Investigating Performance Metrics for Scaling Microservices in CloudIoT-Environments

Manuel Gotin
Robert Bosch GmbH
Renningen, Germany
manuel.gotin@de.bosch.com

Felix Lösch
Robert Bosch GmbH
Renningen, Germany
felix.loesch@de.bosch.com

Robert Heinrich
Karlsruhe Institute of Technology
Karlsruhe, Germany
robert.heinrich@kit.edu

Ralf Reussner
Karlsruhe Institute of Technology
Karlsruhe, Germany
ralf.reussner@kit.edu

ABSTRACT

A CloudIoT solution typically connects thousands of IoT things with cloud applications in order to store or process sensor data. In this environment, the cloud applications often consist of microservices which are connected to each other via message queues and must reliably handle a large number of messages produced by the IoT things. The state of a message queue in such a system can be a challenge if the rate of incoming messages continuously exceeds the rate of outgoing messages. This can lead to performance and reliability degradations due to overloaded queues and result in the unavailability of the cloud application.

In this paper we present a case study to investigate which performance metrics to be used by a threshold-based auto-scaler for scaling consuming microservices of a message queue in order to prevent overloaded queues and to avoid SLA violations. We evaluate the suitability of each metric for scaling I/O-intensive and compute-intensive microservices with constant and varying characteristics, such as service time. We show, that scaling decisions based on message queue metrics are much more resilient to microservice characteristics variations. In this case, relying on the CPU utilization may result in massive overprovisioning or no scaling decision at all which could lead to an overloaded queue and SLA violations. We underline the benefits of using message queue metrics for scaling decisions instead of the more traditional CPU utilization particularly for I/O-intensive microservices due to the vulnerability to variations in the microservice characteristics.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → *Software performance*; *Software reliability*;

KEYWORDS

Cloud Computing, Internet of Things (IoT), Microservices, Message Queues, Performance Metrics, Auto-Scaler, Threshold-based rules, Performance

ACM Reference Format:
Manuel Gotin, Felix Lösch, Robert Heinrich, and Ralf Reussner. 2018. Investigating Performance Metrics for Scaling Microservices in CloudIoT-Environments. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3184407.3184430

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184430

1 INTRODUCTION

The uprising `CloudIoT` paradigm addresses the limitations of IoT by merging it with a cloud infrastructure to provide virtually unlimited computational and storage capabilities [4]. Bosch offers with the Bosch IoT Suite a Platform-as-a-Service (PaaS) to provide a toolbox to quickly build IoT applications and deploy them on the Bosch IoT Cloud which is based on Pivotal CloudFoundry. A common usage scenario in this environment is Sensing-as-a-Service (SaaS) which describes the process of making sensor data available to clients and applications over the cloud infrastructure [20]. Sensor data arise from a range of domains like mobility to smart home and result in a huge amount of data to be processed and stored on the cloud.

In recent years the focus in software industry has shifted from monolithic architectures to the microservice architectural style [2]. This architectural style allows to leverage the capabilities of cloud computing in terms of scalability and maintainability. Whereas traditional applications exhibit a monolithic architecture which tends to put multiple functionality into a single process the microservice architectural style separates functionalities into self-contained services. Breaking down software to loosely coupled and highly cohesive modules offers multiple benefits in terms of flexibility and evolvability [7]. By supporting scaling operations on a fine-granular level infrastructure costs can be reduced up to 70 % compared to a traditional monolithic architecture [25].

Microservices are typically connected to each other via a lightweight communication protocol, most commonly REST or message queues [9]. In this paper we focus on the communication via message queues provided by a message broker system. There are many message broker systems which differ in their mechanism and feature set and a short survey describing the most popular message broker systems can be found in [13]. The state of a message queue

can be a challenge in such a system if the rate of incoming messages continuously exceeds the rate of outgoing messages eventually resulting in performance and reliability degradations. Due to the accumulation of messages in the queue these issues can persist even after the rate of incoming and outgoing messages is in balance again. This underlines the need to reflect the message queue state in scaling decisions.

The Bosch IoT Cloud offers a built-in threshold-based auto-scaling mechanism to scale microservices in and out based on the CPU utilization, HTTP latency or HTTP throughput. In this paper we investigate the suitability of the CPU utilization for scaling microservices which are consuming messages from a message queue. Furthermore we investigate which information of the state of the message queue is suitable for scaling consuming microservices. We focus on two classes of microservices: I/O-intensive and compute-intensive. In the first class the time of processing a task is determined by waiting for I/O-operations to be completed whereas in the second class it is determined by the computation power. We investigate the suitability of each metric to cope with the challenges in this environment using a threshold-based rules auto-scaling setup. This allows us to use a single performance metric and compare it to other metrics in coping with the challenges in respect to each microservice class. Subsequently we explore the vulnerability of each metric to variations in the microservice characteristics.

The remainder of this paper is organized as follows: Section 2 describes a running example which serves as a `CloudIoT` application for the pre-processing and storing part of a SaaS use case. Section 3 explores the challenges in scaling consuming microservices. The case study in section 4 evaluates the suitability of each performance metric using a threshold-based auto-scaling system. Section 5 gives a brief overview of related work. Section 6 concludes the results extracted from the case study.

2 RUNNING EXAMPLE

One of the most common usage scenarios of the Bosch IoT Suite is SaaS in the areas of mobility, industry 4.0 or smart home. The main idea is to store data from things on the cloud and make it available to clients and applications.

In the running example we examine the pre-processing and storing of sensor data on the cloud. Such a system needs to cope with a high amount of data. An architecture to support this use case was proposed by Cecchinel et al. [5]. In order to allow a more fine-granular scaling we refine the proposed architecture. Instead of processing and persisting sensor data within a single service we propose a dedicated service for each of these functionalities to be more in line with the microservice architectural style. The services are connected to each other via message queues provided by a message broker system. By using message queues consuming and producing services are decoupled from each other and communication is asynchronous. Figure 1 illustrates the components in this architecture.

