Scalability Analysis of Cloud Software Services

Grunnar Brataas*, Nikolas Herbst†, Simon Ivanšek‡ and Jure Polutnik‡
*SINTEF Digital, Trondheim, Norway, †University of Würzburg, Germany, ‡XLAB, Ljubljana, Slovenia

***Abstract*—Cloud computing theoretically offers its customers unlimited cloud resources. However, the scalability of software services is often limited by their underlying architecture. In contrast to current scalability analysis approaches, we make work parameters, quality thresholds, as well as the resource space explicit in a conceptually consistent set of equations. We propose two scalability metric functions based on these equations. The resource scalability metric function describes the relation between the capacity of the multi-tier cloud software service and its use of cloud resources, whereas the cost scalability metric function replaces cloud resources with cost. We validate using the CloudStore application. CloudStore follows the TPC-W specification, representing an online book store. We have experimented with 21 different public Amazon Web Service configurations and two private OpenStack configurations.**

I. Introduction

With the growing complexity of cloud software services and their unpredictably increasing workload, the scalability of these software services becomes critical. We refer to *scalability* as *"the ability of a service to increase its capacity by consuming more resources in the resource space."*, extending the definition by Lehrig et al. [1]. By *resource space*, we mean the *set* of possible resource configurations. The term *service* refers to typical multi-tier, session-based software-as-a-service (SaaS) applications, and *resources* are infrastructure-as-a-service (IaaS) offerings.

Cloud computing providers theoretically offer their customers unlimited resources [2]. However, scalability is also determined by the control and data flow. Architectures and implementations can lead to 1) under-provisioning together with SLO (Service Level Objectives) violations, like high response times or low throughput, resulting in dissatisfied customers, or 2) over-provisioning and low utilization of resources, leading to high costs [3].

We propose a conceptually consistent set of equations capturing the influencing factors of observable cloud service scalability. In contrast to current scalability analysis approaches, these equations make work parameters, quality thresholds, as well as the resource space explicit. We select capacity as the base metric to describe the scalability of a service and propose a measurement method that resulted in reliable measurement values. Based on the equations, we propose two scalability metric functions. The resource scalability metric function describes the relation between the capacity of the cloud software service and its use of cloud resources. This metric is applicable with *one* type of cloud resources, for example the size of app VM instances. When more types of cloud resources are added, we use the cost scalability metric function that illustrates the relation between the capacity of a cloud software service and its cost of cloud resources. The proposed metric functions allow for investigating the impact of work parameters, as well as of quality thresholds on service scalability.

With empirically obtained scalability metric functions, a deployment controller knows in advance how many customers the service can handle within SLO-bounds for all evaluated configurations of allocated resources and in addition can decide which configuration is more cost-efficient. The deployment controller can also in an informed way select the layer to scale and whether to scale vertically or horizontally.

We empirically evaluate our scalability metric functions using CloudStore [4], an implementation of the TPC-W specification [5]. CloudStore is deployed on the public Amazon Web Services (AWS) [6] as well as on a private OpenStack [7]-based environment.

We conducted 53 measurements with 21 different AWS configurations. Most AWS measurements and 20 OpenStack measurements were conducted by XLAB. 6 AWS measurements were done at the University of Würzburg. SINTEF repeated some AWS measurements with similar results.

In Sec. II, we specify relevant scalability parameters. Our two scalability metric functions are described in Sec. III. The measurement plan is introduced in Sec. IV. In Sec. V we describe our experiment setup. Measurements are discussed in Sec. VI. In Sec. VII, we consider related work. Conclusions, limitations and further work are offered in Sec. VIII.

II. Service Specification

Fig. 1 describes how users invoke load and work. Operations with quality metrics and thresholds are offered by services. The services are deployed on resources. Using equations, in this paper we describe the essence of factors influencing service scalability. Possible extensions to the equations are outlined. The notation is illustrated with examples from CloudStore.

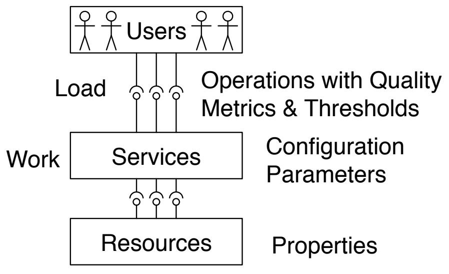

Fig. 1. Essential service scalability concepts [8].

A. Operations

An operation defines a unique way of interacting with a service. It corresponds to a request class in the context of queuing networks. Service s has $\#op$ operations:

$$op_s = (o_{s,1}, \cdots, o_{s,\#op}) \tag{1}$$

For each operation i, $o_{s,i}$ represents the probability of the corresponding operation. The probabilities of all the operations represent the operation mix and sum to 1: $\sum_{n=1}^{\#op} o_{s,n} = 1$.

CloudStore (cs) has 14 operations. We use the TPC-W Browsing Mix [5, p. 98]:

$$\begin{aligned} op_{cs} = (&0.29, 0.11, 0.11, 0.21, 0.12, 0.11, 0.02, 0.0082, \\ &0.0075, 0.0069, 0.0030, 0.0025, 0.0010, 0.0009) \end{aligned} \tag{2}$$

B. Load

Load describes how often the operations in a service are invoked. With a constant number of users, we have a closed system which is specified by the number of users (N), in addition to the think time (Z):

$$lc_s = (N, Z) \tag{3}$$

A variable number of users corresponds to an open system. For an open system, we use arrival rate, λ:

$$lo_s = (\lambda) \tag{4}$$

We consider CloudStore to be a closed system. Load is specified by the number of virtual users (VU). The think time in CloudStore is a constant value of seven seconds, i.e., a virtual user can at most make a new request every seven seconds.

