A Model-Based Scalability Optimization Methodology for Cloud Applications

Jia-Chun Lin
University of Oslo
Norway
kellylin@ifi.uio.no

Jacopo Mauro
University of Oslo
Norway
jacopom@ifi.uio.no

Thomas Brox Røst
Atbrox
Norway
thomas@atbrox.com

Ingrid Chieh Yu
University of Oslo
Norway
ingridcy@ifi.uio.no

***Abstract*—Complex applications composed of many interconnected but functionally independent services or components are widely adopted and deployed on the cloud to exploit its elasticity. This allows the application to react to load changes by varying the amount of computational resources used. Deciding the proper scaling settings for a complex architecture is, however, a daunting task: many possible settings exists with big repercussions in terms of performance and cost.**

In this paper, we present a methodology that, by relying on modeling and automatic parameter configurators, allows to understand the best way to configure the scalability of the application to be deployed on the cloud. We exemplify the approach by using an existing service-oriented framework to dispatch car software updates.

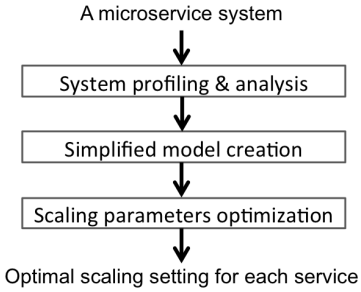


Figure 1: The Scaling Optimization methodology.

I. Introduction

Cloud computing has changed the traditional business model of IT enterprises by offering on-demand delivery of IT resources and applications over the Internet with pay-as-you-go pricing [3]. With the growth of using virtual computing environments, the need to develop scalable, adaptable, modular, and quickly accessible cloud-based applications are in high demand. For this reason, in recent years, service architectures have grown in popularity, allowing applications to be developed as a collection of small services, each running in its own virtual machine and communicating with each other by means of lightweight mechanisms. With the evolution of containers, (distributed) enterprise applications can be constructed as a series of independently deployable components making full use of the service properties (e.g., flexibility, maintainability, reusability, compositionality) with the horizontal scalability of the cloud, i.e., the possibility of adding or removing instances to respond to workload changes.

The cloud infrastructure on which a single service is deployed can be configured to the needs of that component. However, a series of components that do not flexibly adapt to deployment decisions either require wasteful resource overprovisioning or time-consuming reengineering, which may substantially increase costs in both cases. The number of the replicated services have an impact on the performance that is often difficult to assess but important since one needs to take care of it not only during the design and initial deployment, but also whenever changes to application workflow or structure are made. Selecting the best scaling strategies that vary the amount of service instances is far from being trivial since many parameters, application behaviors, and delays need to be taken into account to minimize the overall cost of the system and, at the same time, fulfill a given goal or Service Level Agreement (SLA).

Clearly, classical queuing theory results, bound analysis, and other analytic and approximation techniques [8] can be used to determine the minimal or maximal number of instances that are required to handle a given number of requests with a given latency. Unfortunately, these techniques are not powerful enough to deal with a dynamic system that scales up and down. Due to the exponential explosion of possible settings w.r.t. the number of components and the fact that running a complex system on the cloud is usually an expensive and time consuming-activity, it is often impossible to try all the possible scaling strategies. For this reason, models are created to predict the effect of certain setting choices, thus avoiding the need to run the entire system.

In this paper we propose a methodology to optimize the scaling strategy for components. We discourage to directly proceed with the optimization of the settings of the entire component system. Instead, as shown in Figure 1, as a first step we propose to use worst case analysis (e.g., queuing theory) and profiling techniques to understand what parts of the system can be simplified. This allows in the second step to create a simple model with fewer parameters to tune. Finally, in the third and last step of the methodology, we propose to proceed in the search of good settings by using automatic parameter configurator tools [16], [17], [19], i.e., tools that rely on machine learning techniques to explore in a smarter

978-0-7695-6328-2/17 $31.00 © 2017 IEEE
DOI 10.1109/SC2.2017.32

and more systematic way the possible configurations to come up with good parameter settings.

Paper organization. Section 2 provides a background introduction to services and Amazon Web Services. Section 3 presents the HyVar system as a running example to exemplify our methodology. In section 4, we introduce the proposed scaling optimization methodology. In section 5, we validate our model by comparing it with the real service-oriented system of HyVar on AWS. Section 6 surveys related work and Section 7 concludes the paper.

II. Services and Amazon Web Services

Service architectures [12] allows to build a complex system on the cloud as a set of small independent services. Each of the services provides one or more capability of the system, runs in its own independent process, and is deployed on a potentially different platform and technological stack, independently of the other services. To collaborate with other services each service communicates with the others over a lightweight medium such as RESTful or RPC-based APIs. By making services completely independent in development and deployment, a service-oriented architecture provides great benefits in terms of flexibility, maintainability, reusability, scalability, failure isolation, and resiliency.