Components – The **Connection Service** serves as a gateway for connecting IoT things with the cloud. Typically it retrieves sensor data via a lightweight communication protocol, e. g. REST or MQTT. This sensor data is then enqueued in a message queue. As a consuming service the **Data Processing Service** retrieves messages

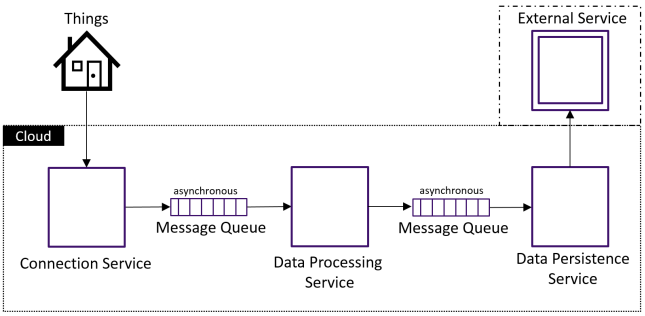

Figure 1: Illustration of the running example.

from this queue and pre-processes them. Pre-processed messages are enqueued towards the **Data Persistence Service**. This service retrieves processed messages and stores them by communicating with an **External Service**, i. e. a database management system via a RESTful API.

Constraints – Communication with the external service is synchronous. To avoid losing sensor data, a consuming service must acknowledge a message as processed before the message queue releases it. In case of a service failure an unacknowledged message can be retransmitted to a healthy service instance.

Microservice classes – We classify the **Data Processing Service** as a compute-intensive microservice, since the processing of messages is in this example a mainly CPU-bound operation. We classify the **Data Persistence Service** as an I/O-intensive microservice since the time for processing a message is determined by communicating with the external service.

Environment – We deploy the running example on the Bosch IoT Cloud which is based on Pivotal CloudFoundry. CloudFoundry abstracts the underlying infrastructure and offers an interface to deploy and manage applications with a customizable runtime environment. For this reason an application developer scales application instances instead of virtual machines. In our environment an application instance of a lightweight microservice – like these in the running example – is provisioned in less than one minute. Furthermore CloudFoundry offers self-healing capabilities by discovering runtime failures of application instances and restarting them. As a message broker system we use Pivotal RabbitMQ. It is an implementation of the AMQP protocol which was initially designed for financial transactions thus aiming for reliability and scalability. In the most common scenario RabbitMQ enqueues and dequeues messages in a first-come, first-served (FCFS) manner [1].

3 CHALLENGES IN SCALING CONSUMING MICROSERVICES

In this section we address specific challenges for scaling microservices consuming messages from message queues. The challenges are motivated by issues we experienced on the messaging middleware caused by an accumulation of messages in message queues due to underprovisioned microservices on the consuming side.

Let q be a queue, l the number of messages in the queue, p the production rate and c the consumption rate. Let the service policy

of q be first-come, first-served (FCFS). This queue is illustrated in Figure 2.

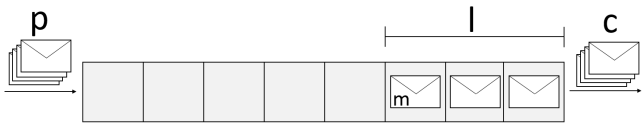

Figure 2: Illustration of a message queue.

There are three basic states for a queue q based on the growth $\Delta l = p - c$:

- **Steady**: A queue q is in a steady state if $\Delta l = 0$
- **Filling**: A queue q is in a filling state if $\Delta l > 0$
- **Draining**: A queue q is in a draining state if $\Delta l < 0$

A filling state leads to the accumulation of messages in the queue. Since the policy is FCFS they induce a delay on application-layer caused by a wait time for each message in the queue. Let $t_q(m)$ be the delay for a message m to pass through a queue q and T_{max} the maximal desired delay, e. g. derived using the applications SLA. Let L_{max} be the maximal length of a queue q the message broker system is able to cope with. We define the following conditions:

- **Congested**: A queue q is considered as congested if $t_q(m) > T_{max}$
- **Flooded**: A queue q is considered as flooded if $l > L_{max}$

A congested queue degrades the performance on application-layer since each message experiences a delay. In a flooded state a message broker system may try to stabilize the queue by blocking and unblocking the connection to keep the rate of incoming messages at a level the consumers can handle. In RabbitMQ this behavior is called **flow control mode**. On application-layer a flooded queue induces like a congested queue a delay but furthermore leads to a degradation of reliability by rejecting messages. A congested or flooded queue may remain after a queue has been stabilized from a filling state to a steady state. For this reason we identify the following challenge:

- **Challenge I** – Recover or avoid flooded or congested queues.

The underlying issue of a congested or flooded queue is based on the provisioning of consuming microservices. Underprovisioned microservices lead to a filling queue state since the consumption rate is lower than the production rate. For this reason it eventually transits to a congested or flooded state inducing a performance degradation on application-layer and may result in reliability issues such as a rejection of messages. Overprovisioned microservices have a low utilization but do not degrade the message queue state since the consumption rate exceeds the production rate. However, due to the typical pay-as-you-go cost model, each provisioned resource increases the operating costs.

Since the microservices are consuming messages from the message queue, information about the state of the message queue could be beneficial if utilized for scaling decisions. For this reason we identify the following challenges regarding microservices in respect to the message queue:

- **Challenge II** – How to utilize informations of the state of the message queue to prevent underprovisioning of consuming microservices?
- **Challenge III** – How to utilize informations of the state of the message queue to prevent overprovisioning of consuming microservices?

4 CASE STUDY

The rationale of this case study is to investigate the suitability of a set of performance metrics in a threshold-based rules auto-scaling setup for different microservices classes to cope with the challenges which were mentioned in section 3. Furthermore we investigate how vulnerable an auto-scaling system is to variations in the microservice characteristics in terms of elasticity and the message queues state in respect to each performance metric.

Threshold-based rules auto-scaling. This class of auto-scaling is one of the most common strategies to address under- and overprovisioning in cloud environments. It exhibits a widespread use in industry due to the simplicity and high availability among commercial cloud providers like Amazon EC2. Rules in this context consist of a condition and an action to be executed. Usually they define a lower and upper threshold for a performance metric. If the current value of the metric exceeds a threshold, the auto-scaling systems scales application instances in or out. The quality of scaling decisions of an auto-scaling systems can be evaluated via its elasticity. Elasticity in this context describes the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources [12].

Performance metrics. A set of performance metrics for scaling decisions is listed in the survey [15]. The internal state of the message broker system poses a challenge in the running example. For this reason we want to compare the suitability of relying directly on message queue metrics for scaling decisions instead of the traditional CPU metric. Many message brokers such as RabbitMQ support the monitoring of queue-specific metrics like arrival rate (ingress), departure rate (egress) and queue length. In order to approximate the queueing delay we measure the end-to-end latency between message transmission and receiving in a consuming microservice. Arrival rate and departure rate are conceptually not viable for scaling decisions in a threshold-based rules auto-scaling setup. For example: the arrival rate is not influenced by scaling decision thus offers no feedback. For this reason we investigate the queue growth which includes both metrics. The following list gives an overview over the set of investigated metrics:

- **Microservice – CPU:** The average CPU utilization is a popular proxy of the current systems workload.
- **Message Queue – Length:** The queue length describes the number of enqueued messages.
- **Message Queue – Growth:** The queue growth is the difference between arrival and departure rate thus describing the current growth in the queue.
- **Message Queue - Delay:** The queueing delay describes the wait time for a message in the queue before being processed.