C. Work

Work characterizes the amount of data to be processed, stored or communicated by a service. The values of each of the $\#wp$ work parameters for service s are:

$$wp_s = (w_{s,1}, \cdots, w_{s,\#wp}) \tag{5}$$

In TPC-W, there are at least three work parameters: 1) number of books, 2) number of customers and 3) size of images. For image sizes, we always use the values in the TPC-W specification [5, pp. 11]: 45% 5KB, 35% 10KB, 15% 50KB, 4% 100KB, and 1% 250KB. In addition, things like average number of books in the shopping cart are hard-coded in the TPC-W specification. Therefore, the work characterization could easily be elaborated. We normally use the following values for the number of books and customers:

$$wp_{cs} = (10\,000, 288\,000) \tag{6}$$

We have now specified global work parameters for a complete service, but work parameters may also apply to specific operations. If this is required, we may distinguish between global and operation specific work parameters, using the following notation: $w_{s,n} = (wo_{s,n,0}, wo_{s,n,1}, \cdots, wo_{s,n,\#op})$ Global work parameters are then specified as for a dummy operation 0. The value of $wo_{s,n,m}$ is the value of the work parameter m for operation n.

Work parameters might be subject to change over time due to side effects of executed operations. In CloudStore, this is not the case for the number of books, but the number of customers is related to the load since new customers are registered with the Customer Registration operation. Therefore, what we specify for the number of customers is the initial value.

D. Quality Metrics and Thresholds

A quality metric defines how we measure a certain quality and is a key part of an SLO (Service Level Objective). We assume one metric for all operations. Quality thresholds (QTs) describe the border between acceptable and non-acceptable quality for a given metric. All $\#op$ operations have a quality threshold as specified by the parameters $q_{s,n}$:

$$qt_s = (q_{s,1}, \cdots, q_{s,\#op}) \tag{7}$$

In CloudStore, we use the 90 percentile response time as our quality metric. The values for each of the quality thresholds, in seconds, are described in [5, p. 104]:

$$qt_{cs} = (3, 5, 5, 3, 3, 10, 3, 3, 3, 5, 3, 3, 3, 20) \tag{8}$$

All 14 operations must obey the quality threshold. Therefore, if one of the 14 operations has more than 10% SLO violation, the quality metric is violated.

Instead of a single metric, we can use $\#met$ different metrics. Then $q_{s,n} = (qm_{s,n,1}, \cdots, qm_{s,n,\#met})$ represents the quality thresholds for all metrics for each operation n.

E. Cloud Resources

Public cloud providers like Amazon Web Services (AWS) can basically be scaled across three dimensions: (1) Instance type, optimized for a given work profile, e.g.: compute or storage optimized. There may also be several generations of each instance type. (2) Instance (t-shirt) sizes for each instance type, like small, medium, large. (3) Number of instances.

For AWS we specifically focus on Elastic Computing Cloud (EC2) and the Relational Database Services (RDS). With more load, there are basically two scaling options: a larger instance type or size (vertical scaling) or more instances (horizontal scaling). EC2 offers horizontal auto-scaling. Vertical scaling of EC2 instances currently requires manual work. For RDS, neither horizontal nor vertical scaling can be done on the fly.

A service is deployed on $\#vmg$ different cloud virtual machine (VM) groups. A VM group consists of homogeneous resources and has a vendor, an instance type, an instance size and number of instances:

$$v_i = (vendor, type, size, inst) \tag{9}$$

A VM group can be considered as a tier in a distributed service architecture. Number of instances in a group are used for horizontal scaling, while vendor, instance type and instance size can be used for vertical scaling.

As an example, (*Amazon, EC2*, *m3*, *medium*, 4) describes a group of 4 instances at Amazon EC2 of size m3.medium.

All the VM groups required by a service are as follows:

$$vmg_{s,d} = (v_1, \cdots, v_{\#vmg}) \tag{10}$$

We use the term deployment configuration when we specify a set of cloud resources. We use the index d to represent a specific deployment configuration for service s. During a scalability analysis, the deployment configurations are subject to be altered in an exploratory or systematic way.

In CloudStore, we focus on changing the size and number of instances in the application VM group and size of the instance in the DB VM group.

F. Configuration Parameters

Configuration parameters are used to tune the cloud resources for a service and are independent of any load- or work-related parameters. These parameters may, for example, represent sizes of software resources like pools or buffers. Formally, each service s has $\#cp$ configuration parameters:

$$cp_{s,d} = (a_{s,d,1}, \cdots, a_{s,d,\#cp}) \tag{11}$$

The optimal value for some configuration parameters may depend on the deployment configuration. Therefore, d is an index. For example, this is the case for the connection pool size in CloudStore. However, the allocation size for garbage collection does not depend on the deployment configuration. Also implementation-related parameters like version numbers should be specified here to increase reproducibility, especially for third party.

