2015 IEEE International Conference on Cloud Engineering

Transforming Vertical Web Applications Into Elastic Cloud Applications

Nikola Tanković*, Tihana Galinac Grbac†, Hong-Linh Truong‡, Schahram Dustdar‡
*Superius d.o.o., Pula, Croatia
nikola.tankovic@superius.hr
†Faculty of Engineering, University of Rijeka, Croatia
tihana.galinac@riteh.hr
‡Vienna University of Technology Vienna, Austria
{truong, dustdar}@dsg.tuwien.ac.at

***Abstract*—There exists a huge amount of vertical applications that are developed for isolated computing environments. Due to increasing demand for additional resources there is a clear need to adapt these applications to the distributed environments. However, this is not an easy task and numerous variants are possible. Moreover, in this transition a new quality requirements become important, such as application elasticity. Application elasticity has to be built into a software system to enable smooth cost optimization at the run-time.**

In this paper, we provide a framework for evaluating different transformation variants of vertical Java EE multi-tiered applications into elastic cloud applications. With support of this framework the software developer is guided how to transform its application achieving optimal elasticity strategy. The framework is evaluated on slicing and evaluating elasticity of existing SaaS multi-tiered Java application used in Croatian market.

Keywords*—*cloud computing; web application elasticity; web service deployment;

I. Introduction

Software engineering discipline aims to develop methods and tools to produce quality software. Elasticity has been just recently recognized as one of the important quality attributes for the cloud software systems [1]. It is a software attribute providing ability to efficiently adapt to dynamically scratched resources, coined by cloud provides that in 'as a service' paradigms offer elastic on-demand resources based on the 'pay-as-you-go' (PAYG) concept. In such an environment the term *elastic* imposes strong requirements on software.

One of the challenges of cloud is to provide dynamic allocation of resources according to workload changes and within Service Level Agreement (SLA) levels but with optimized resource costs. To maximize PAYG benefits, elasticity attribute should be closely related to costs [2]. It has been widely discussed its definition in terms of software quality attributes and how to incorporate this new quality perspective into existing list of well known software quality attributes.

We address the elasticity from the software engineering perspective and propose a software design solution to implement this attribute into a software system. We hypothesize that the structure of software distribution across the logical nodes may influence software system elasticity. Distributed system may be easier to expand and scale then vertical system - a tightly coupled application developed to work on single machine only. Especially, from the performance and resource utilization costs perspective, the way how the application is distributed may provide some benefits for easier dynamic resource scaling.

Our approach is to provide the necessary insights for finding an optimal structure in distributing an existing vertical (monolithic) application. The design of optimal software structure with regards to elasticity is guided by the framework provided in this paper.

The main contributions of our paper are the following:

- an automated mechanism for observing performance of different distributed slices of an existing vertical applications and identifying bottlenecks;
- a decision support system for structure design in transformation process of vertical applications.

In this paper we present a framework that guides developers aiming to transform their vertical applications into elastic cloud applications providing run-time cost optimization. The framework is implemented through Java APIs which are used with existing application to control its distribution. Our framework targets multi-tier web applications implemented by Java EE specification. Only applications without server-side presentation tier (e.g. Java Server Faces) are supported; exposing their functionality through web service interface. Such server-side applications are typically used in modern SaaS software where presentation tier is located on client side. Instances of such applications include modern web and mobile applications or applications with no presentation tier used to expose business APIs. Additionally, to facilitate distribution, only stateless Java Beans are analyzed, which are common in described applications for easier scaling and management.

Proposed framework is thus intended for smaller business applications that require scaling and are planned to be rewritten for distributed architecture such as cloud. By using the framework, application developer is provided with functionality to test different slicing possibilities together with information regarding total cost and performance of each solution. Cost is regarded as overall leased hardware cost at a given, and performance is evaluated as maximal service rate achieved in each slicing configuration. Loosely coupled applications are better suited and will produce better evaluation results with minimum modification.

978-1-4799-8218-9/15 $31.00 © 2015 IEEE
DOI 10.1109/IC2E.2015.15

IEEE computer society

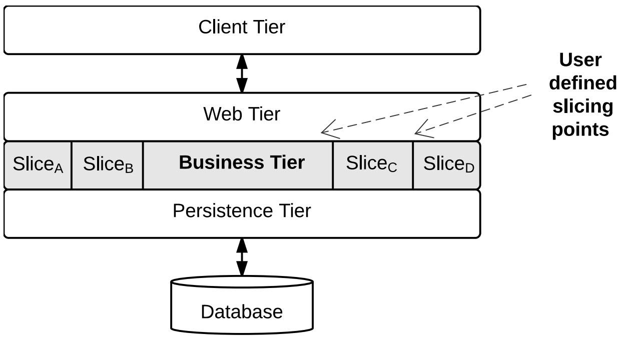


Fig. 1: Slicing occurs at middle Business Tier

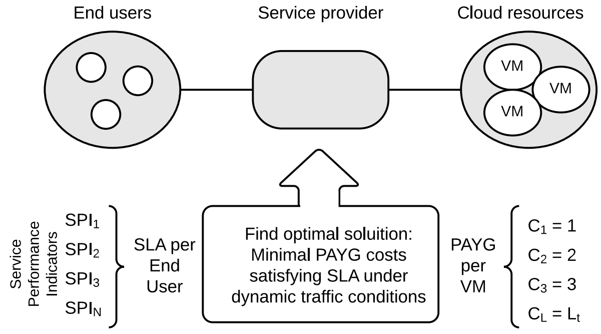


Fig. 2: Schematic of elastic system

From software development perspective, decoupling slices from existing applications has thoroughly been studied [3], so some effort should be expected from application developers in order to fully utilize this framework. Knowledge about application architecture and choosing viable set of slicing points is a prerequisite, and not in scope of this paper.