4.1 Research Questions

We propose the following research questions to address the suitability of performance metrics to represent the systems workloads in a manner which allows an auto-scaling system to cope with the described challenges:

- **RQ1** – What degree of elasticity achieves a threshold-based rules auto-scaling system for each performance metric and each microservice class?
- **RQ2** – How suitable is such an auto-scaling system in avoiding a congested or flooded message queue?

In a dynamic environment like CloudIoT changes or variations in the characteristics of microservices can be expected. For example: the I/O-intensive microservice exhibits a dependency to an external service, how does an already configured auto-scaling system cope with changed external service times? How well does an already configured auto-scaling system cope with variations of the compute-intensive tasks?

- **RQ3** – How vulnerable is a configured threshold to variations in the microservice characteristics for each microservice class in respect to the auto-scalers elasticity?
- **RQ4** – How vulnerable is a configured threshold to variations in the microservice characteristics for each microservice class in respect to avoiding a congested or flooded message queue?

4.2 Methodology

We use a synthetic setup with a threshold-based rules auto-scaling system to perform horizontal scaling operations based on a single performance metric for different microservice classes which are described in the running example in section 2. In this synthetic setup we can directly set the characteristics of the external service and the configuration of the internal services. In order to evaluate each metric for the specific microservice class we investigate the auto-scaling system as a whole since it relies on the metric and its thresholds to trigger actions. To answer the research questions we need to evaluate the elasticity of the auto-scaler and the state of the message queue.

Evaluate scaling decisions. In order to qualify elastic adaptations we apply the elastic speedup measure proposed by SPEC RG Cloud [11]. The elastic speedup measure reflects the difference between supplied and demanded resources within the measurement period regarding timing and accuracy aspects. Whereas the timing aspects are expressed by the share of time in an under- or overprovisioned state the accuracy describes the absolute deviation of each state in respect to the demanded resources. Both aspects are normalized over the measurement period and each aspect is aggregated to a single *accuracy* and *timeshare* metric using a custom weight for under- and overprovisioning. The elastic speedup measure is based on a speedup vector s_k for a benchmarked platform k. The speedup vector s_k is computed with the accuracy and timing aspects of the benchmarked platform k and a baseline platform *base*:

$$s_k = \left(\frac{accuracy_{base}}{accuracy_k}, \frac{timeshare_{base}}{timeshare_k}\right) = (accuracy, timeshare)$$

The elastic speedup measure for a benchmarked platform k is the geometric mean of its speedup vector s_k:

$$elasticspeedupmeasure = \sqrt{s_{k_{accuracy}} + s_{k_{timeshare}}}$$

In our test setup we weight all metrics equally. To obtain the baseline metrics we execute a configured workload on the system using no auto-scaling system. The baseline is used to compute the elastic speedup measure for each configuration of the auto-scaling system in identical workload setups. In this paper we refer to the elastic speedup measure as the elasticity score.

Evaluate queue state. In order to evaluate the suitability of avoiding a flooded or congested message queue we compare the average queue length for each setup run. A queue length aiming towards zero indicates that the auto-scaling mechanism based on the performance metric is suitable to avoid a flooded or congested queue.

Threshold optimization. Threshold-based rules auto-scaling systems rely on an upper and lower threshold for deciding scaling operations. The performance of a threshold-based auto-scaling system depends on the configured threshold. An application developer has degrees of freedom to configure the thresholds to achieve a specific goal regarding reaction speed, costs or performance. We optimize the thresholds for each metric with a heuristic algorithm using the achieved elasticity as fitness function to capture the accuracy and timing aspects of scaling decisions. By optimizing the thresholds for each performance metric we can compare them to each other. The concrete steps for the compute-intensive and I/O-intensive microservices are as follows:

(1) Configure the microservice characteristics for a specific setup.
(2) Configure the workload and execute it on the system without using an auto-scaler to define the baseline score.
(3) For each metric approximate the optimal threshold constellation. In this phase we apply differential evolution (DE) as a heuristic to find the global minima [23]. As fitness function we use the elasticity score.
(4) Compare the results of each optimal threshold metric run in order to quantify the suitability of each metric.

Variations in microservice characteristics. To address research question 3 and 4 we configure the characteristics of each microservice class. For the compute-intensive microservice we variate the computational operation to require more or less processing steps in order to simulate a change in task. For the I/O-intensive microservice we vary the service time of the external service ST_{ext} and investigate the influence of the transferred payload N_{data}.

4.2.1 Auto-Scaling system. The setup of the auto-scaling system is illustrated in Figure 3. The scaling system monitors a single microservice and uses the CPU or message queue metrics as underlying performance metrics for scaling decisions. In this setup we scale application instances in and out (horizontal scaling). After a scaling decision the system enters a cooldown-period which ends after the scaling decision has actually an impact, i. e. the scaled application instance is destroyed or ready, which occurs usually within 20-40 seconds.

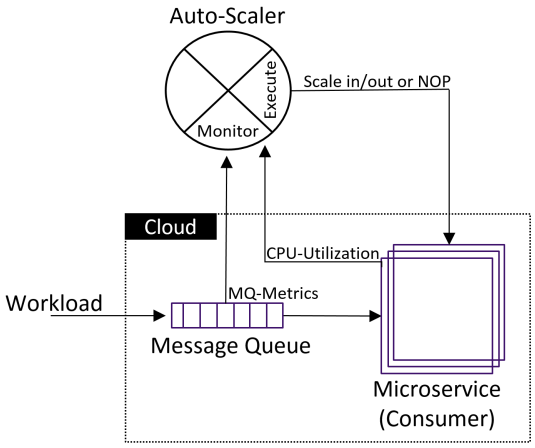

Figure 3: Illustration of the threshold-based rules auto-scaling system

Metric	Threshold	Avg. Q.-Length	Score
CPU	60 / 100 [%]	1.76 [msg]	305.94
Queue Delay	0.75 / 1.25 [sec]	3.17 [msg]	259.13
Queue Growth	0 / 1 [msgs/sec]	95.96 [msg]	247.16
Queue Length	0 / 10 [msgs]	8.41 [msg]	221.01
Baseline	- / -	-	100

Table 1: Elasticity score for each performance metric adapting the compute-intensive microservice

The workload consists of three phases to observe the systems behavior for an increasing, steady and decreasing workload intensity. We send a total of 4000 messages over a duration of 10 minutes. We configure both microservices to process a message in circa 400 ms.