G. Formal Service Specification

When we combine all above, we get a formal specification for service s:

$$serv_s = (op_s, wp_s, lc_s, qt_s, vmg_{s,d}, cp_{s,d}, \delta) \tag{12}$$

We specify the operation mix, op_s, work parameters, wp_s, the load, lc_s, the quality thresholds, qt_k, the deployment for all the VM groups, $vmg_{s,d}$, and the configuration parameters, cp_s. The last term δ represents factors which we have not included in the model, but which still determine the scalability of a service. Some of these factors may be in control of a service consumer. Other factors may be outside of the control of the service consumer and manipulated by providers of cloud resources. A measure of δ is confidence intervals (see Table I).

III. Scalability Metric Functions

We first introduce resource space in Sec. III-A and capacity in Sec. III-B. The resource scalability metric function is described in Sec. III-C. The cost scalability metric function in Sec. III-E depends on cost as described in Sec. III-D.

Our usage of the term metric function $mf(x) = y$ is that it maps a set of deployment configurations x as input to a metric value y. We define the deployment configurations for two metric functions and a way how the y-values can empirically be obtained. For simplicity, we sometimes use "function" instead of "metric function".

A. Resource Space

The set of $\#rc$ cloud deployments which is explored for service s is the resource space. The resource space is defined using the cloud resources required by a service in Eq. 10:

$$rc_s = \{vmg_{s,1}, \cdots, vmg_{s,\#rc}\} \tag{13}$$

For our CloudStore measurements, the resource space is the type, size and number of instances used for the application VM group, and type and size for the DB VM group.

B. Capacity Metric

We do several clarifications when defining capacity: Firstly, we use a fixed operation mix so that we consider the load on the average operation. Secondly, for closed load we fix the think time. Thirdly, we fix the work parameters. The capacity becomes the highest load fulfilling the quality thresholds. More formally, the capacity $cap_{s,d}$ for a given service s with a given deployment d now becomes:

$$cap_{s,d} = f(op_s, wp_s, qt_s, vmg_{s,d}, cp_{s,d}, Z_s) \tag{14}$$

The *scalability range* for service s with a given resource space, is between the lowest and highest capacity in the resource space.

C. Resource Scalability Metric Function

The resource scalability of a service s is represented by corresponding capacities for *all* the deployments d in the resource space rc_s:

$$\forall d \in rc_s, [vmg_{s,d}, cap_{s,d}] \tag{15}$$

Normally, to limit the number of dimensions, the operation mix, the work parameters and the quality threshold are kept constant during a scalability analysis. Most configuration parameters are also fixed, but some depend on the deployment d. For sensitivity checks, operation mix, work parameters, and QTs can be altered for a limited subset of the resource space.

$vmg_{s,d}$ gives complete information of a deployment configuration d for service s. When $vmg_{s,d}$ can be represented by a single number, the capacities of the configurations in the resource space can be illustrated by a two-dimensional graph. This is shown in Fig. 3, where the resource space consists of $\{1, 2, 4, 6, 8, 10, 16, 32\}$ resource units. See Sec. VI-C for more discussion on the resource scalability metric function.

D. Cost

The cost of service s for a given configuration d, depends on the cost of the used VM instances in the $\#vmg$ VM groups:

$$c_{s,d} = \sum_{i=1}^{\#vmg} c_{inst}(v_d) \times inst_i(v_d) + c_{use}(serv_s) \tag{16}$$

In this equation, for each VM group i, $c_{inst}(v_d)$ returns the cost of a particular instance type and instance size in configuration d. $inst_i(v_d)$ returns the *number* of instances used for VM groups i in configuration d. Cost is then the sum of the costs for all VM instances in configuration d.

For PaaS and SaaS, cost usually depends on several elements in the service specification, $serv_s$, e.g. work, load or quality thresholds. This is specified by the term $c_{use}(serv_s)$. Cost can also depend on internal properties of the service, for example, number of payment operations required; factors which depend on $c_{use}(serv_s)$ in more or less complex ways. To account also for these factors in a PaaS or SaaS context, the only viable option may be to *measure* the cost. Note that the cost of using several instances is linear with respect to the number of instances used. There is also a clear relationship between the cost of different instance sizes. Looking at Amazon EC2 [9], the number of vCPUs (virtual CPUs) as well as

GB of memory double when we go to the next size. Also for the cost the relation is about the same. Instance storage and the number of connections do not share this simple pattern.

One drawback of using cost as a measure, is that it may vary because of cost modifications and regardless of changes in the instance types, sizes or number of instances. This drawback becomes smaller when we use *relative* and not *absolute* costs.

E. Cost Scalability Metric Function

In Sec. III-C, we discuss how the capacity of a given deployment configuration depends on specific values for operation mix, work parameters, configuration parameters and for quality thresholds. For cost scalability, we measure the capacities $cap_{s,d}$ and compute the costs $c_{s,d}$ for all the $\#rc$ configurations in the resource space rc:

$$\forall d \in rc_s, [c_{s,d}, cap_{s,d}] \tag{17}$$

When operation mix, work parameters and QTs are fixed, these tuples can then be presented in a figure as shown later in Fig. 4. In this figure cost is on the x-axis and capacity on the y-axis, but the axes may of course be reversed. Configurations that are too expensive relative to their delivered capacity can be excluded. We introduce eff representing the set of cost-efficient configurations for service s for the $\#rc$ configurations within the resource space rc:

$$\begin{aligned}\forall\, \alpha\, \in \mathit{eff}_s \, \{\neg \, \exists \, i \in \mathit{eff}_s \, | \\ [cap_{s,i} > cap_{s,\alpha}) \wedge (c_{s,i} < c_{s,\alpha})]\}\end{aligned} \tag{18}$$

Eq. 18 expresses that for a cost-efficient configuration α there does not exist another cost-efficient configuration i with a higher capacity and a lower cost.