To facilitate users to easily deploy services on the cloud, a cloud service provider such as Amazon Web Services (AWS) [2] provides AWS Elastic Beanstalk [5], which is also an easy-to-use platform that allows the scaling of the services. Users need to upload the code of a service and decide its scaling strategy and AWS Elastic Beanstalk will then handle the deployment details of load balancing and service health monitoring. Each independent service can be automatically scaled up or scaled down based on its specific needs by using easily adjustable scaling settings. Different metrics (e.g., latency, CPU utilization, disk read operations) could be used to trigger the scaling. When a threshold is triggered (e.g., average latency is higher/lower than a predefined value), a certain number of Virtual Machine Instances (VMIs) will be added to support the service or removed to not waste unused resources. By choosing the correct scaling settings, each service is able to handle requests on demand while reducing the cost for renting VMIs. This type of auto scaling is usually classified as reactive since it takes actions with a rule-based design.

III. Running Example

In this section, we introduce the framework, dubbed HyVar, that we use to exemplify our scaling optimization methodology. HyVar [18] is a system for the continuous and individualized evolution of software applications running on cars equipped with ECUs (Electronic Control Units). It is developed within the European Project HyVar where the goal is to propose a development framework for continuous and individualized evolution of distributed software applications running on remote devices in heterogeneous environments. In particular, the HyVar framework, now deployed on AWS, is responsible for collecting information from a fleet of cars

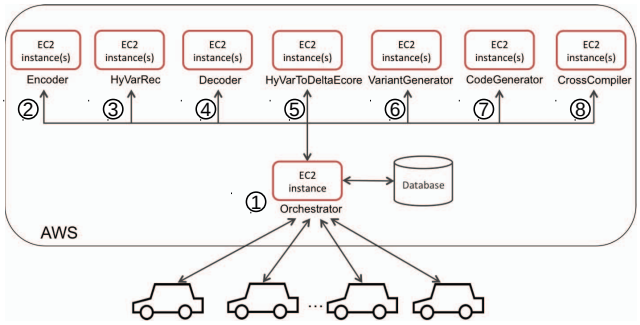


Figure 2: The HyVar system deployed on AWS.

and, in case some software changes are needed, compile the new software update and send it back to the car. HyVar is designed as a service-oriented system with the architecture illustrated in Figure 2. Whenever receiving a request from a car, the orchestrator (① in the figure) manages the entire processing workflow, sending requests to individual services, and collecting the answers. The services are invoked in sequence. First, the Encoder ② is used to convert the car input and all the software information to a format required by the reasoner HyVarRec [20], [22], i.e., the component that is responsible for understanding which is the best software to send to the car based on the car's status. The orchestrator collects the output of the Encoder and then, using its answer, invokes HyVarRec ③. The output of the reasoner is in turn processed and decoded by the Decoder ④. If the output shows that the car does not need any software updates, the orchestrator will simply drop the request. Otherwise, it proceeds to the generation of software updates to be sent back to the car. This part is done in sequence by the remaining components of the toolchain, namely HyVarToDeltaEcore ⑤, VariantGenerator ⑥, CodeGenerator ⑦, and CrossCompiler ⑧ that finally builds the binary image for the car. For more details related to the HyVar system we defer the interested reader to [10].

The above components are individually deployed as independent services on AWS, so each of them has its own pool of VMIs, load balancer, and auto scaling to handle the requests sent from the orchestrator in parallel. The service times of all the components are below 1 second with the exception of HyVarRec that takes less than 1.5 seconds to service a request, and the Cross Compiler that can take more than half a minute.

IV. Scaling Optimization Methodology

In this section we describe the proposed methodology to optimize the scaling parameter setting for each service of a service-oriented system. A graphical representation of the steps involved to optimize the settings is presented in Figure 1. In summary, we propose to start by collecting key information of the system by conducting a profiling activity and analysis. This information is exploited in a second step to define a model of the service-oriented system that is used in a third phase to

derive the best scaling settings. In the following we will detail these phases starting with our assumptions.

Assumptions We start by assuming that we are dealing with a service-oriented system composed by one or more services that can be deployed independently one from another. Indeed, to be able to exploit the elasticity of the clouds, instances of the services should be able to be created and removed on demand. This often implies that services need to be stateless, otherwise the removal of a stateful service means the loss of information that could have an impact on the semantic behavior of the entire system. For this reason, we restrict ourselves to study scaling strategies for services that can be created and removed on demand without causing semantic behavior changes.[1]

We also assume that the DevOps engineer[2] or the responsible for the deployment of the system has a clear goal, represented by a Software Level Agreement (SLA) [28]. Different notions of SLA have been proposed in the literature. In this context we do not deal with non-functional properties (e.g., security) assuming also that every service is implemented correctly. Instead, we target functional and measurable properties of the systems such as requirements on the distribution of the average response time or maximal response time.