The remainder of this paper is structured as follows: The proposed framework is described in Sect. II. In Sect. III we provide implementation details for the framework. In Sect. IV we evaluate proposed framework on existing monolithic application. Related work is given in Sect. V. Finally, concluding remarks and future study areas are presented in sect. VI.

II. Framework

We propose a framework that facilitates software architectural decisions on how to slice vertical multi-tiered web applications into separated web services that are distributed over network to provide effective scaling. Our framework mainly concentrates on slicing at application tier (also referred to as business or domain tier), thus targeting pieces of domain business logic (Fig. 1). It does not address client tier or persistence (database) tier. We believe that the main point of slicing should occur at domain logic tier distributing the application to standalone services. This makes our framework adequate for web applications without thin or no web tier at all, exposing web services directly to client tier. Once the optimal slicing is chosen at business tier, manual redistribution of data in persistence tier is required. Techniques like data replication and sharding are typically used. Slicing at persistence tier requires careful re-design and migration plans which is not in the scope of this work. We leave the starting decision for picking the possible slicing points to the developers and concentrate on evaluating elasticity concerns of every generated deployment option based on these slices. Overall service latency can drastically differentiate with every design change at persistence tier. For such cases, our tool can also facilitate regression testing.

We decompose our solution into four tasks:

- slicing of vertical application into self contained separated slices that could be deployed as web services on separate VMs,
- generating a number of deployment variants as possible deployment solutions; by deployment variant we refer to placement decision for each service and its invocation type (local or remote via SOAP),
- slice assessment mechanism in terms of identification of bottleneck slices, resource demands and scaling trends with help of operational profile of existing vertical application,
- performance and cost evaluation for all variants and its comparison with respect to cloud provider constraints.

In order to facilitate the problem of application slicing like building and packaging all the different slices and determining dependencies for each slice, this framework uses an alternate approach. Actual software artifacts are not physically sliced, but rather logically. Each slice contains whole application package where unnecessary parts belonging outside of the current slice, are never being accessed. The purpose of this framework is exploration and evaluation of different distribution variants so this solution proved practical and fast.

A. Framework notation

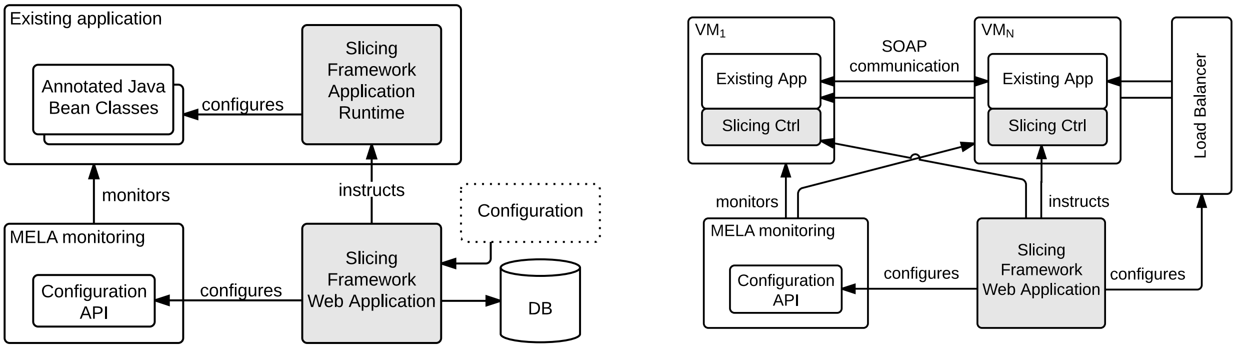
To describe the framework we use a high level schematic of the considered system given in Fig. 2. The user (customer) on the left-hand side of the figure is sending requests for processing of certain application to the cloud on the right-hand side of the figure, and the task of the service provider at the center of the figure is to organize the usage of cloud resources for processing. The goal is to do that at lowest possible cost, but meeting the quality of service requirements agreed with users (customers) in a service level agreement (SLA).

Vertical application that is processed for each process request can be divided in several services. These services are independent of each other and can be processes concurrently in the cloud if resources are available.

B. Simulation

On the other side, there are several virtual machines in the cloud, which can be used to process the application or its services. We denote these virtual machines by VM_m, where $m = 1, \ldots, M$, and M is the maximal number of virtual machines that can be used in the cloud.

The cost of virtual machines is agreed by a pay as you go (PAYG) contract between the service provider and the cloud



(a) Development-time view

(b) Run-time view

Fig. 3: Overview of Slicing framework

provider. That means that the service provider can use virtual machines at different load levels. For each level there is a fixed price per hour. If we denote the load levels of each virtual machine by $1, 2, \ldots, L$, where L is the number of levels, then the cost for lth level is denoted by C_l. Of course, higher loads are more expensive, so that

$$C_1 < C_2 < \ldots < C_L.$$

Note that we are assuming the same prices per level for all virtual machines VM_m. For this exposition we introduce the 0th load level to be the load level with no load at all, that is, the virtual machine at 0th load level is not used. We set $C_0 = 0$ as the cost of zero load is zero.