IV. Measurement Plan

We now describe our scaling scenario for CloudStore. For AWS deployment, we describe three scenarios. The resource space for primary scenario using public AWS deployments is described in Sec. IV-A. We are also interested in the sensitivity of two selected work parameters and the quality threshold values and cover those in the secondary scenario described in Sec. IV-B as well as the tertiary scenario in Sec. IV-C. The resource space for these two scenarios is a condensed version of the resource space for the primary scenario. Sec IV-D describes our private OpenStack scenario. Finally, we describe detailed guiding principles for conducting the measurements in Sec IV-E

A. Primary AWS Scenario

We limit the resource space to present one comprehensive scenario of cloud deployment configurations. Several cloud resources exist, which are not included in our resource space. At the overall level, we focus on AWS and do not consider Microsoft or Google cloud services. At the next level, we use general purpose processing Amazon EC2, and not memory optimized instance types. We use the m3 instance type; the best general app VM instance when we started in 2014. We use the relation 1 m3.2xlarge = 2 m3.xlarge = 4 m3.large = 8 m3.medium which all have the same cost of \$ 0.585 ± 0.001 per hour [10]. Accordingly, we primarily use 1, 2, 4, or 8 instances. For database VM, we use the db.m3 instance type. We focus on the instance sizes large, xlarge as well as 2xlarge. To simplify naming we use 8.M:2XL for a configuration with 8 m3.medium app VMs and 1 db.2xlarge DB VM. The app VMs appear in front of the database back end VMs.

In our primary scenario, we are interested in the cost scalability using the work parameters $wp_{cs} = (10\,000, 288\,000)$ in Eq. 6, for number of books and customers, respectively. We use the quality thresholds in Eq. 8.

B. Secondary AWS Scenario

In this secondary scenario, we explore the sensitivity to changes both in the number of books as well as in the number of customers. We measure only a few different cloud configurations and not the full resource space. We focus on twice as many books: $wp_{cs} = (20\,000, 288\,000)$, as well as twice as many initial customers: $wp_{cs} = (10\,000, 576\,000)$.

C. Tertiary AWS Scenario

Quality thresholds in Eq. 7 may change as a result of altered customer preferences and market situation. A simple option is to scale all quality thresholds by the same amount by multiplying the quality metric vector with a given normalized quality threshold scaling constant, t, defined as:

$$t\vec{q}_s = (tq_{s,1}, \cdots, tq_{s,\#op}) \tag{19}$$

In this tertiary scenario, we use $t = 0.5$ so that all response time quality thresholds were halved. Similarly to the second AWS scenario, we will only investigate a few configurations in this tertiary scenario.

D. CloudStore Scenario on OpenStack

In a different deployment context, we use an OpenStack-based private cloud environment. Our objectives with these measurements are to estimate the impact of performance variability in a public context compared to a controlled cloud environment [11] and to demonstrate the reproducibility of the measurement method itself. The work parameters and QTs are identical to the AWS measurements in the primary scenario.

We conduct 10 measurement repetitions for each of two sizes for the app VM group (1) with 2 vCPU and 7.5 GB of RAM and (2) with 4 vCPU and 15 GB. For both configurations, we use one instance with 4 vCPUs and 15 GB for the database VM. In this way, we can look at the consequences of adding a more powerful app VM instances.

E. Detailed Measurement Principles

We follow an exploratory approach and try to get an overview with as few measurements as possible, both because they are costly, but also because the manual effort takes time. This is done by utilizing feedback during measurements, so that if an instance type and size clearly are not utilized, it does not make sense to add more instances of this type and size (horizontal scaling). We also do not have to explore the whole resource space, as it is most important to establish the upper bound on capacity regardless of the configuration. We also try to exploit the most expensive instance fully. For small configurations, this is clearly the DB VM instance. For larger configurations, it starts to be the app VM instance.

We already know that the cloud resources have significant performance variability [11]. Therefore, a very accurate and costly systematic measurement setup with binary search is not required. We use the faster linear search where we can roughly determine the capacity of a configuration with a single measurement. For some cost optimal measurements, we should do more measurements so that confidence intervals can be established. We stop repeating measurements if a 10% relative error is smaller than the confidence intervals, and we can be sure that a pair of configurations is likely to have different mean values by using Student-t-tests.

Since scalability (in contrast to elasticity behavior) is a steady-state measure, we measure with automatic provisioning (i.e. auto-scaling) turned off. To get reproducible measurements we strive for avoiding caching effects as far as possible. Deploying and filling the databases is therefore done for each new measurement.

V. Experiment Setup

This section describes our experiment setup for scaling CloudStore deployed on public AWS resources and on private OpenStack resources. We also describe the CloudStore architecture and distributed load generation.