As an example, for HyVar the car manufacturer company is willing to let a car wait for hours to get a possible software update.[3] To demonstrate our approach, we restrict ourselves to consider a SLA requiring the average response time to be below 5 minutes. Please note that since single requests are allowed to take even hours, differently to what is usually done and to exemplify the potential of the methodology, for our running example we do not require the minimization of the maximal latency of a single request but only that the global average solving time should be below a given threshold. As we will later see, this allows the use of scaling strategies causing the system to have peaks in latency (something that is usually avoided), but overall using less instances in average. The metric considered by our methodology can however be extended to more elaborate requests, e.g., requiring that 95% of the requests are processed within a given time window.

The key ingredient to being able to optimize the scaling strategies of a service system is to have a reliable and executable model that reflects the system performance. To achieve this, performance indicators on the service are needed. For every service of a service system, we require to know the average response time taken by only one instance to process a request and, if possible, also the corresponding resource consumption, e.g., CPU, Memory, and I/O Bandwidth. Moreover, since a newly created VMI on the cloud takes some time to become operational, we require the time taken for the VMI to become ready to process requests. [4]

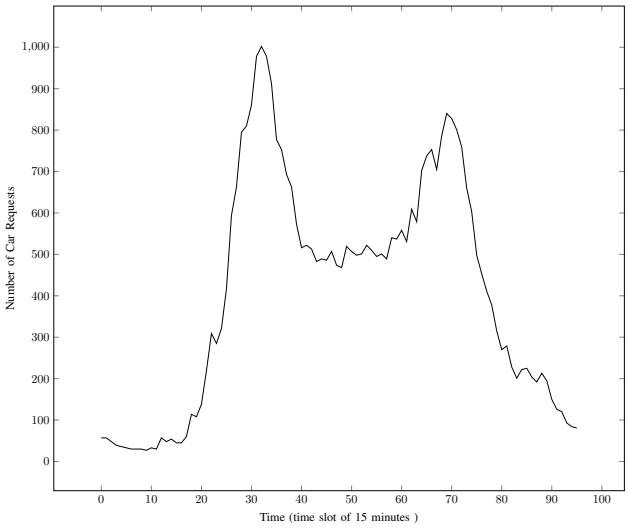


Figure 3: Traffic pattern for HyVar.

To be able to run the simulation, we finally assume to have an estimate of the request arrival pattern. If this is not possible, we require at least to know important factors of the arrival distribution such as the average arrival rate, its variance, or the maximal arrival rate. For the HyVar system, we use as an example a 24-hour car traffic pattern to derive the number of cars requiring updates.[5] Figure 3 shows the number of car requests for time windows of 15 minutes with two peaks, corresponding to the time when people drive to and from work.

System Profiling and Analysis With a request arrival distribution and the average response time for each service, we could immediately create a model having every concrete scaling parameter setting of the real system and then use it to optimize these parameters. However, proceeding directly with this modelization system might not be ideal since we may end up with a complex model with too many parameters to optimize. To ease the optimization task, we suggest instead to simplify the model as much as possible by using worst case analysis and profiling techniques. Since we are dealing with services, one of the most useful techniques to simplify the model and get a bound on its settings is queuing theory [8]. In a cloud such as AWS, a service having c instances can be modeled by c single queues and a switcher that redirects incoming requests to these queues. Assuming that each service does not use any scaling strategy, the entire service system can therefore be represented as a BCMP network [6] and

[1]Obviously, the removal of an service instance has an impact on the performance of the system in that it may take more time to solve a task. However, this does not imply a change in the semantic behavior since the task will still be solved, albeit it will take more time.

[2]DevOps is a clipped compound of "software DEVelopment" and "information technology OPerationS" to denote the people responsible not only for the development of a software, but also its deployment and maintenance.

[3]For security reasons, the software of the car is not updated on-the-fly but requires the car to be shut off. Hence, waiting even an hour is not a problem.

[4]Clearly, the more information is available (e.g., distribution of the solving times) the better, because this information can be used to obtain more precise simulations.

[5]In particular, the traffic pattern is derived from the website data.gov.uk listing the number of cars registered on the A414 highway, UK, on Monday, March 2, 2015. We assume that only 1 out of 3 requests needs a firmware update, implying that 2 out of 3 requests are dropped and do not need to visit all the services of HyVar.

queuing theory may be used to derive analytically important properties without any simulation. For instance, assuming to have a phase-type distribution that approximates the original arrival distribution [14] like in our case, by using tools such as JMT [7] it is possible to compute the average response time for every single service of the system.[6]

For our use case, based on the expected car traffic and the average serving times recorded on Amazon `m1.small` instances,[7] by using queuing theory we can derive that to have a stationary distribution HyVarRec requires between 1 and 2 instances and the CrossCompiler requires between 1 and 13 instances. The other components are fast enough that they need only one instance and do not need to scale up. Clearly, this information allows us to simplify the model since we only need to find optimal scaling settings for 2 out of the 8 services. Moreover, due to the fact that by using queuing theory we can compute analytically the average response time for each service that does not need to scale, we can further simplify the model by merging sequentially the services that do not need to scale up. As an example, considering HyVar, the components HyVarToDeltaEcore, VariantGenerator, and CodeGenerator can be faithfully replaced by one dummy component having an average response time that combines their average response times (and the time taken by the orchestrator to send and receive the messages to these services).