4.3 Results

In the first test setup we optimize the thresholds of each performance metric for a specific microservice characteristic.

Compute-intensive microservice. Table 1 shows the ranking of each metric in respect to the elasticity score and also shows the average queue length in order to address the second research question. Figure 4 shows the adaptation behavior of each scaling system in respect to the demanded and supplied number of application instances.

The CPU-metric achieves a substantially high ranking by exhibiting a stable and accurate behavior. Message queue metrics are inferior by having a high number of adaptations which increases the non-ideal timeshare and inaccuracy. All metrics except queue growth tend to have an empty queue, indicating that they are suitable to cope with the challenges of a message broker system.

I/O-intensive Microservice. In the next setup we investigate the achieved elasticity for each performance metric to scale an I/O-intensive microservice. We define as external service time $ST_{ext} = 400ms$ and as transferred payload $N_{data} = 15kB$. Table 2 shows the ranking and Figure 5 the adaptation behavior of each metric.

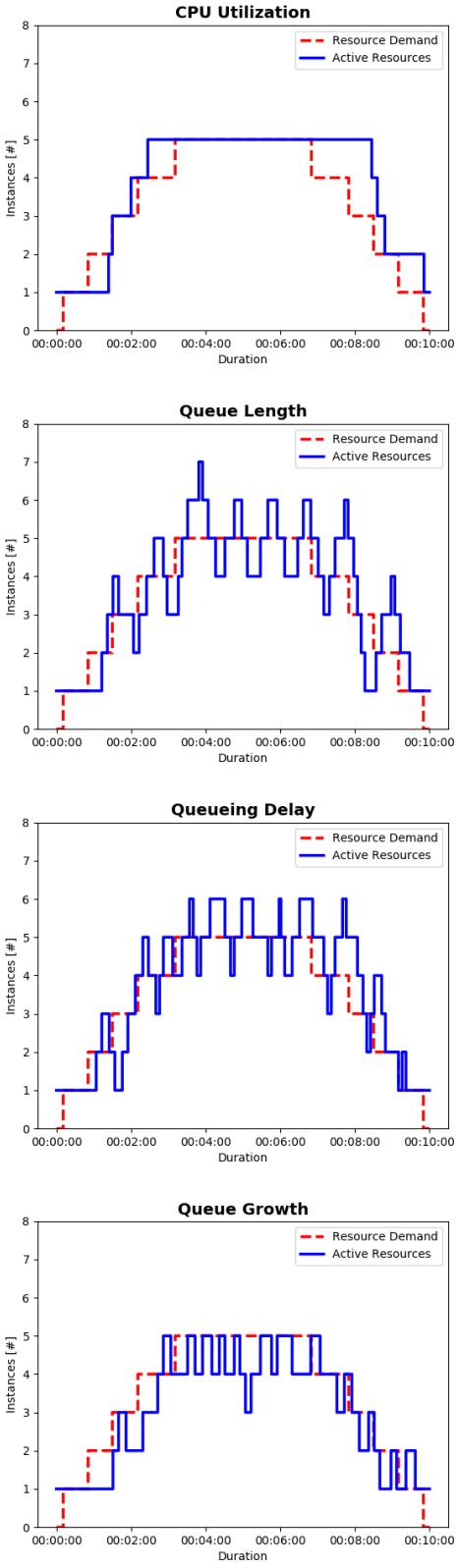

Figure 4: Adaptation behavior of a threshold-based rules auto-scaler for each performance metric in a compute-intensive microservice setup.

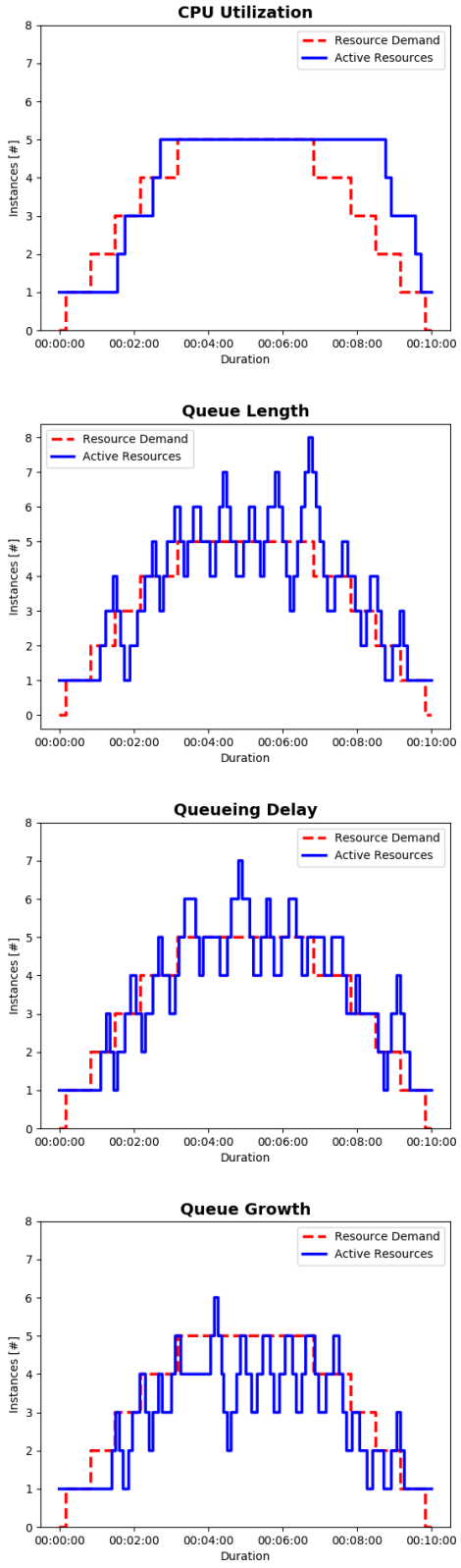

Figure 5: Adaptation behavior of a threshold-based rules auto-scaler for each performance metric in an I/O-intensive microservice setup.