A. CloudStore Architecture

Fig. 2 represents a simplified view of the relation between the services when CloudStore [4] is deployed on AWS. In this figure, there are two types of links between the services. Both links describe the use of services, but the filled link also describes deployment. We use a distributed version of the workload generator JMeter [12] to simulate the load of a given number of virtual CloudStore users. JMeter instances are deployed on EC2 in a different AWS availability zone than CloudStore or in case of OpenStack on individual hardware. AWS services S3 and CloudFront are used to obtain VM images.

CloudStore is implemented using the Spring framework, and deployed on Tomcat. Ngnix is used as a web proxy in front of Tomcat. The CloudStore app VM instances communicate with the MySQL database deployed on AWS RDS via Hibernate, using a JDBC driver. Both Tomcat and Ngnix are deployed on the same AWS EC2 instances. CloudStore has a payment service and for this, we have made a response time generator. This response time generator [13] is deployed on the PaaS Heroku [14].

CloudStore is deployed on OpenStack with the same software and configurations as we use on AWS. A MySQL database is installed and configured in an OpenStack VM. Instead of using CloudFront and S3 services, we hosted static content (images) on Chef, an OpenStack solution for object storage. The Payment service is deployed on Heroku.

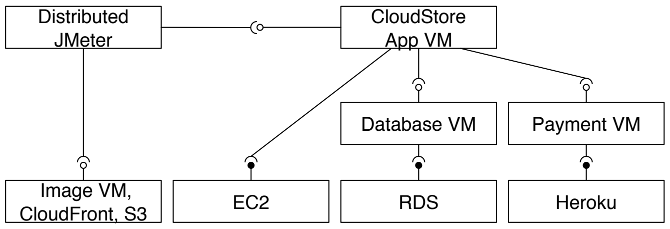

Fig. 2. CloudStore deployed on AWS.

B. Configuration Parameter Values

There is a limit to the number of concurrent connections for each database instance. For db.m3.large the default value is 600, for db.m3.xlarge it is 1 200 and for db.m3.2xlarge it is 2 400 connections. For best performance, these connections should be evenly distributed to the app VM instances. Accordingly, this parameter depends on the deployment configuration. Maximum number of connections in each app VM instance is specified in maxPoolSize in Hibernate.

C. Load Generation

We used JMeter [12] as a load generator, and evaluated configurations one by one. For finding the capacity limit we used linearly increasing load for 16 minutes. Depending on the expected capacity, we configure JMeter to reach between 5 000 and 20 000 virtual users. The capacity was defined as the load just before we experienced the first SLO violations.

Assume JMeter was configured to reach a maximum load of 14 000 virtual users after 16 minutes. Further assume that the last minute satisfying the SLO was the minute between 12 and 13 minutes with an average of $12.5/16 * 14\,000 = 10\,938$ virtual users (VU). With 7 seconds think time (and zero response times) the maximum number of requests at 12.5 minute was:

$$\#\text{requests} = \frac{14\,000 * 60}{7} \frac{12.5}{16} = 93\,750 \qquad (20)$$

In some measurements, a marginally higher load might again satisfy the SLO, but this was ignored. This approach rests on the assumption that the capacity with a linearly increasing load is the same as the capacity with a stable load. This assumption is confirmed by our own measurements with stable load, as well as by Kossmann et al. [15]. With this assumption, we also consider CloudStore to be a closed system, even though we linearly increase the load.

VI. Results

Actual measurements from the CloudStore [4] implementation are used to illustrate our scalability functions. This section describes the results and discusses them, before outlining the time required to conduct these measurements.

A. Results on Amazon

Table I shows the 49 CloudStore measurements on Amazon with 10 000 books and 288 000 customers described in Eq. 6 as well as with normal quality thresholds, shown in Eq. 8. The first column characterizes the deployment, initially with

TABLE I
AMAZON MEASUREMENTS ORDERED AFTER COST.

Instances	Cost	Cap.	#	Conf.	U_{App}	U_{DB}
M:L	0.27	563	1		75	6
2.M:L	0.35	1 125	1		85	14
3.M:L	0.42	1 625	1		68	17
M:XL	0.47	563	1		87	3
2.M:XL	0.55	1 125	1		71	5
4.M:XL	0.69	2 250	1		74	12
XL:XL	0.69	3 406	6	± 155	66	18
6.M:XL	0.84	3 438	1		75	20
M:2XL	0.86	563	1		76	2
2.M:2XL	0.93	1 094	1		69	3
8.M:XL	0.98	5 313	1		93	33
2.XL:XL	0.99	9 313	4	± 1 021	89	55
4.M:2XL	1.08	938	1		99	3
6.M:2XL	1.22	3 500	1		81	10
8.M:2XL	1.37	4 500	1		80	14
2XL:2XL	1.37	6 219	4	± 450	58	15
10.M:2XL	1.52	5 500	1		66	15
4.XL:XL	1.57	10 302	6	± 582	44	71
4.XL:2XL	1.96	10 938	1		21	69
2.2XL:2XL	1.96	13 039	8	± 764	56	38
4.2XL:2XL	3.13	11 709	6	± 1 102	25	58

the number of application instances, then with the application size (M, L, XL, 2XL) and finally with the database size (L, XL, 2XL), e.g., 2.2XL:2XL has two m3.2xlarge app instances as well as one db.m3.2xlarge DB instance. The second column is the overall cost for the app VM instances and the database VM instance. All costs were recorded in February 2017 for Ireland with MySQL, and are measured in \$ per hour [10]. Capacity in column three is measured in terms of maximum number of virtual users (VU) which fulfill the quality thresholds for all the CloudStore operations. The sixth and seventh columns are the average % of CPU utilization for the app and the DB, corresponding to the measured capacity.