By the contract, the virtual machines are payed per hour, which means that we may specify freely the number of machines in use and their load level which is payed according to fixed costs C_l. In this way we can adjust to different intensity of incoming requests during the day. We form a cost matrix $\mathbf{C}$ with M rows representing virtual machines and 24 columns representing the hours in a day. The entry c_{ij} in the ith row and jth column in the cost matrix $\mathbf{C}$ is the cost of the load level at which the virtual machine VM_i is operating in the jth hour of the day (i.e., between $j-1$ o'clock and j o'clock). The total cost of operation C_{total} for a day in which the cloud resources are used according to the cost matrix $\mathbf{C}$ is simply the sum of all entries in $\mathbf{C}$, i.e.,

$$C_{\text{tot}} = \sum_{i=1}^{M} \sum_{j=1}^{24} c_{ij}. \tag{1}$$

From the cost matrix we can easily see how each VM_m is used during the day by looking at mth row of $\mathbf{C}$. The columns, on the other hand, tell us the load levels of all virtual machines during each hour in a day. Note that $c_{ij} = 0$ means that ith virtual machine is not in use between $j-1$ o'clock and j o'clock.

The services of the application can be processed at different virtual machines. The scheme saying which services are processed at the same or different virtual machines is called the service deployment strategy. Let $\mathcal{S}$ denote the set of services. The service deployment strategy is just a partition of this set into disjoint subsets. Each subset of services is processed at the same virtual machine, and services in different subsets of the partition are processed at different virtual machines.

The service deployment strategy is fixed per each evaluation run, and can not be changed during the operation. In this paper we investigate how different deployment strategies combined with the usage of virtual machines based on the known incoming process request distribution during the day influence the total costs of operation assuming the requirements of SLA are met. This kind of analysis is useful to the service providers when developing the application because they can plan the division into services according to the results of such analysis.

In order to perform slice assessment with performance and cost evaluation, we need to simulate the operation of the service provider during a typical day. To achieve that, information about the distribution of incoming process requests is required. For each simulation we fix a service deployment strategy, say

$$\mathcal{S} = \coprod_{k=1}^{d} \mathcal{S}_k,$$

where $\mathcal{S}_k$ are the disjoint subsets of services forming a partition of the set $\mathcal{S}$ of all services. We also fix a cost matrix $\mathbf{C}$, which is, in other words, the plan of usage of virtual machines during the day. The total cost of such operation is just the value C_{tot}, obtained from the cost matrix in formula 1. However, the question is whether such operation will satisfy the quality of service requirements specified in the service level agreement between the consumer and the service provider.

These quality of service requirements could be for example an upper bound on response time or total processing time of each request, or a lower bound on the percentage of requests that are not answered due to congestion. These are studied using the simulated requests. To make this work, one needs to know the processing time of every subset $\mathcal{S}_k$, and use the queuing theory to obtain simulated waiting times and virtual machine load. Using these parameters, the requirements can be checked. Such simulated operation for a day should be simulated many times to see that certain service deployment strategy with certain cost matrix would satisfy the requirements in sufficiently high percentage of times (specific values, of course, depend on the service provider's policy).

Finally, among all possible service deployment strategies, one may find the one which meets the quality of service requirements at lowest possible cost. This can help application developers to plan the division of an application into services in such a way that the operation costs can be minimized. On the other hand, if the software deployment strategy is already fixed, our model can be used to find the way of using the cloud resources so that the user requirements are met at the lowest possible cost.

III. Implementation

We developed proposed framework as an application transformation developer kit for Java web applications. It provides APIs to indicate which existing code fragments should be transformed to enable remote invocation. All standard Java Bean[1] classes are valid candidates for such transformation. This development kit serves application developers to test their existing application behavior in a distributed deployment as an elastic application - it is primarily used to obtain the best cost effective scaling method based on the future needs.

Fig. 3a displays the architecture overview of slicing framework solution. There are two operating parts of slicing framework: a controller part that is integrated within existing application and a central self-contained web application with APIs for configuration and coordination of controllers. This is displayed in Fig. 3b where Slicing Web Application configures each controller in order to achieve desired deployment. We base our implementation on the monitoring and analysis tool MELA [4] collecting and aggregating selected metrics for every running VM instance. MELA monitoring and a load balancer are automatically configured according to current deployment.

A. *Slicing of vertical application*

Implementation is based on the Java EE platform beginning from version 6. The J2EE platform is a Java environment that provides a run-time infrastructure for hosting applications and a set of Java extension APIs to build distributed applications.

Slicing API transforms Java EE Bean class from existing application into separated self contained web service by adding adequate code. Such code directs the framework to provide necessary communication capabilities outside its logical node (e.g. virtual machine). We illustrate our approach on example of *InvoiceService.java* Java EE Bean class. To make this class invocable as a web service outside of current Java Virtual Machine (JVM) one should annotate it as *@WebService* annotation which instructs the Enterprise Java Beans (EJB) container to automatically construct the necessary WSDL file and provide this class as an SOAP web service. This example is presented in Fig. 4.

```
// IInvoiceService.java

@ElasticService
@WebService(targetNamespace=...)
public interface IInvoiceService {
  ...
}
```

Fig. 4: Annotations required for class interface

Our framework provides some additional annotations: *@ElasticServiceProvided* and *@ElasticService*. *@ElasticService* annotation signals the framework that current Java Bean interface should be considered as an elastic service such that framework can experiment with its deployment. For such marked service, framework provides an additional configuration to specify deployment variants to be used across experiments.

To invoke a service as a dependency from another class, besides the standard *@Inject* annotation, provided by Java EE specification to instruct EJB container a need to instantiate or delegate an instance in run-time, our implementation also provides additional *@ElasticServiceProvided* annotation so we can override the default EJB procedure and provide class instance based on current selected deployment configuration. This is noted in Fig. 5. Our framework will instantiate a local class or automatically create a remote client communicating with remote service, depending on current deployment configuration. Dependent services which are annotated are used the same way whether they are residing locally in same JVM or remote on another JVM/VM. E.g. in *InvoiceService.java* both *FiscalizationService* and *DataService* are marked elastic, so our framework has the possibility to provide local service instances from within same JVM, or remote services residing on different VM. We can observe that in *createInvoice* method code dealing with invocation of the FiscalizationService remains the same.

