Journal of Grid Computing (2021) 19: 20
https://doi.org/10.1007/s10723-021-09561-3

Cloud Resource Demand Prediction using Machine Learning in the Context of QoS Parameters

Piotr Nawrocki · Patryk Osypanka

Received: 24 November 2020 / Accepted: 20 April 2021 / Published online: 8 May 2021

Abstract Predicting demand for computing resources in any system is a vital task since it allows the optimized management of resources. To some degree, cloud computing reduces the urgency of accurate prediction as resources can be scaled on demand, which may, however, result in excessive costs. Numerous methods of optimizing cloud computing resources have been proposed, but such optimization commonly degrades system responsiveness which results in quality of service deterioration. This paper presents a novel approach, using anomaly detection and machine learning to achieve cost-optimized and QoS-constrained cloud resource configuration. The utilization of these techniques enables our solution to adapt to different system characteristics and different QoS constraints. Our solution was evaluated using a system located in Microsoft's Azure cloud environment, and its efficiency in other providers' computing clouds was estimated as well. Experiment results demonstrate a cost reduction ranging from 51% to 85% (for PaaS/IaaS) over the tested period.

P. Nawrocki (✉) · P. Osypanka
Institute of Computer Science, AGH University of Science and Technology, al. A. Mickiewicza 30, 30-059 Krakow, Poland
e-mail: piotr.nawrocki@agh.edu.pl

P. Osypanka
ASEC S.A., ul. Wadowicka 6, 30-415 Krakow, Poland
e-mail: patryk.osypanka@agh.edu.pl

Keywords Cloud computing · Resource usage prediction · Anomaly detection · Machine learning · Quality of service · Resource cost optimization

1 Introduction

Cloud computing has seen widespread use among enterprises of all sizes. Its benefits are undisputed; however, the cost of cloud resources can be considerable, especially when resources are provisioned with a large safety margin to avert system unavailability during unexpected emergencies. Unrestricted scaling can entail severe costs: according to *Flexera 2020 State of the Cloud Report*[1] companies spend on average 23% of their budgets on cloud resources and expect this percentage to increase further. All major computing cloud providers, such as Amazon (Amazon Web Services), Microsoft (Azure) or Google (Google Cloud Platform), offer access to auto-scaling components built into cloud platforms. There are also cloud provider-independent, commercial applications[2] designed to optimize cloud resource utilization; likewise, various studies, which we present in Section 2, dedicated to

[1]Flexera 2020 State of the Cloud Report – https://info.flexera.com/SLO-CM-REPORT-State-of-the-Cloud-2020

[2]Azure Cost Management – https://azure.microsoft.com/en-us/services/cost-management

this topic can be found in the literature [13, 26, 31]. Resource usage optimization efficiency (i.e. the percentage of saved resources) depends on the optimization technique. The standard approach to resource optimization [1, 8, 34, 38, 39] focuses on system model formalization, which attempts to categorize system load. This can result (as presented in Section 2) in poor efficiency when confronted with real-life data with random anomalies. On the other hand, machine learning (ML) approaches [24, 41], while properly handling the randomness of data, are less controllable in terms of optimization degree. Aggressive optimization, although cost-efficient, causes quality of service (QoS) degradation. In some systems, a QoS decline cannot be accepted, in others it is acceptable, but even then, it has to be managed and cannot exceed reasonable limits.

Our idea is to enable adaptation to irregular data while introducing control measures that can mitigate the impact of optimization on QoS. The solution proposed by us, the Long-Term Prediction System (LTPS), automatically determines the relationship between optimized system load and its QoS parameters by performing a series of trials. Machine learning and anomaly detection techniques are used to build knowledge, which is subsequently utilized when predicting resource provisioning.