Metric	Threshold	Avg. Q.-Length	Score
Queue Delay	0.75 / 1.25 [sec]	2.89 [msg]	270.78
CPU	0.4 / 0.9 [%]	3.99 [msg]	253.98
Queue Length	0 / 0 [msgs]	4.32 [msg]	228.13
Queue Growth	0 / 1 [msgs/sec]	163.87 [msg]	219.24
Baseline	- / -	-	100

Table 2: Elasticity score for each performance metric adapting the I/O-intensive microservice

The CPU-metric is in this setup a suitable metric. Its threshold is in a narrow and small area induced by waiting for the external service. This renders this metric vulnerable to background processes on the same system. Queueing delay has achieved the highest score but has as the other message queue metrics a high number of adaptations.

Discussion. If the microservices have a constant behavior all metrics are suitable for scaling decisions and exhibit a high score. However, message queue metrics suffer from oscillation which can increase the costs depending on the pricing model. CPU is a suitable proxy to represent the current work on the system in both microservice classes. Queue growth is the only metric which has a high number of messages in the queue. The queue growth metric considers the relation of arrival and departure rate thus aiming to a steady state of the queue. If the queue length is not zero and the arrival rate is zero such a system tends to scale down since it has a negative growth resulting in a slow draining of the accumulated messages in the queue.

To answer the third and fourth research question we vary the service characteristics and observe the behavior of each configured auto-scaling system.

Compute-intensive microservice. To investigate the influence of varying computational time on the performance of a configured threshold-based rules auto-scaling system we vary the computational steps required to process a message. Figure 6 shows the elasticity score and Figure 7 the average queue length.

With decreasing computation time the throughput of the microservice increases. For this reason the resource demand can be so small that no scaling operation is required and the baseline exhibits an ideal provisioning. Therefore the score of each metric is approximating the baseline score. With increasing computation time the difference between baseline – with a potentially massive underprovisioning – and scaling operations based on the performance metrics is more prominent leading to a generally higher score.

The elasticity score is heavily influenced by the computation time but is in nearly all the cases above the baseline score. The length of the message queue is stable in all cases, coping with the challenge of a flooded queue.

I/O-intensive Microservice. We address research questions 3 and 4 by varying the service time of the external service in a range of $0ms - 1200ms$. The thresholds for each metric were optimized for an external service time of $400ms$. Figure 8 shows the ranking of each metric in dependency to the external service time and Figure 9 the average queue length.

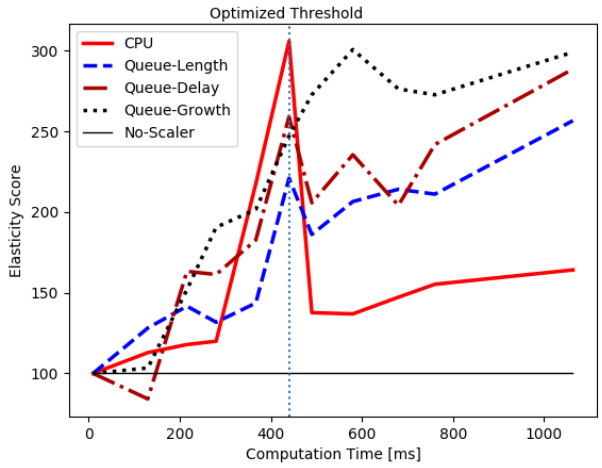

Figure 6: Elasticity for each performance metric with optimized thresholds for varying computational service time of a compute-intensive microservice.

Figure 8: Elasticity for each performance metric with optimized thresholds for a specific external service time setup of an I/O-intensive microservice.

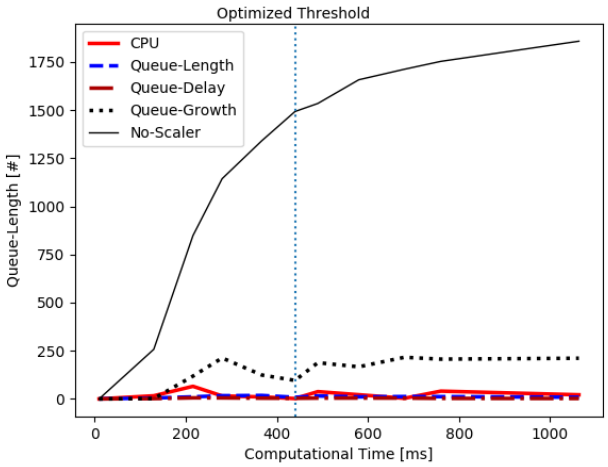

Figure 7: Average queue length for each performance metric with optimized thresholds for varying computational service time of a compute-intensive microservice.

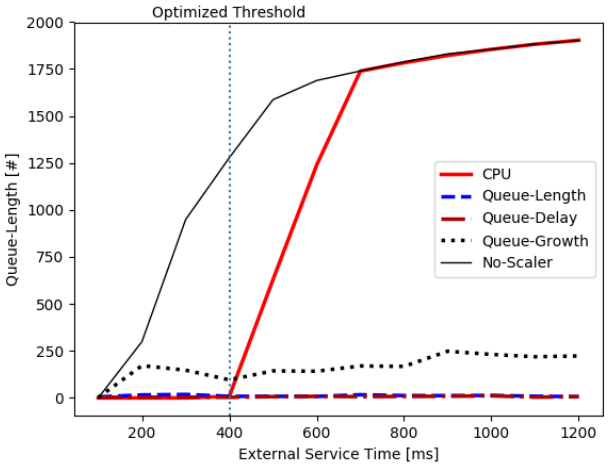

Figure 9: Average queue length for each performance metric with optimized thresholds for a specific external service time setup of an I/O-intensive microservice.

The CPU utilization is sensitive to changes of the external service time. The underlying issue is the strong influence of the external service time on the microservices CPU utilization. We observe that with an increasing external service time the CPU utilization decreases and with an decreasing external service time it increases. For this reason the auto-scaling system will scale out with a decreasing external service time even though the throughput of the microservice is increased. With an increasing external service time the auto-scaling system scales in even though the throughput of the microservice is degraded. Whereas the first case leads to high costs, the second case leads eventually to a flooded queue thus degrading both performance and reliability.

Discussion. This set of experiments has shown that the CPU is sensitive to changes in the external service characteristics and shows a significantly worse behavior than relying on message queue metrics. Thresholds for message queue metrics are much more resistant to changes in microservice characteristics. For this reason we underline the benefits on relying on message queue metrics for I/O-intensive microservices instead of the traditional CPU metric. However, if the microservices have a constant behavior

the CPU is more suitable due its capability to proxy the work on the system.

4.4 Discussion

In this section we investigate the influence of the external service time and the transferred payload on the microservices CPU utilization in order to understand the results of the previous section.