All possible deployment variants are obtained from set of services which developer has chosen to annotate with *@ElasticService*. In Sect. II-A we denoted this set as $\mathcal{S} = \{S_1, S_2, ...S_n\}, n = |\mathcal{S}|$. Next, we arrange these services in disjoint sets. We define such set as Elastic Group (EG): a set of services that are to be deployed and scale together. Only one EG is deployed on each VM, meaning that deployment variant - D, is one possible division of $\mathcal{S}$ to EG sets. A set of all possible deployment variants, denoted as $\mathcal{D}$, correlates to all possible combinations of forming EG sets.

Even a slightly complex Java application yields very large amount of possible deployment variants. To obtain the total number of deployments we need to count all possible partitions of set $\mathcal{S}$. Formula 2 uses *Stirling number of second kind* [5] to calculate this.

$$|\mathcal{D}| = \sum_{k=1}^{n} \frac{1}{k!} \sum_{j=0}^{k} (-1)^j \binom{k}{j} (k-j)^n, n = |\mathcal{S}| \,. \tag{2}$$

To narrow down the number of solutions, we provide further configurations capabilities to denote which services should always remain inside same EGs. These could be services that

[1] JavaBeans specification is available at http://www.oracle.com /technetwork/java/javase/documentation/spec-136004.html

```
// IInvoiceService.java
@Stateless
@WebService(...)
public class InvoiceService implements IInvoiceService {

   // service dependencies
   @Inject @ElasticServiceProvided IFiscalizationService fService;
   @Inject @ElasticServiceProvided IDataService dataService;

   public CreateInvoiceResponse createInvoice(CreateInvoiceRequest req) {
      // calling other service
      fiscalizationService.fiscalizeInvoice(request);
   }
}
```

Fig. 5: Annotations required for class implementation and dependencies

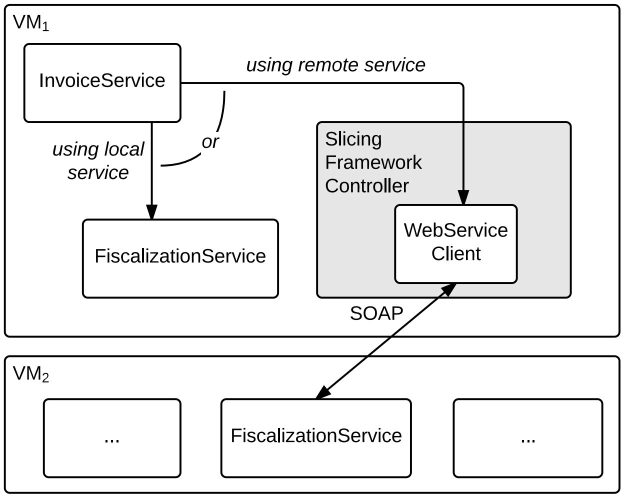


Fig. 6: Framework can use local or remote service

developers decides to be tightly connected and is reasonable for optimal performance to keep them together for lowest possible invocation latency.

Scaling is thus achieved simply by replicating VMs. EGs also define an important implication on dependency resolving. If S_i invokes S_j then this invocation can be: a) local, if S_i and S_j reside in same EG group, thus same VM, or b) remote via SOAP, otherwise. Slicing framework controllers assert that correct invocation is being made, as shown in Fig. 6.

B. Variant generator

We developed a variant generator as REST service with Java API for generating a number of deployment variants. Each variant is obtained by number of different grouping combinations of different slices, obtained as explained in subsection III-A. Each variant is deployed to minimal number of VMs (one per group) and then MELA Java APIs are applied to monitor and analyze elasticity criteria. To test elasticity, SYBL is used to define elasticity triggers on when to perform certain scaling actions. SYBL functionality is explained at [6]. We created a SYBL configuration to be used across testing where we defined a cost-effective scaling based on VM load thresholds. There are two basic scaling actions for every EG: (1) *scaleIn* for releasing a VM when load drops below defined level, and (2) *scaleOut* for running another VM instance when load is above upper threshold. For this research, we applied this simple SYBL rules across all tests, developers can experiment with different SYBL configurations for each variant by rerunning tests with different SYBL configurations. For future version of framework, we plan semi-automatic support for such experiments. Variant generator is accessed with HTML user interface provided by Slicing Framework Web Application. Same interface is used to schedule and run tests.

C. Deployment variant performance evaluation

Implementation of our framework uses MELA to capture performance metrics (response time, throughput, CPU usage) across different service levels (down to specific VMs) so that many aspects of scaling can be evaluated accordingly. Also, by using MELA measurements one can easily balance between VM local and application global performance and validate each solution thoroughly. From performance observations of global application performances achieved in different deployment variants and local slice performances achieved by each slice in particular deployment variant one can identify the most effective set of elastic groups of application slices (EGs). Different elastic groups or combinations of slices can be evaluated separately.

For the evaluation purpose we developed a simulator API in Python programming language that generates Poission distribution of input requests and it is capable to send heterogeneous requests with different probabilities. The request rates are defined by developers and should be based on the monitoring results from current vertical application in use. Ratios for different transactions should also be obtained accordingly.