Since it is well known [24] that I/O read/write speed and the performance of the instances in a public cloud vary a lot, it may be beneficial to use profiling techniques to assess if the bound obtained analytically holds also in practice. Due to the distributed nature of the service system, it is relatively easy to conduct stress tests on single components for a short amount of time to understand if they can sustain the expected maximal traffic. For our use case, we performed stress tests considering only one instance for every service verifying empirically that they were able to sustain the traffic. In this analysis we also noticed that the I/O resources of the orchestrator were close to the limit when its VMI had performance degradation, probably due to the concurrent use of its physical machine by other cloud users. Since the Orchestrator service is the entry point of the HyVar system, we decided in this case to be on the safe side and use a `large` instance with more I/O capacity instead of the default `small` that is used for all the other VMIs.

Profiling is also important to discover potential unforeseen problems of the system. For instance, by running a first version of HyVar we discovered that the Amazon Classic Load Balancer that was supposed to route requests using the least outstanding requests routing algorithm[8] was not behaving exactly as advertised: sometimes it routed two requests to an instance that should have received only one. This may be acceptable for short, light-weight request systems (for which Amazon Web Services is designed), but it is certainly not good for components like the Cross Compiler that, different from typical web services, may take more than half a minute to process a request. In this case then the profiling allowed us to discover the deviance from the expected behavior, forcing us to replace the Amazon load balancer with HAProxy.

Simplified Model Creation To create an executable model, it is possible to use existing simulators such as, .e.g, CloudSim [9] or iCanCloud [23]. In our case we decide to create a model in ABS [1], i.e., a formal, executable, object-oriented language for modeling distributed systems by means of concurrent object groups. While every language can be used for this purpose, we adopted ABS because: i) it natively supports CPU, memory resources and the notion of deployment components thus easing the task of modeling the performance of a single instance in few lines of code, ii) being a full-fledged language it is more flexible than an ad-hoc simulator, iii) it has parallel run-time support in Erlang, iv) tools for further worst case performance analysis are available.

Regardless of the simulation approach used, what is important is that the model needs to expose and allow the tuning of some parameters. For simplicity, here we consider the following scaling setting parameters that can be found in most of the elastic scaling cloud platforms.

- `scale_up_threshold`: the metric threshold that if violated triggers the scaling up action of a VMI,
- `scale_down_threshold`: the threshold that if violated triggers the scaling down action of a VMI,
- `scale_up_amount`: the amount of new VMIs that are created on every scaling up action,
- `scale_down_amount`: the amount of new instances that are discontinued on every scaling down action (obviously it is not possible to discontinue all the VMIs associated to a service —at least one instance must be connected to the load balancer),
- `cooling_off`: the time window where no scaling actions are allowed after a scaling action has been triggered.

Scaling Parameters Optimization To control how the services scale, the model provides the ability to set scaling parameters for each one of them. These parameters translate directly to real settings of the cloud service provided by Amazon. For our use case, we decided to use the latency of the individual service response time to decide when the system needs to scale up or down.[9] Deciding good values of these parameters is important for the minimization of the cost of the running system. Usually the parameters are tuned manually but empirical evidence shows that using automatic configurators such as [16], [17], [19] can lead to better configurations. In our methodology we therefore propose to use these tools to methodically explore the possible scaling settings. In particular, for our experiments we used the state-of-the-art configurator SMAC [16]. SMAC can be used to optimize different parameters with different input patterns

[6]For an overview of the performance upper bounds obtainable by using queue theory we refer the interested reader to [8].

[7]https://aws.amazon.com/ec2/instance-types/

[8]http://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/how-elastic-load-balancing-works.html

[9]We noticed that the CPU load was not very correlated with latency performances since not all the jobs of the HyVar system are CPU load intensive.

and stochastic simulations. It requires the definition of the parameters to configure, their nature (categorical, ordinal, integer) and their bounds, if any.

The configurator is the most computationally intensive task requiring to run many simulations with different parameters. It is often impossible to exhaustively try all possible settings due to their exponential blow up w.r.t. the number of services, but the more time the configurator is allowed to run, the greater are the chances that the best solution is close to the optimal one. If the search space can not be explored exhaustively, it is up to the user to decide when to interrupt the configurator. Once terminated, the configurator returns the best settings to use to configure all the services of the system.

V. Optimizing HyVar

In this section, we present the results we obtained by applying the methodology described in the previous section to the HyVar use case. Before presenting the experimental results we first start by presenting some details of the developed model and how we used the SMAC parameter configurator.

Model As previously stated, we created the model of HyVar in ABS. A detailed description of the ABS language and the model is beyond the scope of this paper. In the following we detail the relevant aspects of the model referring the reader to [1] for details about the ABS language, and to https://github.com/HyVar/abs_optimizer/blob/master/abs_model/HyVarSym.abs for the model code.