The I/O-intensive microservice of the running example communicates with the external service via the lightweight REST-protocol in a synchronous manner. The previous evaluation has already revealed an intense influence on the CPU utilization and throughput of a microservice in dependency to the characteristics of the external service. We assume that the response time for a single request is driven by the external service time ST_{ext}, the internal service time ST_{int} and the transfer time $T_{transfer}$. Thus leading to the following throughput model in a fully utilized service:

$$X = \frac{1}{ST_{ext} + ST_{int} + T_{transfer}}$$

The average CPU utilization U_{CPU} is the fraction of the CPU busy time T_{CPU} to the average response time $T_{response} = \frac{1}{X}$:

$$U_{CPU} = 100 * \frac{T_{CPU}}{T_{response}}$$

To simplify the relation between CPU and data transfer let λ be a factor of the transfer time $T_{transfer}$ to describe a linear influence on the CPU time such as $T_{CPU} = T_{BaseCPU} + \lambda * T_{transfer}$.

The transfer time $T_{transfer}$ is influenced by the transfer rate v such as $T_{transfer}(N_{data}) = \frac{N_{data}}{v}$ This leads to the final model for the throughput X and CPU utilization U_{CPU}:

$$X(ST_{ext}, N_{data}) = \frac{1}{ST_{ext} + ST_{int} + T_{transfer}(N_{data})}$$

$$U_{CPU}(ST_{ext}, N_{data}) = 100 * \frac{T_{BaseCPU} + \lambda * T_{transfer}(N_{data})}{T_{response}}$$

To parameterize this analytical model we perform two measurements with $ST_{ext} = 1ms$ and $N_{data} = [1Byte, 10MByte]$ for the sending and receiving scenario. In the first case we neglect the influence of data transfer to retrieve the base service time ST_{int} and base CPU time $T_{BaseCPU}$ per request. In the second case we calculate the transfer speed and can derive the factor λ for the CPU utilization. We solve this formula analytically to predict the CPU utilization and throughput in dependency to the external service time and the transferred payload. The measurements and prediction results for the receiving scenario are shown in Figure 10, 12 and for the sending scenario in Figure 11, 13. The dashed line shows the predicted value whereas the solid line shows the measured values.

The model has a relative error in predicting the CPU utilization of 22.0 % in the sending scenario and 21.6 % in the receiving. The relative error of the throughput is 5.2 % and 3.9 %, respectively. We assume that the relative high error of predicting the CPU utilization is caused by modeling a linear relationship between CPU utilization and transfer time since the prediction is much more precise for variations in service time.

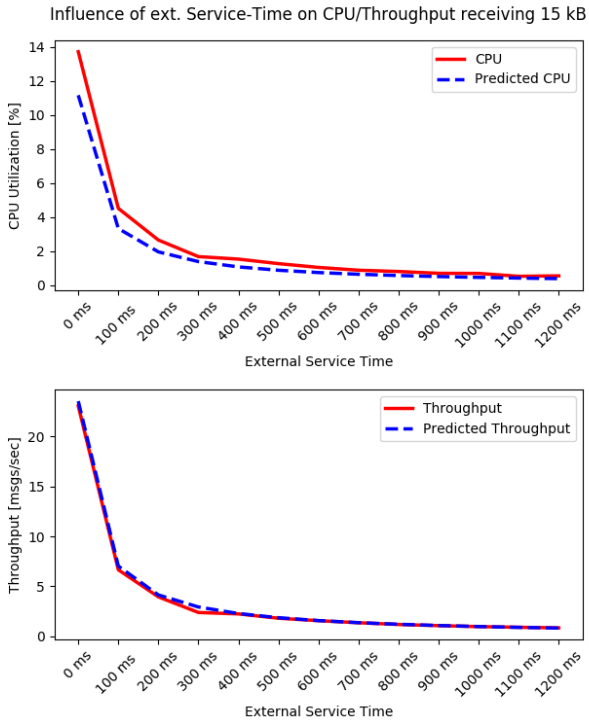

Figure 10: Influence of the external service time on the CPU utilization and throughput of the I/O-intensive microservice in a receiving scenario. With an increasing external service time the wait time increases leading to a decreased CPU utilization and throughput.

Ext. Service Time	Payload	CPU Utilization	Throughput
0 ms	1,25 MB	12.92 %	4.13
50 ms	15 kB	11.16 %	23.07

Table 3: Variations of throughput with a similar CPU utilization.

The influence of the external service time ST_{ext} is intuitive as it directly affects the throughput. However, by increasing the payload N_{data} the CPU utilization grows in a receiving and shrinks in a sending scenario. This is counter-intuitive and requires further investigation.

Some constellation of the external service characteristics can lead to fundamentally different throughput with a similar CPU utilization. Especially when using static thresholds the CPU threatens to be a false indicator of the actual performance. We measure a difference up to 600 % with a roughly equal CPU utilization in some configurations for the external service, as shown in Table 3.

Another factor is the transfer rate v, which is strongly influenced by the network bandwidth. Since many cloud environments like Amazon AWS or Microsoft Azure exhibit a varying network throughput on the same configuration [18][19] variations affect the transfer time and influence CPU and throughput.

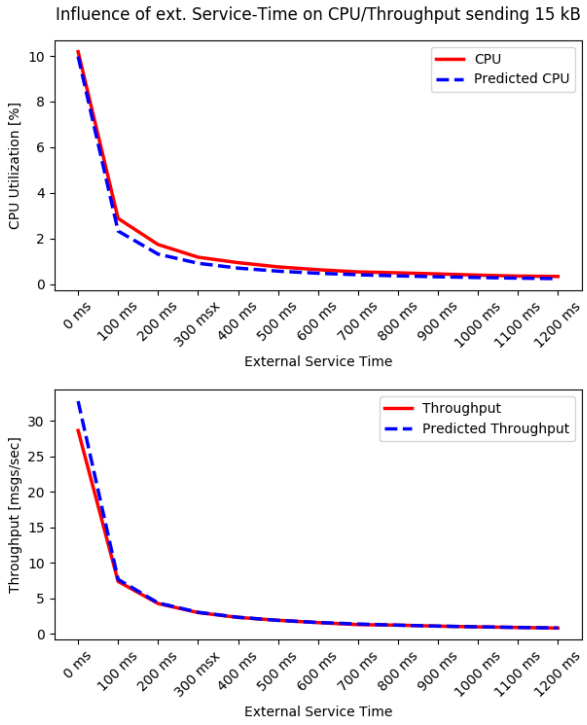

Figure 11: Influence of the external service time on the CPU utilization and throughput of the I/O-intensive microservice in a sending scenario. With an increasing external service time the wait time increases leading to a decreased CPU utilization and throughput.