To get a feeling for confidence intervals, some of the measurements in Table I were repeated. In this case the values in column three Table I represents the average capacity. The number of repetitions is shown in column four and two-sided confidence intervals for the repeated measurements in column five. We use 90% confidence intervals and Student *t*-distribution, e.g., with 90% probability, the capacity for configuration 2.2XL:2XL is between 12 275 and 13 803.

With work parameter values of $wp_{cs} = (10\,000, 288\,000)$, used until now, configuration 4.XL:XL has a capacity of 10 302. In our secondary scenario, we double the number of books to $wp_{cs} = (20\,000, 288\,000)$ and the capacity reduces to 2 250 VU. When doubling the number of customers to $wp_{cs} = (10\,000, 576\,000)$, capacity becomes 4 500 VU. In both these cases, we used two measurements, so that we in total got $49 + 2 + 2 = 53$ CloudStore measurements on AWS.

In our tertiary scenario, we halve the quality thresholds (see Eq. 19) for all operations for the two configurations XL:XL and 4.2XL:2XL. In both configurations, we used the original work parameters $wp_{cs} = (10\,000, 288\,000)$. As a result, the capacity did not change. This tertiary scenario did not require new measurements, just recalculation of old measurements.

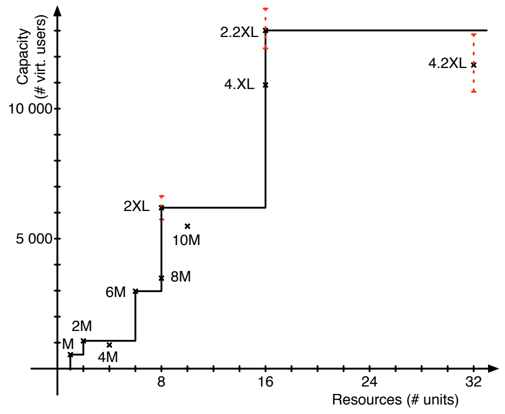

Fig. 3. Resource scalability for some CloudStore configurations.

B. Results on OpenStack

On OpenStack, we did 10 measurements for two different configurations as described in Sec. IV-D, and computed 90% confidence intervals. With a 2 vCPU, capacity was 1488 ± 61. App and DB VM group's average utilization were $6\% \pm 0.24\%$ and $48\% \pm 1.29\%$, respectively. With 4 vCPU, capacity was 1563 ± 102. App and DB VM group's average utilization were $6\% \pm 0.39\%$ and $22\% \pm 0.51\%$, respectively.

C. Resource Scalability

Fig. 3 plots resources on the x-axis versus capacity on the y-axis for the 10 configurations with db.m3.2xlarge as a database VM. For the app VM instances, we use the relation 1 m3.-2xlarge = 2 m3.xlarge = 4 m3.large = 8 m3.medium which all have the same cost of \$ 0.585 ± 0.001 per hour [10]. The x-axis in Fig. 3 represents the number of m3.medium. In two cases, we have two competing configurations with the same cost, namely 8.M and 2XL both with 8 app resources, and 4.XL and 2.2XL with 16 app resources. In both cases, it seems like the largest app instance has the best capacity.

The continuous line in Fig. 3 shows the maximum capacity for a given number of base unit resources. With a larger resource space, either because of more AWS configurations or because of configurations from other vendors, we would have got more dots in this figure. As a result, the graph could also change; not because of a change in CloudStore, but because of a more fine-grained procedure for evaluating the empirical function.

In line with Eq. 18, Fig. 3 also shows how configurations below the horizontal line are not cost-effective, i.e., configuration 4M, 8M, 10M, 4:XL and 4.2XL.

In this example, there is only one type of resource unit, namely the size of app VM instances. If we also included a DB (database) VM instance, we get two types of resource units, giving a three-dimensional graph, where the number of resources for these two resource units becomes the x- and y-axis and where the z-axis becomes their corresponding capacity. Another solution is to normalize the size of these resources [16]. We can then still use a two-dimensional graph.

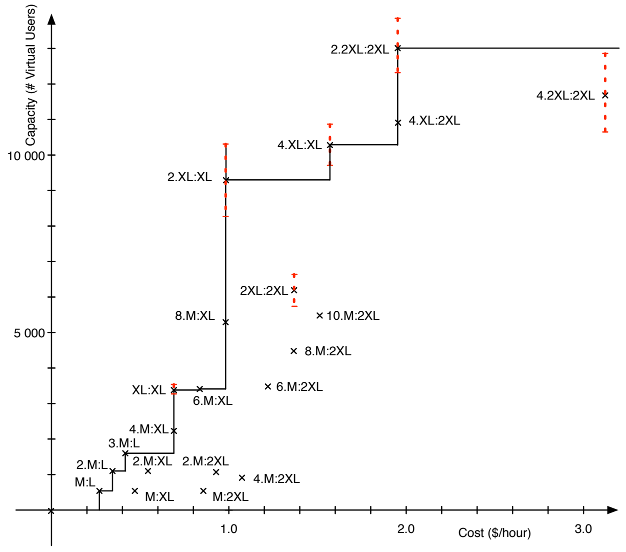

Fig. 4. Cost Scalability for all CloudStore configurations

In cloud computing, cost is an obvious normalizing factor, and in this case, the combined cost for both resource types could be the x-axis while the capacity is the y-axis. We refer to this as cost scalability, which is illustrated in the next section.