Every service is modeled using two classes: `Instance` and `Load Balancer`. An instance object mimics the behavior of a service deployed on a VMI. It therefore can be invoked to solve a task that consumes some resources. The `Load Balancer` class, as the name entails, mimics instead the behavior of the load balancer distributing the requests to the different service deployed on the VMIs. This is the component that also tracks the execution times of the jobs and the performances of the deployed services, deciding to scale up or down when the thresholds are surpassed.

The simulation of the workflow can be done by simply assigning tasks to the different load balancers, invoking the load balancers in the desired order. As an example, the following ABS code assigns the request to a load balancer `loadbalancer1`, waits for its output and assigns the request to the next load balancer `loadbalancer2`, dropping the request randomly in two third of the cases.

```
1 Fut<Bool> future = loadbalancer1!exe(request);
2 await future?;
3 Bool result = f.get;
4 if (result && random(3) == 0) {
5   future = loadbalancer2!exe(request); }
```

In the first line the request is assigned to the load balancer `loadbalancer1` by invoking its method `exe`. The invocation is done asynchronously (this in ABS is specified by `!`) and returns a future, i.e., an object that acts as a proxy for the result of the call (when done asynchronously, at the invocation time the result is unknown). At Line 2, the `await` statement waits on the future, meaning that it waits for the return of the previously invoked `exe` method. Since we are abstracting from the concrete behavior of the service represented by the load balancer, the `exe` method returns a boolean value representing if the method has successfully terminated or there has been some errors. The return value is collected at Line 3 by using the `get` statement. The remaining lines are then used to send a request to the load balancer `loadbalancer2` similarly to what has been done for `loadbalancer1`. The `random` function is used to perform the request in only two third of the cases.

We remark that while the model of the load balancer and the services can be reused, the encoding of the workflow is application specific. Luckily, the definition of the workflow often requires few lines of code depending on the complexity of the iterations between the services. Clearly, the model can also be developed in another language. In this case the only requirement is that the model should allow setting scaling parameters and use them to derive the running cost of the system (e.g., average cost of the VMIs used) and if three are SLA violations.

We also remark that the model can be stochastic. This can be extremely important for taking into account the variability of performances in a cloud. Note that the model does not need to be fully faithful: the stochasticity or other approximations may distance the simulation results from a concrete execution of the system. What is important is that the model could be used to evaluate worst case performances of the system.

Automatic parameter configurator To perform the optimization of the parameters we used SMAC [16]. This tool can be run on a stand-alone computer or in a cluster. However, in the latter case, it requires a memory sharing architecture. To conduct the experiment we used 64 nodes of a Numascale Shared Memory Cluster [10] to run in parallel 64 instances of SMAC for 12 hours. We used the experimental SMAC Shared Model Mode that allows SMAC to use multiple runs to share data and construct better models quickly. The optimizer can however also be run on a single machine. For this reason, to ease the installation and its execution, we packaged the model and the optimizer in a Docker container, publicly available at https://github.com/HyVar/abs_optimizer.

For the use case of HyVar, due to the analysis step (see Section IV) we found out that only two components are required to scale up. We therefore had to optimize only 5 parameters for HyVarRec and the Cross Compiler, for a total of 10 parameters.[11] As an example, the SMAC input used to configure the scalability settings is the following.

```
scale_up_thr_hyvarrec integer [4,300] [10]
scale_up_thr_code_gen integer [41,300] [100]
scale_down_thr_hyvarrec integer [3,300] [5]
scale_down_thr_code_gen integer [40,299] [50]
...
cooling_off_time_code_gen ordinal
   {240,300,360,420,480} [300]
```

[10] https://www.numascale.com/

[11] Note that without the model simplification the total number of parameters would have been 35.

```
{scale_up_thr_code_gen <=
   scale_down_thr_code_gen ||
 scale_up_thr_hyvarrec <=
   scale_down_thr_hyvarrec}
```

In the first line we define the parameter `scale_up_thr_hyvarrec` representing the scaling threshold of the HyVarRec. The line starts by defining the name of the parameter followed by its type (`integer`), its range `[4,300]`, and finally its default value (`[10]`). Similarly, the following lines define the other parameters. The meaning associated with the parameters vary according to the parameter. The last four lines add some constraints on the parameters enforcing the scaling up thresholds to be bigger than the scaling down thresholds.

The last ingredient needed by the configurator is the metric to optimize. For the HyVar system we are interested in having the least expensive system able to answer the requests in less than 5 minutes in average. Hence, we set the average number of VMIs used as the metric to minimize, penalizing with a high value the configuration for which the average latency of the system was greater than 5 minutes. The following Python code presents how the metric is computed.

```
if average_latency < 300:
  return average_latency +
    int(average_vmi_number * 100 * 300)
else:
  return average_latency + 600000
```

Since, thanks to the worst case analysis, we know that a system with only 13 Cross Compiler and 2 HyVarRec VMIs is enough, minimizing this metric is the equivalent of minimizing the cost of the system using the average latency to distinguish two configurations having the same cost.

