, F., Binz, T., Breitenb, U., Kopp, O., Binz, T., & Breitenb, U. (2014). TOSCA: Portable Automated Deployment and Management of Cloud Applications. In A. Bouguettaya, Q. Z. Sheng, & F. Daniel (Eds.), *Advanced Web Services* (pp. 527–549). New York: Springer. doi:10.1007/978-1-4614-7535-4_22.

Böhme, R., & Reussner, R. (2008). Validation of predictions with measurements. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *4909 LNCS*, 14–18. doi:10.1007/978-3-540-68947-8_3.

Bui, L. T., Essam, D., Abbass, H. A., & Green, D. (2004). Performance analysis of evolutionary multi-objective optimization methods in noisy environments. In *Proceedings of the 8th Asia Pacific symposium on intelligent and evolutionary systems* (pp. 29–39). volume 11.

Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., & Buyya, R. (2011). CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, *41*, 23–50. doi:10.1002/spe.995.

Canfora, G., Penta, M. D., Esposito, R., & Villani, M. L. (2005). An approach for QoS-aware service composition based on genetic algorithms. *Genetic And Evolutionary Computation Conference*, (p. 1069). doi:10.1145/1068009.1068189.

Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F., & Mirandola, R. (2009). Qos-driven runtime adaptation of service oriented architectures. In *Proceedings of the 7th joint meeting of the European software engineering*

conference and the ACM SIGSOFT (pp. 131–140). New York, New York, USA: ACM Press. doi:10.1145/1595696.1595718.

Celik, T., & Tekinerdogan, B. (2013). S-IDE: A tool framework for optimizing deployment architecture of High Level Architecture based simulation systems. *Journal of Systems and Software*, *86*, 2520–2541. doi:10.1016/j.jss.2013.03.013.

Ciavotta, M., Ardagna, D., & Koziolek, A. (2015). Palladio optimization suite: QoS optimization for component-based cloud applications. *9th EAI International Conference on Performance Evaluation Methodologies and Tools, ValueTools 2015*, (pp. 3–4). doi:10.4108/eai.14-12-2015.2262562.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., & Stafford, J. (2010). *Documenting Software Architectures Views and Beyond*. (2nd ed.). Addison-Wesley.

Copil, G., Moldovan, D., Truong, H.-l., & Dustdar, S. (2013). SYBL: An Extensible Language for Controlling Elasticity in Cloud Applications. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing* (pp. 112–119). IEEE. doi:10.1109/CCGrid.2013.42.

Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, *6*, 182–197. doi:10.1109/4235.996017.

Dhingra, M. (2014). *Elasticity in IaaS Cloud, preserving performance SLAs*. Ph.D. thesis Indian Institute of Science.

Dillon, T., Wu, C., & Chang, E. (2010). Cloud Computing: Issues and Challenges. *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, (pp. 27–33). doi:10.1109/AINA.2010.187.

Donoso, Y., & Fabregat, R. (2016). *Multi-Objective Optimization in Computer Networks Using Metaheuristics*. Auerbach Publications.

Durán, F., & Salaün, G. (2015). Robust and Reliable Reconfiguration of Cloud Applications. *Journal of Systems and Software*, (pp. 179–184). doi:10.1016/j.jss.2015.09.020.

Dustdar, S., Guo, Y., Satzger, B., & Truong, H.-L. (2011). Principles of Elastic Processes. *IEEE Internet Computing*, *15*, 66–71. doi:10.1109/MIC.2011.121.

Easterbrook, S., Singer, J., Storey, M.-A., & Damian, D. (2008). Selecting Empirical Methods for Software Engineering Research. *Guide to Advanced Empirical Software Engineering*, (pp. 285–311). doi:10.1007/978-1-84800-044-5_11.

Eiben, A. E., & Smith, J. E. (2003). *Introduction to Evolutionary Computing*. Springer. doi:10.1162/evco.2004.12.2.269. arXiv:9809069v1.

Etemaadi, R., & Chaudron, M. R. V. (2015). New degrees of freedom in metaheuristic optimization of component-based systems architecture: Architecture topology and load balancing. *Science of Computer Programming*, *97*, 366–380. doi:10.1016/j.scico.2014.06.012.