D. Cost Scalability

Based on the tuples in Eq. 17, Fig. 4 plots the cost on the x-axis versus the capacity on the y-axis for all the 21 CloudStore AWS configurations shown in Table I. Like in Fig. 3, the continuous line in Fig. 4 shows the maximum capacity, but now for a given cost. Again, we see that some configurations are not cost-effective, e.g., configuration M:2XL. Based on Fig. 4, we can also get a feeling for how linear scalability is for certain capacity ranges. For example, CloudStore scales approximately linearly up to 13 000 virtual users, with one exception: we see that configuration 2.XL:XL is particularly good. For a cost of $0.99 per hour, it delivers a capacity of 9 313 virtual users. This gives a cost of 0.0106 cent per virtual user per hour.

E. Discussion

In Table I, several measurements are limited by a saturated CPU in the app VM instances. An expensive, but poorly utilized database VM does not contribute to a cost-efficient configuration, e.g., compare configuration M:2XL to 6.M:XL, where the latter costs less but delivers a far better capacity.

Confidence intervals are relatively broader for CPU utilizations compared to capacities. When we increase work parameters, each request of an operation demands more resources to be processed. Increasing the two work parameters, for books and customers, have a drastic effect on capacity, especially for the number of books where capacity drops to one fifth. The results for halving of the response times indicate that when we hit the capacity limit, then queues are building up rapidly, especially when we use the strongest configuration 4.2XL:2XL, close to CloudStore's maximum capacity.

According to our measurements, the capacity of CloudStore is limited to 13 039 virtual users for configuration 2.2XL:2XL. In this configuration neither the (average) app nor DB VM CPU seems to be a bottleneck. The (average) app VM instance CPU has the highest utilization, but adding more app VM instances in configuration 4.2XL:2XL actually reduces the capacity, even though the CPU utilization is approximately halved. Based on the repeated measurements, a Student-t-Test results in a probability of 96% for the hypothesis that configuration 4.2XL:2XL to exhibits smaller capacity than configuration 2.2XL:2XL.

We would like to clarify that the capacity limit in the CloudStore implementation is the result of its use of underlying AWS resources and not a result of shortcomings in the measurement set up itself. We tried to tune several Tomcat, Ngnix and Linux network configuration parameters as well as connection pooling parameters in both the app and DB VM instances. None of these parameters resulted in a higher capacity, although some of the parameters affected some measurements. This points us to an inherent scalability problem in the relational database. The number of RDS connections could be ruled out since doubling them using 2xlarge instead of xlarge as the RDS instance, does not significantly increase the number of virtual users. We see from the MySQL slow query log that when we reach the capacity, the query response time increases dramatically. Possible causes could be database locking issues, Hibernate transaction handling or RDS/MySQL configuration issues like number of writer threads.

F. Time to Run the Experiments

The time required to experiment, including starting the CloudStore application and distributed JMeter, depends on: 1) number of app VM instances, 2) number of virtual users to be simulated by JMeter instances, 3) the number of books and customers in the database as well as 4) the scenario duration. Deploying and filling the database for CloudStore with $wp_{cs} = (10\,000, 288\,000)$ and with four app VM instances take approximately 30 minutes. To avoid caching effects, deploying and filling the databases must be done for each new measurement. After this, it takes approximately 20 minutes to deploy the distributed JMeter on eight instances of m3.xlarge (each JMeter instance with m3.xlarge can handle approximately 2 000 virtual users). In addition, we linearly increase the load during 16 minutes. We also need time to clean up and generate graphs. This is automated using a script, and does not take more than a few minutes. In summary, one experiment takes approximately five quarters of an hour.

VII. Related Work

Our two scalability metric functions center around capacity, and evaluate how capacity changes as a result of altering the cloud resources. This understanding of scalability is supported by the scalability definition in the systematic literature review by Lehrig et al. [1]: *"Scalability is the ability of a cloud* *layer to increase its capacity by expanding its quantity of consumed lower-layer services."* Our two scalability functions operationalize and detail Lehrig's scalability definition. The functions relate scalability to work, load, quality thresholds and resource space.

Lehrig lists Tsai et al. [17] as the only validated scalability metric for cloud computing. Our cost-related scalability function is basically similar to the metric by Tsai et al. However, we consider different operations with individual quality thresholds explicitly, and also differentiate between work and load. Moreover, our way of presenting cost scalability in Fig. 4 has not been done by Tsai. Our resource scalability function as well as our concept of resource space are new.

Bondi's [18] concept of load scalability is close to our scalability functions, but ours are richer since we explicitly consider work, load, quality thresholds as well as configuration variables and deployments. We also apply our scalability functions in a cloud computing context where cost is explicit.