IV. Experiment

We demonstrate the use of our framework on a prototype built as viable subset of an actual Java EE application. This SaaS cloud application is used by a large base of customers

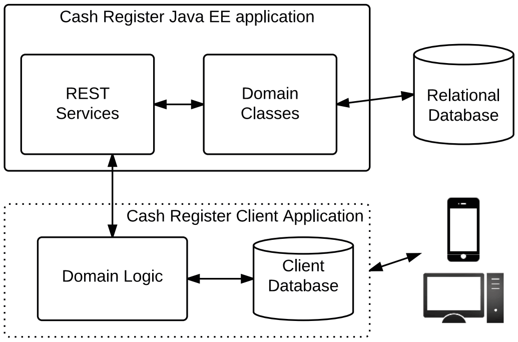


Fig. 7: Architectural overview of Cash Register application

in Croatia to issue invoices in retail. We will refer to it as Cash Register Solution (CRS). Currently, the server-side of CRS is a monolithic Java EE application (Fig. 7). We apply the framework to CRS application to make certain classes available for remote invocation and distribute the system across different VMs. CRS is built by Superius, a small company located in Croatia which observe a need to make its CRS application service-based and distributed for better control of overall service, better elasticity control and most importantly to share underlying services in CRS application and provide integration with other applications in their SaaS portfolio. Basically, Superius is targeting a more service-oriented approach in building SaaS [7]. They provide CRS as a SaaS service on Cloud resources for dynamic amount of clients. Currently, CRS application is not elastic and runs on a set of dedicated servers optimized to satisfy demands in peak hours. Off-peak periods reveals under-utilization of resources yielding unnecessary costs.

It is a challenge for the small software development company to transform its vertical applications, mainly because of lack of knowledge and experience in building a complex distributed and elastic system. Quite often these companies are set on a path of trial and errors. By using the proposed framework, developers can get quicker and better insights into performance and elasticity implications of distributing their applications in different ways.

A. Test-bed setup

Testing platform is set up to resemble Amazon business model, it has a PAYG service issuing VMs at hourly cost. The elasticity strategy must be implemented in conformance with existing SLAs for CR application. These include Service Level Objectives regarding maximum response time and overall service availability. Upper bounds for response time are specified with two parameters: a percentile $\gamma\%$ and T_r, meaning that $\gamma\%$ of the time customer will receive response in less than T_r [8]. These SLA criteria have been configured and monitored by MELA.

All the results are gathered by using a test-bed setup over a set of VMs running on OpenStack[2] platform. OpenStack is hosted on dedicated server with Intel Xeon E31230 Quad-core processor and 8GB of RAM. These are dedicated servers

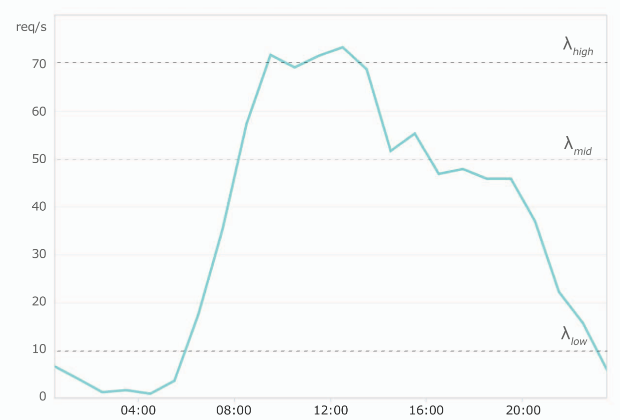


Fig. 8: Real production request rate distribution for Cash Register application and request rate classes

similar to ones used to deploy production version of CRS application. HAProxy[3] has been used as a configurable load balancer. Each deployment variant that is tested means automatic alteration of HAProxy configuration to enable desired service distribution.

B. Current application analysis

We measured current average daily request arrival rates from our production system, displayed at Fig. 8. Arrival rate is significantly higher around normal working hours in retail business, reaches its maximum around noon, and decreases towards evening. This kind of load is a common example for reducing costs by introducing elasticity capabilities. Based on this input and available test-bed hardware, we classify arrival rates λ_x at three classes modeled similar to production usage:

$$\lambda_{low} = 10req/s, \lambda_{mid} = 50req/s, \lambda_{high} = 70req/s$$

All tests are conducted by using a set of request types modeled after analyzing current production load: *createInvoice* request is invoked with a probability of $p_1 = 0.217$, *getServiceStatus* request with a probability of $p_2 = 0.52$, *getData* request with a probability of $p_3 = 0.135$, and *listInvoices* request with a probability of $p_4 = 0.128$.

C. Slicing the Cash Register application

We have chosen these four services to be tested with remote usage on different deployments:

- *InvoiceService (IS)* - main service for issuing retail invoices. This service receives the request for invoice with selected products and customer information and is used to generate all the calculations and persist invoice to database.
- *FiscalizationService (FS)* - Croatian laws regulate that invoices issued in retail with cash as payment type should be sent to a central *fiscalisation* web service provided by government where every invoice receives a unique identifier. FS is a delegate for this process.

[2] OpenStack cloud platform - available at http://www.openstack.org/

[3] HAProxy - available at http://www.haproxy.org/

TABLE I: Deployment variants used in testing.

Variant	EG_1	EG_2	EG_3	EG_4
D_1	IS, FS, DS, SS	-	-	-
D_2	IS	FS, DS, SS	-	-
D_3	IS	DS	FS, SS	-
D_4	IS, FS	DS, SS	-	-
D_5	IS	FS	DS	SS

- *StatusService (SS)* - this service is used by Android and PC client tier to check various critical operational data such as invoice numbers, time synchronization with server time, and to checking service and network availability.
- *DataService (DS)* - this service is also used by the clients to gather necessary data for creating an invoice. This service provides all the customer and product information.

D. Deployment variants

By using these annotated classes, framework generated 15 possible deployment variants. For brevity, we will further examine five most interesting and characteristic variants to state the importance of careful distribution and deployment decisions. Fig. 9 shows the dependency graph between these four services, and Table I gives the formed groups of services.