Feliot, P., Bect, J., & Vazquez, E. (2017). A Bayesian approach to constrained single- and multi-objective optimization. *Journal of Global Optimization*, *67*, 97–133. doi:10.1007/s10898-016-0427-3. arXiv:1510.00503.

Fonseca, C. M., Knowles, J. D., Thiele, L., & Zitzler, E. (2005). A tutorial on the performance assessment of stochastic multiobjective optimizers. In *Third International Conference on Evolutionary Multi-Criterion Optimization (EMO 2005)* (p. 240). volume 216.

Franceschelli, D., Ardagna, D., Ciavotta, M., & Di Nitto, E. (2013). Space4cloud: A tool for system performance and costevaluation of cloud systems. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds* (pp. 27–34). ACM.

Franks, G., Al-Omari, T., Woodside, M., Das, O., & Derisavi, S. (2009). Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering*, *35*, 148–161. doi:10.1109/TSE.2008.74.

Frey, S., Fittkau, F., & Hasselbring, W. (2013). Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *International Conference on Software Engineering (ICSE-13)* (pp. 512–521).

Galante, G., & De Bona, L. C. E. (2012). A survey on cloud computing elasticity. *Proceedings - 2012 IEEE/ACM 5th International Conference on Utility and Cloud Computing, UCC 2012*, (pp. 263–270). doi:10.1109/UCC.2012.30.

Gregg, B. (2013). *Systems Performance: Enterprise and the Cloud*. (1st ed.). Prentice Hall.

Gross, D., Shortle, J. F. J., Thompson, J. J. M., & Harris, C. C. M. (2008). *Fundamentals of Queueing Theory*. Wiley-Interscience.

Hadded, L., Charrada, F. B., & Tata, S. (2015). An efficient optimization algorithm of autonomic managers in service-based applications. In C. Debruyne, H. Panetto, R. Meersman, T. Dillon, G. Weichhart, Y. An, & C. A. Ardagna (Eds.), *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (pp. 19–37). Cham: Springer International Publishing volume 9415 of *Lecture Notes in Computer Science*. doi:10.1007/978-3-319-26148-5.

Han, R., Guo, L., Guo, Y., & He, S. (2011). A deployment platform for dynamically scaling applications in the cloud. *Proceedings - 2011 3rd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2011*, (pp. 506–510). doi:10.1109/CloudCom.2011.75.

Harchol-Balter, M. (2013). *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press.

Huang, D., He, B., & Miao, C. (2014). A Survey of Resource Management in Multi-Tier Web Applications. *IEEE Communications Surveys & Tutorials*, *16*, 1574–1590. doi:10.1109/SURV.2014.010814.00060.

Huang, K.-C., & Shen, B.-J. (2015). Service deployment strategies for efficient execution of composite SaaS applications on cloud platform. *Journal of Systems and Software*, *107*, 127–141. doi:10.1016/j.jss.2015.05.050.

Jedlitschka, A., Ciolkowski, M., & Pfahl, D. (2008). Reporting experiments in software engineering. *Guide to Advanced Empirical Software Engineering*, (pp. 201–228). doi:10.1007/978-1-84800-044-5_8.

Jula, A., Sundararajan, E., & Othman, Z. (2014). Cloud computing service composition: A systematic literature review. *Expert Systems with Applications*, *41*, 3809–3824. doi:10.1016/j.eswa.2013.12.017.

Katsaros, G., Menzel, M., Lenk, A., Revelant, J. R., Skipp, R., & Eberhardt, J. (2014). Cloud Application Portability with TOSCA, Chef and Openstack. In *2014 IEEE International Conference on Cloud Engineering* (pp. 295–302). IEEE. doi:10.1109/IC2E.2014.27.

Korte, B., & Vygen, J. (2012). *Combinatorial Optimization* volume 21. Springer-Verlag GmbH. doi:10.1007/978-3-642-24488-9.

Koziolek, A. (2011). *Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes*. Ph.D. thesis Karlsruher Institut fr Technologie. doi:10.5445/KSP/1000032342.

Koziolek, A., Koziolek, H., & Reussner, R. (2011). PerOpteryx: Automated Application of Tactics in Multi-Objective Software Architecture Optimization. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS - QoSA-ISARCS '11* (p. 33). doi:10.1145/2000259.2000267.