The major contributions of this paper can be briefly outlined as follows:

- designing an autonomous solution, which, as opposed to many current solutions presented in Section 2, does not require a formal definition of load characteristics or a specification of the system being optimized; the solution automatically collects data that define the relationships between the optimized system load and QoS, supported by an anomaly detection filter;
- developing a novel approach to utilizing the relationship data collected during resource prediction while observing the QoS constraint set (with the help of ML models), which yields significant cost reductions without QoS deterioration;
- demonstrating that the proposed solution is an industrial-grade one ready to be deployed in an unmoderated domain by implementing it in a cloud provider's environment, conducting tests using real-life historical data from the production system, presenting experiment outcomes and verifying the efficiency of the proposed solution along with actual cost reductions for three major cloud providers;
- comparing the efficiency of the LTPS to state-of-the-art solutions and achieving improvements in optimization results in most cases, as shown in Section 2 and Section 4.3.

The rest of this paper is structured as follows: Section 2 contains a description of related work, Section 3 is concerned with defining in detail the QoS-driven cloud resource prediction approach, Section 4 describes the evaluation of the prediction solution, and Section 5 contains the conclusion and further work.

2 Related Work

Several studies have been devoted to resource allocation optimization in the context of quality of service. As such optimization processes usually consist of several stages and are applied to different types of optimized systems, there is a great diversity of approaches and methods used. In the following section, the most common optimization approaches, their properties and comparison with the LTPS are presented. QoS parameters fluctuate between the solutions proposed and evaluation methods vary, making a comparison of different solutions very problematic. Notwithstanding, many works include comparisons with state-of-the-art solutions, so – bearing in mind the shortcomings of this approach – we used the results obtained in this respect to compare the different works, since they were the only common denominator we were able to find.

One of the popular optimization methods is optimization based on task characteristics. Akintoye et al. [2] presented a genetic algorithm for the task allocation problem; however, the authors only used system cost as a QoS factor. A somewhat different approach is described in [1], as this work is focused on the optimization of scientific workflows. Multi-objective QoS optimization accounting for factors such as cost and response time was described and evaluated for the most popular workflows with known load characteristics, despite less clean test data used for evaluation, the optimization efficiency of the LTPS is higher than that presented in [1]. Similarly, the authors of [17] compared 50 papers devoted to optimization. The review covers papers with multiple QoS metrics, e.g. Chen et al. [10] proposed a Pareto optimal approach and tested it with WSC'09 QoS parameters; load data, however, were generated artificially. Ye et al. [40] described multivariate time series analysis and tested the idea using real-life QoS data from [20] supplemented by randomly generated cost values; load data were nonetheless artificially generated. Comi et al. [12] presented a reputation-based model, and tests using the WS-DREAM dataset were planned but were finally not conducted. The LTPS was able to provide better optimization results than the solutions depicted in [10, 40]. The authors of [35] proposed an artificial bee colony algorithm for scheduling in a cloud environment, a cloud resource optimization algorithm based on particle swarm optimization was presented in [15], a modified bird swarm algorithm for edge cloud optimization was depicted in [19] and an eagle strategy cloud service composition was presented in [14]. The LTPS obtained optimization results many times better than those presented in [15, 19] and similar or better than in [14, 35]. The aforementioned works focus on the optimization of various QoS parameters with predefined weights assigned to these parameters and were mainly tested using artificial data in different simulation environments. On the other hand, the LTPS allows us to set QoS parameters, the optimization process is aware of the constraints required and adjusts cloud resources accordingly, and we tested it using real-life historical noisy data with anomalies in terms of QoS and load, and it also uses real-life services in a real-life cloud environment, showing that it can be deployed in an actual system.

Reactive optimization, unlike optimization mentioned beforehand, does not require the assessment of resource demand related to incoming requests. In [22], the authors proposed performance models to optimize cloud resources with QoS constraints, and Beheda et al. [6] presented QoS and performance optimization using a dynamic provisioning technique. The systems described use optimized system feedback to ensure the QoS level requested, the authors did not provide a comparison to the existing solutions. A number of surveys also discuss QoS-based reactive resource provisioning, a technical survey on [36] consists of works on the subject of resource availability. The authors of two of them [7, 16] use optimized system feedback; the authors of [7] compared its optimization to static provisioning; the LTPS, when evaluated in the same way, generates much better results. Reactive solutions tend to poorly handle rapid changes in incoming requests, while the LTPS predicts demand using machine learning and scale resources in advance, which allows us to mitigate the delays caused by the scaling process.