E. Results

For each variant, we configured MELA to monitor end-to-end response times for three incoming request rates values: *low*, *mid* and *high* traffic. We also measured overall performance expressed as average service rate across testing period, measures in number of successful responses per time unit. Results are given in Table II. For each arrival rate class, a number of VMs running at that time is given and total costs for 24 hours of operation C_{tot} by applying the cost matrix introduced in Sect. II-B. We also measured maximal service rate each service variant can provide under selected number of VMs. Costs are modeled following Amazon EC2 pricing chart[4].

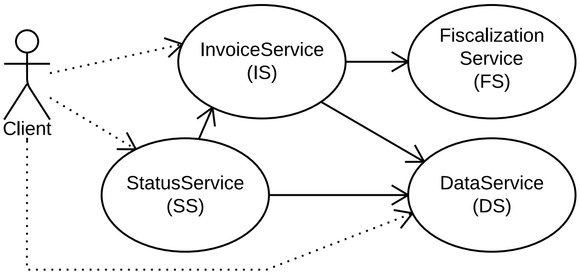


Fig. 9: Dependency graph between selected services

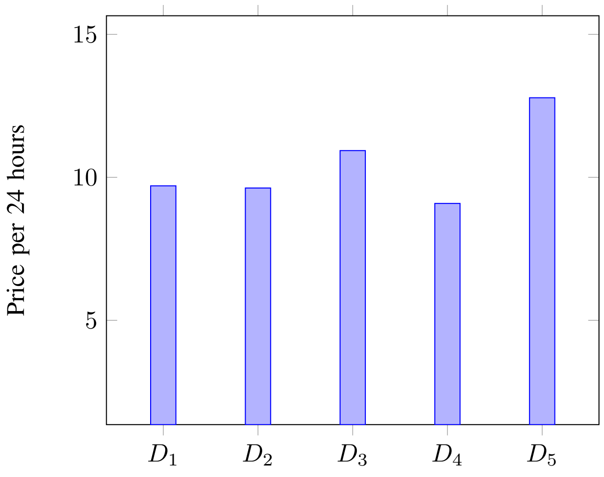


Fig. 10: Comparison of price for each 24-hour deployment variant

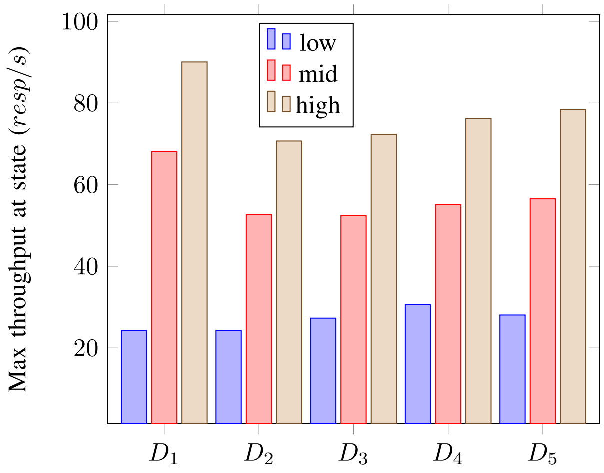


Fig. 11: Comparison of performance in each deployment variant

Results indicate that total costs for Cash Register migration to distributed deployment can be up to 32% higher (Fig. 10) in most expensive deployment variant D_5 where each service is isolated, but we observe no improvement in performance and quality (Fig. 11). This is primarily not because of the overhead that serialization and communication over web service protocols are bringing, but mainly because of over-provisioning meaning that EGs used a small percentage of resources given. However, if a system is able to change deployment variant in run-time at lower traffics these costs can be significantly reduced. A new reasoner for elasticity is thus required, one that could apply different deployment variants in order to further reduce operational costs. By using SYBL we could manually define *switchDeployment* action to change composition in run-time, or *isolateService* and *consolidateServices* to divide or group services.

We observe that for CR application, the bottleneck is the *InvoiceService*, it is the only service that required significant

[4] Amazon EC2 pricing chart is available at http://aws.amazon.com/ec2/pricing/

TABLE II: Overall service results for different deployment variants

Variant D_i	Arrival rate (req/s)	# of VM (EG_1)	# of VM (EG_2)	# of VM (EG_3)	# of VM (EG_4)	Max Service rate (response/s)	Avg.resp.time	Cost/h	Total cost per 24h - C_{tot} ($)
D_1	low	1	-	-	-	24.25	384.72	0.154	9.702
	mid	3	-	-	-	68.06	118.37	0.462	
	high	4	-	-	-	90.04	101.62	0.616	
D_2	low	1	1	-	-	24.28	374.12	0.231	9.625
	mid	2	1	-	-	52.66	187.52	0.385	
	high	3	2	-	-	70.68	110.12	0.616	
D_3	low	1	1	1	-	27.29	310.21	0.308	10.934
	mid	2	1	1	-	52.43	165.48	0.462	
	high	3	1	1	-	72.33	105.32	0.616	
D_4	low	1	1	-	-	30.61	222.02	0.231	9.086
	mid	2	1	-	-	55.06	174.90	0.385	
	high	3	1	-	-	76.16	123.75	0.539	
D_5	low	1	1	1	1	31.06	168.20	0.385	12.782
	mid	2	1	1	1	56.52	163.36	0.539	
	high	3	1	1	1	78.39	126.77	0.693	

scaling under higher request rates. Other elastic groups containing rest of the services ware underutilized thus should be considered for consolidation, in a manner similar to consolidation of VMs across underlying servers [9]. Deployment variants D_2 and D_4 are such examples, and we can observe cost savings because of this. Isolation of these services should be made if there are some special security conditions or if they are shared between many applications.