Kossmann, Kraksa, and Loesing [15] compare the capacity for a TPC-W application deployed on three different cloud services: Amazon AWS, Google AppEngine, and Microsoft Azure. As the number of users increase they observe throughput that satisfies the quality thresholds. In contrast to our work, they do not scale work parameters or quality thresholds. Their focus is on the database VM group, while the app VM group is not explicit. This means that they do not consider how the capacity varies when the number of app VMs is scaled.

VIII. Conclusions, Limitations & Further Work

The results demonstrate that our proposed scalability metric functions allow detailed insights like identification of cost-efficient resource configurations for a given capacity. We demonstrate that repeated measurements allow for computation of confidence intervals with a reasonable size that underline differences between configurations.

CloudStore is a typical, three-tier session-based enterprise application, and our equations capture the essence of scalability for this type of applications. Other types of systems may have further factors influencing scalability.

According to Erl [2], service quality metrics should be quantifiable, repeatable, comparable and easily obtainable. We claim that both the resource scalability as well as the cost scalability metric functions fulfill these characteristics as they are based on capacity as defined earlier, assuming a properly defined SLO. Concerning how easy these scalability metric functions are to obtain, Sec. VI-F gives indications.

Our feedback-driven explorative measurement process is not specified formally. We scale the bottleneck resources first. However, identifying new bottleneck resources usually require human interpretation. In a new configuration, configuration parameters shall ideally be tuned again. Manually, this is impossible, due to the large effort involved. Similarly, it is also a question if we have selected the best cloud resources in our resource space, since exploring all cloud resources is not feasible time- and cost-wise.

Automated resource space exploration can give more measurements without increasing the manual work and may also simplify finding optimal values of the plentiful configuration parameters.

We focus on empirical scalability evaluation in this article, but the scalability concepts are general and also applicable for an analytical or model-based simulation approach that may enable early architectural decision. With a model of the deployed application, we are then no longer limited to cloud resources that are actually deployed and measured. Resource space exploration may also be automated.

Finally, a conceptually consistent specification of the influencing factors for scalability is useful, e.g. for handling scalability in agile development [8].

IX. Acknowledgments

The research leading to these results has received funding from the Norwegian Research Council under grant #256669 (ScrumScale) and EU under grant #317704 (CloudScale), as well as supported by the SPEC RG [19]. Kjetil Haslum and Brynjar Viken investigated the configuration parameters to ascertain that measurements could be reproduced. Richard Sanders and Georg Ræder contributed with proofreading.

References

[1] S. Lehrig, H. Eikerling, and S. Becker, "Scalability, Elasticity and Efficiency in Cloud Computing: a Systematic Literature Review of Definitions and Metrics," in *Quality of SW Arch.* ACM, 2015.
[2] T. Erl, Z. Mahmood, and R. Puttini, *Cloud Computing: Concepts, Technology & Architecture*. Prentice Hall, 2013.
[3] G. Brataas *et al.*, "CloudScale: Scalability Management for Cloud Systems," in *ICPE, Conf. on Performance Eng.* ACM, 2013.
[4] CloudScale, "CloudStore," https://github.com/CloudScale-Project/CloudStore, visited: 21 Feb 2017.
[5] T. P. Council, "TPC-W benchmark (web commerce) specification version 1.8," Original link http://www.tpc.org/tpcw/spec/tpcw_V1.8.pdf now at http://www.pdf-archive.com/2016/05/20/tpcw-v1-8/, Feb. 2002, last visited: 21 Feb 2017.
[6] Amazon, "AWS," aws.amazon.com, visited: 21 Feb 2017.
[7] OpenStack, "Open Source Software for Creating Private and Public clouds," www.openstack.org, visited: 21 Feb 2017.
[8] G. Brataas and T. E. Fægri, "Agile Scalability Requirements," in *ICPE, Conf. on Performance Eng.* ACM, 2017.
[9] Amazon, "EC2 Instance Types," https://aws.amazon.com/ec2/instance-types, last visited: 21 Feb 2017.
[10] ——, "Amazon EC2 Pricing," https://aws.amazon.com/ec2/pricing/on-demand, last visited: 21 Feb 2017.
[11] A. Iosup, N. Yigitbasi, and D. Epema, "On the Performance Variability of Production Cloud Services," in *CCGrid 2011*, 2011, pp. 104–113.
[12] The Apache Software Foundation, "Apache JMeter," jmeter.apache.org, 2014, last visited: 21 Feb 2017.
[13] CloudScale, "Response Generator," https://github.com/CloudScale-Project/Response-Generator, visited: 21 Feb 2017.
[14] Heroku, https://www.heroku.com/, visited: 21 Feb 2017.
[15] D. Kossmann, T. Kraska, and S. Loesing, "An Evaluation of Alternative Architectures for Transaction Processing in the Cloud," in *Conf. on Management of data*. ACM, 2010, pp. 579 – 590.
[16] G. Brataas, P. H. Hughes, J.-A. Fagerli, and O. C. Landmark, "Exploring Architectural Scalability," *Software Engineering Notes*, vol. 29, no. 1, pp. 125 – 129, 2004.
[17] W.-T. Tsai, Y. Huang, and Q. Shao, "Testing the Scalability of SaaS Applications," in *SOCA*. IEEE, 2011.
[18] A. B. Bondi, *Foundations of Software and System Performance Engineering*. Addison Wesley, 2015.
[19] SPEC, "Research Group," http://research.spec.org, visited: 21 Feb 2017.