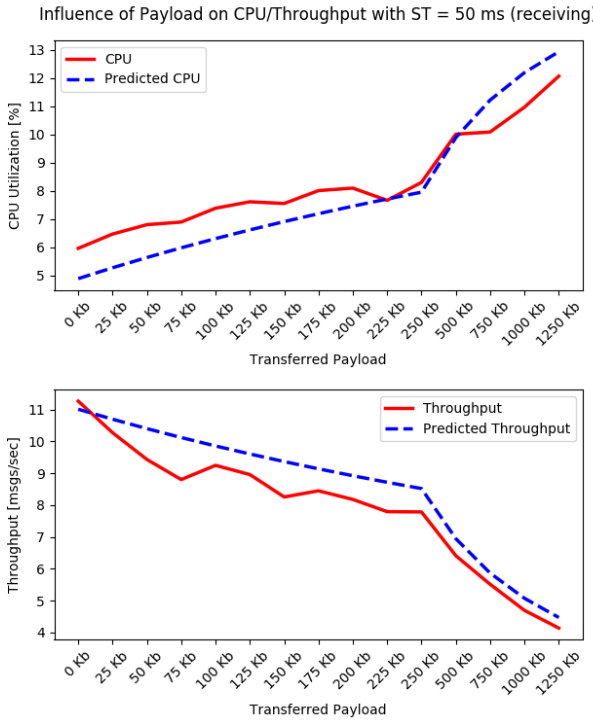

Figure 12: Influence of the transferred payload on the CPU utilization and throughput of the I/O-intensive microservice in a receiving scenario. With increasing payload the CPU utilization increases whereas the throughput decreases.

4.5 Threats to validity

In this section we consider the four classes of validity of a case study [22]:

4.5.1 Internal validity. In the running example the I/O-intensive microservice utilizes a single thread to process messages and communicate with the external service. By relying on a single blocking thread, the characteristics of the external service have a stronger influence on the microservice thus limiting the influence of other factors.

4.5.2 External validity. The running example is a simplified model of a real-world application for the SaaS use case. Nevertheless the fundamental architecture is already discussed in academia and derived of the work of [5]. For this reason we assume that it is sufficient to represent this class of cloud applications.

Message queues offers built-in benefits for the communication of microservices by supporting a loose coupling and reliability. However, often microservice communication is based on the typically synchronous REST paradigm. Relying on such a communication paradigm would abolish the challenge regarding the message queue state of a message broker system. However, the influence of the external service characteristics on a microservice in terms of CPU utilization and throughput are still present in this case so it is possible to generalize the results for microservices which do not utilize a message queue.

4.5.3 Construct validity. Threshold-based rules auto-scaling is one of the simplest mechanisms to provision resources in a cloud environment. In this setup we identified that the quality of performance metrics for scaling decisions depends on microservice characteristics. We cannot exclude the possibility that other scaling mechanisms are more adaptive to changes in the microservice characteristics and their influence on the performance metrics. However, since we perceive threshold-based rules as one of the most common scaling strategies in industry we see validity of our investigation in real-world scenarios. By creating an analytical model of the influence of the external service characteristics on the microservices CPU utilization and throughput we emphasize the validity of the case study in an analytical manner.

In the case study we investigate the behavior based on a homogeneous workload. In practice the workload is expected to be heterogeneous, varying in size and computational requirements. However, since microservices usually deliver one functionality it mitigates the heterogeneity of workload compared to a monolithic application.

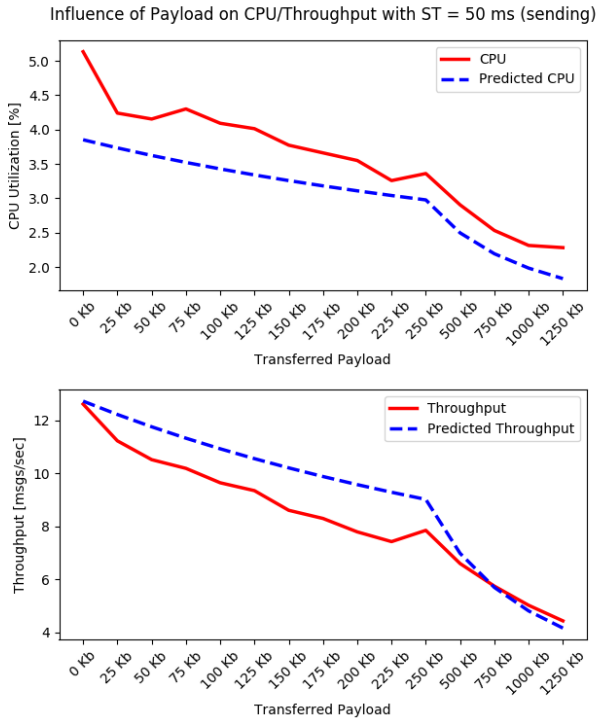

Figure 13: Influence of the transferred payload on the CPU utilization and throughput of the I/O-intensive microservice in a sending scenario. With an increasing payload the CPU utilization and the throughput decreases.

4.5.4 Reliability. By describing the concrete environment, the workload characteristics and the functionality of the microservices and the threshold-based rules auto-scaling system we strongly assume that the results are reproducible thus making it possible for another researcher to conduct the same study and obtain the same or very similar results. We see a limitation in the dynamic nature of the cloud environment which could possible change the optimal thresholds based on the time of experiment.

5 RELATED WORK

Runtime management is a well researched area in cloud computing with resource provisioning in an automatic manner as one of the main strategies. Extensive surveys can be found in [15] and [6]. An usual technique are threshold-based rules auto-scalers but there are approaches based on control theory, queueing theory, reinforcement learning or time-series analysis. More sophisticated approaches like [3] rely on workload forecasting, online resource demand estimation and performance models to improve the adaptation behavior. These techniques deeply rely on metrics which are able to represent the condition of the cloud system.

The performance of microservices is discussed in [10] and [7]. Whereas the first work addresses performance engineering for microservices the second work discusses challenges for the performance of microservices in general and mentions reliability challenges through the message-passing mechanisms. Both works do not discuss challenges associated with using performance metrics for scaling message consuming microservices in this environment.

With the uprising microservice architectural style message broker systems gain more importance. A short survey can be found in [13]. An approach to scale message queues is proposed in [24]. Whereas it offers the possibility to mitigate the challenge of a flooded or congested queue by replicating queues it does not address the underlying challenge of a long-pending imbalance in consumption and production rate. However, it is still a suitable strategy to improve the performance by reconfiguring the middleware if the number of consumers and producers exceeds the capacity of the message broker system. In [8] an approach is presented to load balance message queues to ensure that provisioned resources are used at max capacity. Furthermore it scales message queues and consumers in case of an overload. However, this is a sophisticated approach which does not address scaling consuming microservices in a generic threshold-based auto-scaling setup.