Real Run Experiments To validate if the predictions of our model were accurate enough we have run the HyVar framework with the traffic pattern presented in Section IV. To verify that the bounds obtained by the worst case analysis hold, we first run HyVar by disallowing the scaling, starting with 2 VMIs for the HyVarRec service and 13 VMIs for the Cross Compiler. In the following we named this experiment `no-scaling`. Then we run the experiment by setting the parameters returned by the SMAC optimizer. We named this experiment `scaling`. For the scaling experiment, considering the HyVarRec and the Cross Compiler services in this order, SMAC decided to scale up a component when the latency was above 20 and 300 seconds, scale down when the latency was below 4 and 253 seconds, and increase and decrease the VMIs by one. The cooling off time was 6 minutes for HyVarRec and 8 minutes for the Cross Compiler. These values were computed taking 1650 simulations into account.[12]

Note that these settings were decided by the optimizer and are guaranteed to be the ones of the best simulation. Unfortunately, due to the mechanics of the optimizer, we can not directly associate an explanation justifying why a single setting value is better than another.

Figure 4a presents the plots of the average latency time of the request for every 5 minutes of the `no-scaling` experiment. It is clear from the plot that HyVar is able to handle the traffic with an average latency that is lower than 40 seconds. As intended, the simulation slightly differs from the real system due to some approximations added to take into account that cloud VMIs often experience performances decreases. The two spikes where the average latency curve of the real system overcomes the average latency time of the simulation is indeed due to this reason. In any case, as desired, the average solving time of the real system is significantly statistically lower than the simulation one. [13]

Figure 4b presents instead the latency and the number of VMIs for the `scaling` experiment. Due to the (reactive) scaling framework usually adopted by the clouds, when the load increases the system has big peaks of latency due to the fact that requests are queuing or being processed in parallel. The latency is then reduced as soon as more VMIs are added. This can be observed both in the simulation and in the logs of the real system. The latencies of the simulation, as intended, are just slightly worse that the reals ones to account for possible performance degradation of the cloud VMIs. As before, the average latency of the real experiment is statistically significant lower than the simulation.

As can be seen from Figure 5 presenting the plot of the VMIs used, the scaling decisions taken by both the system and the simulation are similar. As expected, there is indeed a strong correlation (0.93) between the number of VMIs in the simulation and those actually used, and the peaks of the latency of the simulation follow the same pattern of those in the real experiment.

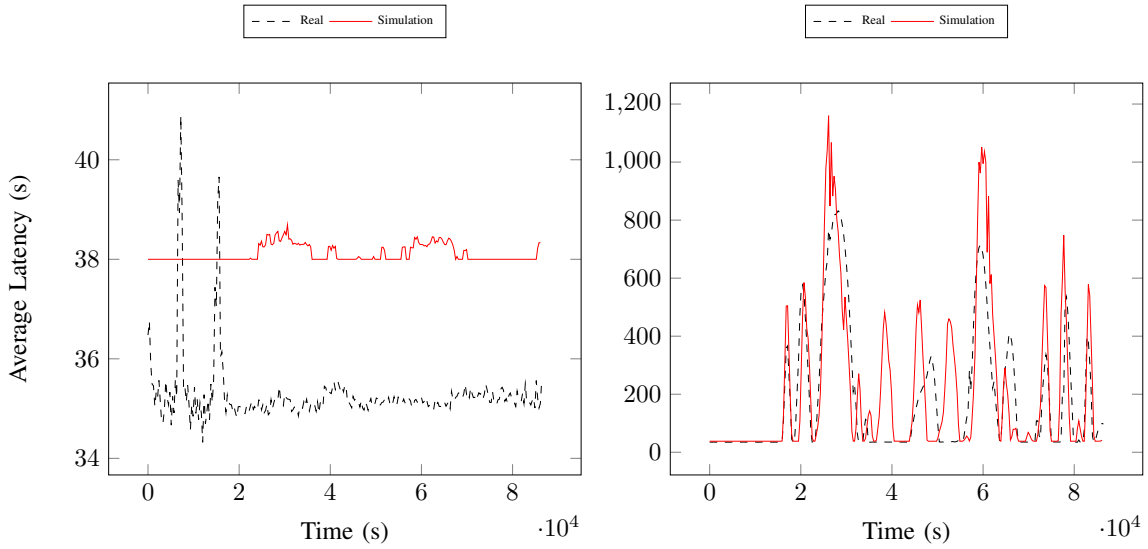
In summary, the simulation was robust enough to mimic the real system and offers a performance estimation that can be used to set good scaling parameters. The average cost in terms of VMIs used went from 20 to 12.29 VMIs, at the price of increasing the average latency from 35.2 to 263,2 seconds. While at a first glance this seems a huge price to pay, it was instead exactly what we were looking for: a good scaling strategy that was using fewer instances but still capable of keeping the average response time within 5 minutes, no matter how big the maximal latency was.