Based on obtained data-set, system provider for CRS application can reason on adequate deployment strategy. Developers should decide on their own metrics and apply them to MELA monitoring and SYBL elasticity control based on their set of SLA objectives and obtain the best variant for their needs. Despite the cost, there are many deciding factors for choosing a more distributed environment, like ability to choose better VM characteristics for certain EG, per-EG VM optimization, enforced security policies, special caching strategies or elasticity rules. For this experiment, if we disregard such factors, based on evaluating all 15 variants, and using a low-cost priority SYBL elasticity strategy, D_4 proves to be the best solution with a cost savings of 6.35% in comparison to naïve approach D_1. This is due to fine granularity achieved by specifying elasticity criteria on per-service group level.

We also noticed an opportunity to further reduce costs by executing certain Cash Register services in parallel. This way we could reduce the latency and possibly increase throughput. Also, a part of some service requests could be conducted asynchronously when prices of VMs are lower, e.g. by using Amazon Spot Instances[5]. We are researching different ways to extend our framework to enable experiment with such deployment variants.

V. Related work

Migration of software to cloud has been thoroughly studied. A tertiary study in form of Systematic Literature Review conducted by Jamshidi et al. [10] reveals that elasticity has been enabled only if original application was engineered for load balancing between the resources. These researches are mainly studying feasibility, requirements, costs and migration strategies. Andrikopoulos et al. [11] mention elasticity as a cross-cutting concern in migration, opening many research challenges around it. Elasticity has been only enabled on per tier level and whole application level. Our research enables experimenting with a fine-grained per service elasticity strategies.

Elasticity in cloud is subject of many recent research. From proposals of individual elasticity controllers or frameworks like in [12], [13] to more advanced solutions like PaaS. García et al. [14] proposed a PaaS solution to dynamically provision cloud resources with regards to predefined set of SLA metrics. Truong et al. [15] define an integrated PaaS solution for the whole life-cycle of elastic systems, from design to testing and monitoring based on multi-dimensional elasticity [1].

Industry offers several PaaS solutions for building elastic applications. Amazon Elastic Beanstalk [16] is distinguishable for providing a wide variety of programming languages and deployment servers like Apache[6], IIS[7] or Nginx[8] that are already familiar to developers of conventional applications. There are also research proposals [17], [6] that combine elasticity parameters within deployment descriptors like *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [18], a recently initiated standardization effort from OASIS.

Schulte et al. [2] state the importance of cost reduction in elasticity strategies. Han et al. [19] also take cost and QoS as important criteria for elasticity of multi-tier applications. They provide means to scale such application only at bottleneck tiers thus reducing amount of over-provisioned resources. They also differentiate different type of requests and their effects on elasticity. We apply similar efforts to specifically analyze only business tier and slice it further to better pinpoint bottlenecks within it. An extensive survey of different approaches to cost-aware scaling specialized to multi-tier applications has been conducted in [20].

[5] Amazon Spot Instances, available at http://aws.amazon.com/ec2/purchasing-options/spot-instances/

[6] Apache Web Server - http://httpd.apache.org/

[7] Microsoft IIS - http://www.iis.net/

[8] Nginx - http://nginx.org/en/

In the area of experimenting with different deployment options and their effect on performance, Lloyd et al. [21], [22] examine how different components placed across IaaS service can affect performance. Gómez Sáez et al. [23] also analyze how to distribute different application layers with a special devotion to persistence layer. Our research focuses on slicing the application component itself to smaller parts, and then deals with evaluating different distribution and deployment options with regards to elasticity. Papaioannou et al. [24] took a different approach by developing an architecture for collecting long-term histories of different distributed application deployments along their life-cycle. Such way, an insight into which deployment options performed best is also given, but only compared within set of previously deployed variants.

Kaviani et al. [25] developed a framework for splitting Java OLTP applications within hybrid cloud. Their research is also focused on splitting the application across tier, rather than between tier, but also expands to database tier as well. They focus on optimal application division between public and on-premise deployment based on average execution time and overall cost. Our framework provides slicing into individual service groups which are evaluated against different elasticity criteria set by user within MELA. Since our framework is used as decision support, we designed it to require minimal change to original application code.

VI. Conclusions and Future Work

Elasticity is a term widely discussed recently in area of cloud computing introduced with 'Pay-as-you-go' concept of providing cloud resources. For service providers, this concept enables significant cost reduction. However, there may be a number of different variants on how to use PAYG elastic resources in cloud environment.

In this paper we investigate elasticity as a software attribute in ability to cost effectively adapt using dynamically allocated resources. We show that the software distribution across logical nodes in the cloud environment, so called software structure, may influence software system elasticity. There may be number of different possible software structures deployable in cloud. Some software structures are easier (e.g. cost effective) to scale than others under the specific cloud provider's PAYG constraints. Therefore, we propose a framework that should be used by service providers that want to transform their vertical applications into distributed applications and use PAYG concept to reduce costs.

We demonstrate on existing cloud application example that deployment variants significantly influence cost and quality of cloud service. The maximum cost difference between deployment variants for our Cash Register application was 32% based on 24 hour usage, and we managed to find a variant which was 6% cheaper from simple naïve approach. Careful design can provide benefits for service providers. With slicing we achieved that request peaks for one service do not necessarily affect others, as we isolate executions on different hardware. This also enables further optimization by configuring each service group based on different criteria like request types (high CPU, or high I/O etc.). The downside is in extra I/O cost for communication between services when such communication is required. Such costs should be addressed by carefully choosing slicing points. We see further improvements in this area by analyzing software structure and proposing adequate slicing points in advance.

This work demonstrated slicing functionality on a smaller application. Feasibility of this approach on a larger scale applications from different domain should be examined. To scale our approach on these applications we will apply modeling on software structure and software network level.