Since message queues resemble structurally a queueing system performance modeling of this area could be transferable. Performance modeling of queueing systems is described in-depth in [16]. Furthermore simulation of queueing petri nets like [14] could be suitable to performance analyze cloud applications with message queues in order to have a fine-granular view of the message queue state at design- and run-time.

The disadvantages of relying on the CPU utilization are addressed in [21]. In this work they measured a 150 % variation in response time of an E-Commerce benchmark with an equal CPU utilization of 80 %. In [17] the disadvantages of relying on a single performance metric – like CPU utilization – is mitigated by merging heterogeneous metrics into a single representation.

To the best of our knowledge we cannot identify related work which investigates the suitability of performance metrics for scaling different microservice classes and evaluates the resilience of thresholds to changes in the microservices characteristics.

6 CONCLUSION

In this paper we investigated the suitability of a set of performance metrics in a threshold-based rules auto-scaling system for scaling a SaaS cloud application in terms of elasticity and coping with the message queue state. We have shown, that the CPU utilization is a suitable metric for scaling all classes of microservices if they exhibit constant characteristics. However, it is a vulnerable metric for changes in the microservice characteristic. Especially if the CPU utilization of an I/O-intensive microservice is used as performance metric, it can result in no scaling decisions at all, threating the application in performance and reliability. We modeled the influence of the external service time and the transferred payload on the CPU utilization and identified further factors which renders the CPU unreliable. We have shown that thresholds based on message queue metrics are much more resilient to changes in the microservice characteristics. For this reason we underline the benefits on relying on message queue metrics instead of the microservices CPU utilization in similar setups.

REFERENCES

[1] [n. d.]. RabbitMQ Queue Description. https://www.rabbitmq.com/queues.html. ([n. d.]). Accessed: 2017-09-14.

[2] Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A Software Architect's Perspective.* Addison-Wesley Professional.

[3] André Bauer, Nikolas Herbst, and Samuel Kounev. 2017. Design and Evaluation of a Proactive, Application-Aware Auto-Scaler: Tutorial Paper. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 425–428.

[4] Alessio Botta, Walter de Donato, Valerio Persico, and Antonio Pescapé. 2016. Integration of Cloud computing and Internet of Things: A survey. *Future Generation Computer Systems* 56 (March 2016), 684–700. https://doi.org/10.1016/j.future.2015.09.021

[5] Cyril Cecchinel, Matthieu Jimenez, Sebastien Mosser, and Michel Riveill. 2014. An Architecture to Support the Collection of Big Data in the Internet of Things. IEEE, 442–449. https://doi.org/10.1109/SERVICES.2014.83

[6] You Chen, Yang Li, Xue-Qi Cheng, and Li Guo. 2006. Survey and taxonomy of feature selection algorithms in intrusion detection system. In *Information security and cryptology*. Springer, 153–167.

[7] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2016. Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036* (2016).

[8] Ahmed El Rheddane, Noël De Palma, Alain Tchana, and Daniel Hagimont. 2014. Elastic message queues. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE, 17–23.

[9] Martin Fowler and James Lewis. 2014. Microservices. *ThoughtWorks. http://martinfowler.com/articles/microservices.html [last accessed on February 17, 2015]* (2014).

[10] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. 2017. Performance Engineering for Microservices: Research Challenges and Directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ACM, 223–226.

[11] Nikolas Herbst, Rouven Krebs, Giorgos Oikonomou, George Kousiouris, Athanasia Evangelinou, Alexandru Iosup, and Samuel Kounev. 2016. Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics. *arXiv preprint arXiv:1604.03470* (2016).

[12] Nikolas Roman Herbst, Samuel Kounev, and Ralf H Reussner. 2013. Elasticity in Cloud Computing: What It Is, and What It Is Not.. In *ICAC*, Vol. 13. 23–27.

[13] Vineet John and Xia Liu. 2017. A Survey of Distributed Message Broker Queues. *arXiv preprint arXiv:1704.00411* (2017).

[14] Samuel Kounev. 2006. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *IEEE Transactions on Software Engineering* 32, 7 (2006), 486–502.

[15] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. 2014. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing* 12, 4 (Dec. 2014), 559–592. https://doi.org/10.1007/s10723-014-9314-7

[16] Randolph Nelson. 2013. *Probability, stochastic processes, and queueing theory: the mathematics of computer performance modeling.* Springer Science & Business Media.

[17] Valerio Persico, Domenico Grimaldi, Antonio Pescape, Alessandro Salvi, and Stefania Santini. 2017. A Fuzzy Approach Based on Heterogeneous Metrics for Scaling Out Public Clouds. *IEEE Transactions on Parallel and Distributed Systems* 28, 8 (2017), 2117–2130.

[18] Valerio Persico, Pietro Marchetta, Alessio Botta, and Antonio Pescapé. 2015. Measuring network throughput in the cloud: the case of amazon ec2. *Computer Networks* 93 (2015), 408–422.

[19] Valerio Persico, Pietro Marchetta, Alessio Botta, and Antonio Pescapé. 2015. On network throughput variability in microsoft azure cloud. In *Global Communications Conference (GLOBECOM), 2015 IEEE*. IEEE, 1–6.

[20] B. B. P. Rao, P. Saluia, N. Sharma, A. Mittal, and S. V. Sharma. 2012. Cloud computing for Internet of Things and sensing based applications. IEEE, 374–380. https://doi.org/10.1109/ICSensT.2012.6461705

[21] Jia Rao, Yudi Wei, Jiayu Gong, and Cheng-Zhong Xu. 2013. QoS guarantees and service differentiation for dynamic cloud applications. *IEEE Transactions on Network and Service Management* 10, 1 (2013), 43–55.

[22] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. 2012. *Case study research in software engineering: Guidelines and examples.* John Wiley & Sons.

[23] Rainer Storn and Kenneth Price. 1997. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization* 11, 4 (1997), 341–359.

[24] Nam-Luc Tran, Sabri Skhiri, Esteban Zim, et al. 2011. Eqs: An elastic and scalable message queue for the cloud. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*. IEEE, 391–398.

[25] Mario Villamizar, Oscar Garces, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, and Mery Lang. 2016. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. IEEE, 179–182. https://doi.org/10.1109/CCGrid.2016.37