VI. Related Work

In cloud computing, elasticity is defined as the degree to which a system is able to automatically provision and de-provision resources so as to adapt to workload changes [15]. Auto-scaling, also called automatic scaling, is a method used in cloud computing to achieve elasticity. As far as the modelization of elasticity is concerned, Suleiman and Venugopal [27] enable users to study the effects of elasticity

[12]Please note that while the use of a cluster certainly helps in conducting in parallel different simulations, good values of the parameters are still possible to obtain by simply running the model using the docker container and a normal machine. For instance, by using a 4 core machine with 8GB of RAM in an OpenStack cloud we were able in 12 hours to perform 117 simulations, having a solution only 5% worse than the one obtained using the cluster.

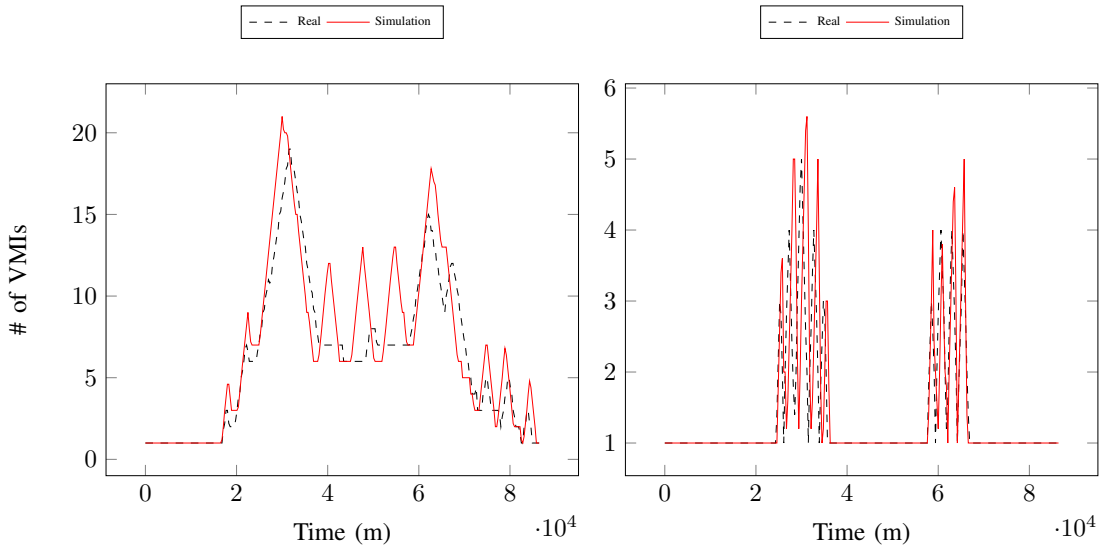
[13]To test the statistically significance we used the Welch's t-test, i.e., a two-sample location test which is used to test the hypothesis that two populations have equal means.



(a) `no_scaling` experiment.

(b) `scaling` experiment.

Figure 4: Average latency results with a time window of 5 minutes.



(a) Cross Compiler.

(b) HyVarRec.

Figure 5: Average number of VMIs with a time window of 5 minutes.

rules on multi-tier Internet applications deployed on IaaS cloud by modeling a multi-tier Internet application as an M/M/m queue and presenting mathematical models to approximate the values of CPU utilization, application response time, and the number of servers needed to serve the application's workload. Based on these models, the authors developed an algorithm to simulate scale-up and scale-down logic. Singh et al. [26] handled the non-stationarity property of Internet application workload by proposing a mix-aware dynamic provisioning technique, which employs the k-means clustering algorithm to automatically capture workload mix changes in Internet applications and a queuing model to predict the number of servers needed to process expected workload over time. Roy et al. [25] proposed a look-ahead resource allocation algorithm to scale resources to users ahead-of-time based on model-predictive techniques which predict future workload based on a second order autoregressive moving average method. It is clear that all above approaches ensure elasticity at the application level, not at the service level as we do in this paper. Yataghene et al. [29] modeled the elasticity of Service-based Business Processes (SPB) by using a queuing network in which each service that composes SPB is represented by

a queuing model. However, the authors neither investigate how to optimize auto-scaling for each service nor provide an empirical model evaluation.

Other studies aim to optimize auto-scaling mechanisms on the cloud. Among them, one closely related to our work is [11] which presents a model-driven approach to optimize the configuration, energy consumption, and operating cost of cloud auto-scaling infrastructure by capturing various virtual machine configurations in feature models, transforming these models into constraint satisfaction problems (CSP), and using a constraint solver to derive the optimal auto-scaling configurations. The authors in [13] instead proposed an auto-scaling approach with the goal to satisfy the application providers' objectives while optimizing for the cost incurred by resource usages by formulating auto-scaling as a stochastic model predictive control problem and solving the problem using a convex optimization solver. Similarly, the author in [21] introduced a learning automata auto-scaling approach to optimize cost, rate of SLA violations and also the stability in the presence of traffic workload. Asgari et al. [4] proposed an auto-scaling approach using Markov Decision Process (MDP) to manage SLA violation and scaling expense and to preserve system stability. Nevertheless, all these approaches aim to optimize auto-scaling at the application level, rather than at the service level.