Król, D., & Kitowski, J. (2016). Self-scalable services in service oriented software for cost-effective data farming. *Future Generation Computer Systems*, *54*, 1–15. doi:10.1016/j.future.2015.07.003.

Kruskal, W. H., & Wallis, W. A. (1952). Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, *47*, 583–621. doi:10.1080/01621459.1952.10483441.

Lazowska, E. D., Zahorjan, J., Graham, G. S., & Sevcik, K. C. (1984). *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc.

Leymann, F., Fehling, C., Mietzner, R., Nowak, A., & Dustdar, S. (2011). Moving Applications To the Cloud: an Approach Based on Application Model Enrichment. *International Journal of Cooperative Information Systems*, *20*, 307–356. doi:10.1142/S0218843011002250.

Li, J. Z. W., Woodside, M., Chinneck, J., & Litoiu, M. (2011a). CloudOpt: Multi-goal optimization of application deployments across a cloud. *Network and Service Management (CNSM), 2011 7th International Conference on*, (pp. 1–9).

Li, R., Etemaadi, R., Emmerich, M. T. M., & Chaudron, M. R. V. (2011b). An evolutionary multiobjective optimization approach to component-based software architecture design. *2011 IEEE Congress of Evolutionary Computation, CEC 2011*, (pp. 432–439). doi:10.1109/CEC.2011.5949650.

Lloyd, W., Pallickara, S., David, O., Lyon, J., Arabi, M., & Rojas, K. (2013). Performance implications of multi-tier application deployments on Infrastructure-as-a-Service clouds: Towards performance modeling. *Future Generation Computer Systems*, *29*, 1254–1264. doi:10.1016/j.future.2012.12.007.

Mann, H. B., & Whitney, D. R. (1947). On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, *18*, 50–60.

Mann, Z. A. (2017). The Top Eight Misconceptions about NP-Hardness. *Computer*, *50*, 72–79. doi:10.1109/MC.2017.146.

Martínez-Álvarez, A., Restrepo-Calle, F., Vivas Tejuelo, L. A., & Cuenca-Asensi, S. (2013). Fault tolerant embedded systems design by multi-objective optimization. *Expert Systems with Applications*, *40*, 6813–6822. doi:10.1016/j.eswa.2013.06.060.

McGraw, K. O., & Wong, S. P. (1992). A common language effect size statistic. *Psychological Bulletin*, *111*, 361–365. doi:10.1037/0033-2909.111.2.361.

Moldovan, D., Copil, G., Truong, H.-L., & Dustdar, S. (2013). MELA: Monitoring and Analyzing Elasticity of Cloud Services. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science* (pp. 80–87). doi:10.1109/CloudCom.2013.18.

Mostinckx, S., Van Cutsem, T., Timbermont, S., Boix, E. G., Tanter, É., & De Meuter, W. (2009). CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning

algorithms. *Software - Practice and Experience*, *39*, 661–699. doi:10.1002/spe.

Newman, S. (2015). *Building Microservices*. O'Reilly Media, Inc.

Nitto, E. D., & Meilander, D. (2012). Research challenges on engineering service-oriented applications. In *Proceedings of the ICSE 2012 Workshop on European Software Services and Systems Research* (pp. 14–20).

OMG (2015). Unified Modeling Language® (UML®) Version 2.5 Specification. URL: http://www.omg.org/spec/UML/2.5/.

Papazoglou, M., & Georgakopoulos, D. (2003). Service Oriented Computing. *Communications of the ACM*, *46*, 25–28. doi:10.1109/MC.2006.102.

Reussner, R., Becker, S., Burger, E., Happe, J., Hauck, M., Koziolek, A., Koziolek, H., Krogmann, K., & Kuperberg, M. (2011). The Palladio Component Model. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering - WOSP/SIPEW '10* March (p. 257). Karlsruhe. doi:10.5445/IR/1000022503.

Ross, S. M. (2012). *Simulation* volume 33. (5th ed.). Academic Press.

Schwarzkopf, M., Murray, D. G., & Hand, S. (2012). The seven deadly sins of cloud computing research. *HotCloud'12 Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, (pp. 1–1).

Serafini, P. (Ed.) (2012). *Mathematics of Multi Objective Optimization*. Springer.