A distinct approach to QoS as robust resource provisioning (robust optimization) was presented in [34]; the authors defined various types of uncertainty in cloud resource provisioning and presented a scheduling policy; however, no tests of this approach were performed. Chaisiri et al. [8] presented a robust cloud resource provisioning optimization algorithm based on stochastic programming and robust optimization. A robust multi-resource allocation approach is proposed in [39]; the authors defined the cost function as a resource allocation metric and modeled resource demand uncertainties. Both aforementioned solutions were evaluated using artificial test data. Robust optimization was also applied in [18], and the model described was used to solve an interesting problem (albeit different than our research interests): one was of the optimization of cloud resources required in a disaster recovery scenario; the model was evaluated using a numerical simulation. On the other hand, the robustness of our work stems from proactive resource allocation based on anomaly detection and ML-driven prediction. The LTPS was tested with both Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) cloud providers' service models, using an actual cloud environment and real-life historical data. Only the authors of [8] provided an efficiency comparison with other works, resulting in optimization that was an order of magnitude worse than the LTPS.

Resource demand prediction, which is a different approach to the optimization problem, aims to overcome the drawback of the aforementioned methods, combining the versatility of reactive models with prediction capabilities. A solution involving fuzzy logic was used by T. Senthil Kumar [21] (achieving results worse than the LTPS), and auto-scaling of network resources based on machine learning, tested against real-life data from [27], was proposed by Rahman et al. [29]. The results shown in [29] vary significantly depending on the optimization parameter; in best cases, an optimization level is achieved which is comparable to our results. Sniezynski et al. [31] describe a virtual machine reservation plan generator based on computing level history. The authors evaluated different ML algorithms and performed tests reflecting the system's performance over a year; however, no comparison to similar solutions was made. A QoS-oriented cloud resource optimization was proposed by Sun et al. [32]. The evaluation of the C2RAM solution proposed was performed in the Amazon and Google cloud environments using artificial load data prepared in advance. The simultaneous use of different cloud providers was also presented in [4]. The authors described a hierarchical receding horizon algorithm that accounts for computing cloud usage only in the IaaS model and uses workload data from a real-life web site. The LTPS performed much better than [32] and similar to [4] when optimization efficiency was compared. A different view of QoS and cost optimization is presented by authors of [26], where an agent-based method is used to explore the existing services' QoS capabilities. Next, a particle swarm optimization (PSO) algorithm is used to select the best combination of resources; nevertheless, the LTPS demonstrated better optimization efficiency than the results presented in [26]. Another approach is presented in [13], where the authors propose a benchmark process that tests different virtual machine types from different cloud providers, but there is no comparison to state-of-the-art works. In comparison, the LTPS explores the QoS topic more thoroughly, allowing the direct configuration of QoS parameters as opposed to the generic coefficient in the works mentioned above. Our proposition accounts for the PaaS model while the solutions described only take into consideration the provisioning of virtual machines. Additionally, the LTPS was tested using a real-life system and noisy data, while the works mentioned above use system simulators. In [28], we described our initial research on resource usage cost optimization; we used both anomaly detection and machine learning to predict resource usage and reduce cloud resource cost. We measured optimization impact on the system's QoS; however, we did not consider QoS as an optimization parameter. The LTPS extends and complements previous research, adding the ability to define QoS parameters. Adding initial analysis of relationships between load and QoS parameters (e.g. response time or availability) allows a more comprehensive approach to the cloud resource optimization process (e.g. meeting SLAs, ensuring good user experience). We have also extended ML and anomaly detection usage. We use them, as previously, for resource usage prediction, but additionally we use machine learning and anomaly detection to achieve desired QoS parameters.