VII. Conclusion and Future Work

In this paper we have presented how profiling, worst case analysis, and modelization can be used in combination with an automatic parameter configurator to determine the best scaling parameter for a complex service-oriented system. The key idea behind the proposed methodology is to define the simplest possible model of the system and then use a state of the art configurator to explore in a systematic way the possible configurations, using the artificial intelligence of the configurator to focus on the most promising settings. This methodology has been applied as an example to a service-oriented framework deployed on the public Amazon cloud.

For future work we would like to analyze bigger and more complex systems, extending the configuration to even more parameters (e.g., initial instance, instance type). In particular, we would also like to go beyond the current reactive scaling modality and use configurators to study different proactive strategies that by using time traffic series analysis can anticipate the traffic peaks. This would be extremely important to avoid the peak of latency observed and deploy far more latency-stable systems.

Acknowledgement. The authors thank SIRIUS Centre at the University of Oslo for providing the Numa machine used in our experiments. This work was supported by the European project HyVar (grant agreement H2020-644298).

References

[1] ABS Developers. ABS Manual. http://abs-models.org/documentation/manual/.

[2] Amazon Web Services. https://aws.amazon.com/.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[4] B. Asgari, M. G. Arani, and S. Jabbehdari. An effiecient approach for resource auto-scaling in cloud environments. *International Journal of Electrical and Computer Engineering (IJECE)*, 6(5), 2016.

[5] AWS Elastic Beanstalk. https://aws.amazon.com/elasticbeanstalk/.

[6] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, Closed, and Mixed Networks of Queues with Different Classes of Customers. *J. ACM*, 22(2):248–260, 1975.

[7] M. Bertoli, G. Casale, and G. Serazzi. JMT: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009.

[8] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. Wiley-Interscience, 1998.

[9] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw., Pract. Exper.*, 41(1), 2011.

[10] C. Chesta, F. Damiani, L. Dobriakova, M. Guerinieri, S. Martini, M. Nieke, V. Rodrigues, and S. Schuster. A Toolchain for Delta-Oriented Modeling of Software Product Lines. In *ISoLA*, 2016.

[11] B. Dougherty, J. White, and D. C. Schmidt. Model-driven auto-scaling of green cloud computing infrastructure. *Future Generation Computer Systems*, 28(2), 2012.

[12] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[13] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, and G. Iszlai. Optimal autoscaling in a IaaS cloud. In *ICAC*. ACM, 2012.

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

[15] N. R. Herbst, S. Kounev, and R. H. Reussner. Elasticity in cloud computing: What it is, and what it is not. In *ICAC*, 2013.

[16] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION*, 2011.

[17] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *J. Artif. Intell. Res. (JAIR)*, 2009.

[18] HyVar: Scalable Hybrid Variability. http://www.hyvar-project.eu/hyvar/.

[19] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC - Instance-Specific Algorithm Configuration. In *ECAI*, 2010.

[20] J. Mauro, M. Nieke, C. Seidl, and I. C. Yu. Context Aware Reconfiguration in Software Product Lines. In *VAMOS*. ACM, 2016.

[21] K. Mogouie, M. G. Arani, and M. Shamsi. A novel approach for optimization auto-scaling in cloud computing environment. *International Journal of Modern Education and Computer Science*, 7(8), 2015.

[22] M. Nieke, J. Mauro, C. Seidl, and I. C. Yu. User Profiles for Context-Aware Reconfiguration in Software Product Lines. In *ISoLA*, 2016.

[23] A. Nuñez, J. L. Vázquez-Poletti, A. C. Caminero, G. G. Castañé, J. Carretero, and I. M. Llorente. iCanCloud: A Flexible and Scalable Cloud Infrastructure Simulator. *J. Grid Comput.*, 10(1):185–209, 2012.

[24] Philipp Leitner. Benchmarking the Performance Variability in Public Clouds. https://blog.thestove.io/benchmarking-the-performance-variability-in-public-clouds-480b95ad7cca.

[25] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *CLOUD*. IEEE, 2011.

[26] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *ICAC*. ACM, 2010.

[27] B. Suleiman and S. Venugopal. Modeling performance of elasticity rules for cloud-based applications. In *EDOC*. IEEE, 2013.

[28] P. Wieder, J. M. Butler, W. Theilmann, and R. Yahyapour. *Service Level Agreements for Cloud Computing*. Springer Publishing Company, Incorporated, 2011.

[29] L. Yataghene, M. Amziani, M. Ioualalen, and S. Tata. A queuing model for business processes elasticity evaluation. In *IWAISE*. IEEE, 2014.