Sharp, H. (1968). Cardinality of finite topologies. *Journal of Combinatorial Theory*, *5*, 82–86. doi:10.1016/S0021-9800(68)80031-6.

Sun, Y., White, J., Eade, S., & Schmidt, D. C. (2015). ROAR: A QoS-oriented modeling framework for automated cloud resource allocation and optimization. *Journal of Systems and Software*, . doi:10.1016/j.jss.2015.08.006.

Tanković, N., Bogunović, N., Galinac Grbac, T., & Žagar, M. (2015a). Analyzing incoming workload in Cloud business services. In *2015 23rd International Conference on Software, Telecommunications and Computer*

Networks (SoftCOM) (pp. 300–304). IEEE. doi:10.1109/SOFTCOM.2015.7314068.

Tanković, N., Galinac Grbac, T., Truong, H.-l., & Dustdar, S. (2015b). Transforming Vertical Web Applications Into Elastic Cloud Applications. In *International Conference on Cloud Engineering (IC2E 2015)* (pp. 135–144). IEEE. doi:10.1109/IC2E.2015.15.

Teixeira, M., Ribeiro, R., Oliveira, C., & Massa, R. (2015). A quality-driven approach for resources planning in Service-Oriented Architectures. *Expert Systems with Applications*, *42*, 5366–5379. doi:10.1016/j.eswa.2015.02.004.

Thönes, J. (2015). Microservices. *IEEE Software*, *32*, 116–116. doi:10.1109/MS.2015.11.

Tofan, D., Galster, M., Avgeriou, P., & Schuitema, W. (2014). Past and future of software architectural decisions - A systematic mapping study. *Information and Software Technology*, *56*, 850–872. doi:10.1016/j.infsof.2014.03.009.

Trihinas, D., Pallis, G., & Dikaiakos, M. D. (2014). JCatascopia: Monitoring Elastically Adaptive Applications in the Cloud. *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, (pp. 226–235). doi:10.1109/CCGrid.2014.41.

Vaquero, L. M., Rodero-Merino, L., & Buyya, R. (2011). Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, *41*, 45. doi:10.1145/1925861.1925869.

van Vliet, H., & Tang, A. (2016). Decision Making in Software Architecture. *Journal of Systems and Software*, . doi:10.1016/j.jss.2016.01.017.

Wu, L., & Buyya, R. (2010). Service Level Agreement (SLA) in Utility Computing Systems. *IGI Global*, (p. 27). doi:10.4018/978-1-60960-794-4.ch001. arXiv:1010.2881.

Wu, L., Kumar Garg, S., Versteeg, S., & Buyya, R. (2013). SLA-based Resource Provisioning for Hosted Software as a Service Applications in Cloud Computing Environments. *IEEE Transactions on Services Computing*, *7*, 1–1. doi:10.1109/TSC.2013.49.

Wulf, V., Pipek, V., & Won, M. (2008). Component-based tailorability: Enabling highly flexible software applications. *International Journal of Human-Computer Studies*, *66*, 1–22. doi:10.1016/j.ijhcs.2007.08.007.

Xiong, K. X. K., & Perros, H. (2009). Service Performance and Analysis in Cloud Computing. In *2009 Congress on Services - I* (pp. 693–700). IEEE. doi:10.1109/SERVICES-I.2009.121.

Zhao, X., Lin, Q., Chen, J., Wang, X., Yu, J., & Ming, Z. (2016). Optimizing security and quality of service in a Real-time database system using Multi-objective genetic algorithm. *Expert Systems With Applications*, *64*, 11–23. doi:10.1016/j.eswa.2016.07.023.

Zheng, W., Zhu, Q., Natale, M. D., & Vincentelli, A. S. (2007). Definition of Task Allocation and Priority Assignment in Hard Real-Time Distributed Systems. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)* (pp. 161–170). IEEE. doi:10.1109/RTSS.2007.40.

Zitzler, E., Laumanns, M., & Thiele, L. (2001). SPEA2: Improving the Strength Pareto Evolutionary Algorithm. *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems*, (pp. 95–100). doi:10.1.1.28.7571. arXiv:arXiv:1011.1669v3.

Zitzler, E., & Thiele, L. (1999). Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *IEEE Transactions on Evolutionary Computation*, *3*, 257–271.