Future work will also focus on distributing application by enabling asynchronous and parallel service execution and on developing optimization algorithms that would be useful as decision support for application deployment. Furthermore, these algorithms may be implemented as separated function offered to service providers that would enable optimizing their deployment variant for particular cloud at run-time, and moreover, among different cloud providers as well.

Acknowledgment

The work presented in this paper is supported by COST action 1304 Autonomous Control for a Reliable Internet of Services (ACROSS) and the research grant 13.09.2.2.16. from the University of Rijeka, Croatia.

References

[1] Schahram Dustdar, Yike Guo, Benjamin Satzger, and Hong-Linh Truong. Principles of Elastic Processes. *IEEE Internet Computing*, 15(5):66–71, September 2011.

[2] Stefan Schulte, Dieter Schuller, Philipp Hoenisch, Urlich Lampe, Ralf Steinmetz, and Schahram Dustdar. Cost-driven Optimization of Cloud Resource Allocation for Elastic Processes. *International Journal of Cloud Computing*, 1(2):1–14, 2013.

[3] Frank Tip. A Survey of Program Slicing Techniques. *Journal of programming languages*, 5399:1–65, July 1994.

[4] Daniel Moldovan, Georgiana Copil, Hong Linh Truong, and Schahram Dustdar. MELA: monitoring and analyzing elasticity of cloud services. In *IEEE 5th International Conference on Cloud Computing Technology and Science, CloudCom 2013, Bristol, United Kingdom, December 2-5, 2013, Volume 1*, pages 80–87, 2013.

[5] Henry Sharp. Cardinality of finite topologies. *Journal of Combinatorial Theory*, 5(1):82–86, July 1968.

[6] Georgiana Copil, Daniel Moldovan, Hong-linh Truong, and Schahram Dustdar. SYBL: An Extensible Language for Controlling Elasticity in Cloud Applications. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 112–119. IEEE, May 2013.

[7] WeiTek Tsai, XiaoYing Bai, and Yu Huang. Software-as-a-service (SaaS): perspectives and challenges. *Science China Information Sciences*, 57(5):1–15, March 2014.

[8] Kaiqi Xiong and Harry Perros. Service Performance and Analysis in Cloud Computing. In *2009 Congress on Services - I*, pages 693–700. IEEE, July 2009.

[9] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, April 2010.

[10] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. Cloud Migration Research: A Systematic Review. *IEEE Transactions on Cloud Computing*, 1(2):142–157, 2013.

[11] Vasilios Andrikopoulos, Tobias Binz, Frank Leymann, and Steve Strauch. How to adapt applications for the Cloud environment. *Computing*, 95(6):493–535, December 2012.

[12] Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45, January 2011.

[13] Pankaj Deep Kaur and Inderveer Chana. A resource elasticity framework for QoS-aware execution of cloud applications. *Future Generation Computer Systems*, 37:14–25, July 2014.

[14] Andrés García García, Ignacio Blanquer Espert, and Vicente Hernández García. SLA-driven dynamic cloud resource management. *Future Generation Computer Systems*, 31:1–11, February 2014.

[15] HL Hong-linh Truong, Schahram Dustdar, Georgiana Copil, Alessio Gambi, Waldemar Hummer, Duc-hung Le, and Daniel Moldovan. CoMoTA Platform-as-a-Service for Elasticity in the Cloud. *2014 IEEE International Conference on Cloud Engineering*, (iv), 2014.

[16] Amazon AWS Elastic Beanstalk. http://aws.amazon.com/elasticbeanstalk/.

[17] Rui Han, Moustafa M. Ghanem, and Yike Guo. Elastic-TOSCA: Supporting Elasticity of Cloud Application in TOSCA. In *CLOUD COMPUTING 2013, The Fourth International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 93–100, May 2013.

[18] OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA). http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca, October 2011.

[19] Rui Han, Moustafa M. Ghanem, Li Guo, Yike Guo, and Michelle Osmond. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Generation Computer Systems*, 32:82–98, March 2014.

[20] Dong Huang, Bingsheng He, and Chunyan Miao. A Survey of Resource Management in Multi-Tier Web Applications. *IEEE Communications Surveys & Tutorials*, 16(3):1574–1590, 2014.

[21] Wes Lloyd, Shrideep Pallickara, Olaf David, Jim Lyon, Mazdak Arabi, and Ken Rojas. Migration of Multi-tier Applications to Infrastructure-as-a-Service Clouds: An Investigation Using Kernel-Based Virtual Machines. In *2011 IEEE/ACM 12th International Conference on Grid Computing*, pages 137–144. IEEE, September 2011.

[22] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. Rojas. Performance implications of multi-tier application deployments on Infrastructure-as-a-Service clouds: Towards performance modeling. *Future Generation Computer Systems*, 29(5):1254–1264, July 2013.

[23] Santiago Gómez Sáez, Vasilios Andrikopoulos, and Frank Leymann. Performance-aware Application Distribution in the Cloud. In *Proceedings of the Workshop on Enterprise Architekturen mit Big Data & Cloud (EABDC 2014)*, pages 1–9. Gesellschaft für Informatik e.V. (GI), September 2014.

[24] Antonis Papaioannou and Kostas Magoutis. An Architecture for Evaluating Distributed Application Deployments in Multi-clouds. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 1, pages 547–554. IEEE, December 2013.

[25] Nima Kaviani, Eric Wohlstadter, and Rodger Lea. Cross-tier application and data partitioning of web applications for hybrid cloud deployment. In David Eyers and Karsten Schwan, editors, *Middleware 2013*, volume 8275 of *Lecture Notes in Computer Science*, pages 226–246. Springer Berlin Heidelberg, 2013.