An analysis of existing solutions reveals that currently none of them solve the problem of optimizing different cloud service models (IaaS, PaaS) with anomaly detection, incorporating cloud-provider specific mechanisms, automatic resource allocation, proactive usage prediction and accounting for QoS constraints. As mentioned beforehand, it is difficult to compare results, as most referenced works were designed for slightly different circumstances and used diverse evaluation techniques, yet the approximate comparison presented demonstrates that the LTPS performs similarly to, or better than, the works presented (as depicted in Section 4.3) while providing additional advantages of broader cloud providers' service models and QoS constraint configuration. The contribution of this study is to define such an autonomous and complete solution along with a demonstration of its industrial application possibilities by evaluating its performance in a real-life environment with historical usage data from a production server. The LTPS works with different cloud service models (IaaS, PaaS) and accounts for various resource types (RAM, CPU, etc.). The optimization mechanism is resilient to temporary usage spikes and adapts to the pricing policy obtained directly from the cloud provider.

3 Cloud Resource Demand Prediction

Resources allocated to systems located in the cloud are often overprovisioned to avoid quality of service degradation in case of unexpected emergencies. This leads to both unnecessary spending and electricity overuse. Cloud-located systems involve various resource types, e.g. computing power, RAM, storage or I/O operations. Different optimization techniques reduce resource usage, unfortunately causing QoS (e.g. response time or availability) degradation at the same time. Thus, it is difficult to ascertain cloud resource levels which will be optimal simultaneously from the cost and QoS perspectives.

Our research interests cover systems that involve applications deployed to the most popular IaaS and PaaS cloud service models,[3] which are able to scale up

[3]Flexera 2020 State of the Cloud Report – https://info.flexera.com/SLO-CM-REPORT-State-of-the-Cloud-2020

(cloud component quality) and scale out (cloud component quantity); these two approaches do not interfere with each other. Additionally, they are deployed to separate cloud components and thus can be optimized independently. For simplification purposes, we present our idea using a single application; nevertheless, as shown in Section 4, the solution is fully applicable to systems composed of multiple applications deployed to both PaaS and IaaS models. To optimize cloud costs and simultaneously maintain QoS parameters, we have designed and developed a solution which:

– determines the relationship between available resources and the QoS of the system being optimized (*Working system*);
– predicts future *Working system* load levels;
– predicts future resource demand on the basis of determined relationships and future system load;
– creates a resource provisioning plan, which optimizes the resource cost while maintaining the required QoS parameters on the basis of predicted resource levels.

Our optimization solution consists of different modules (Fig. 1), which cooperate in two stages – Discovery (the preparation of the QoS model) and Execution (the optimization process) which are presented in Section 3.1 and Section 3.2, respectively. Additionally, Section 3.1 illustrates the *Copy of the system* validation process.

3.1 Discovery Stage

The *Working system*, the *Copy of the system*, the *QoS module* and the *Load emulator* are involved in the *Discovery stage*. The QoS of the system is conditioned by two variables: the resources available and the amount of system workload. The aim of this stage is to determine the relationship between the workload, the resources available and the QoS. For clarity, we use computing power (measured in cores) as the resource, request level (in requests per second) as system load and response time (in seconds) as the QoS; nevertheless, the LTPS is able to take into account load, resources and QoS expressed as multiple parameters. To determine the relationship mentioned, sufficient examples of cloud resources, load levels and response times are needed. In cases where the system is rarely scaled or not scaled at all, usage history data will suffer from insufficient resource level diversity. To overcome this drawback, the LTPS generates additional relationship examples as it performs a series of load emulations on the *Copy of the system*. To achieve this goal, the *Load emulator* gathers from the *Working system* historical load characteristics (i.e. minimum and maximum observed request per second values) and historical resource levels (the maximum observed core count, which is treated as the starting point for the optimization) to determine the emulation range required. Load emulations cannot be performed on the *Working system* directly as emulation results would be affected by standard system usage and, on the other hand, load emulations could disrupt *Working system* stability. Therefore, the *Load emulator* uses the *Copy of the system*, which is a copy of the *Working system*.

The *Load emulator*, using obtained historical load characteristics and historical resource levels, prepares a set of data to perform emulations. Regarding resources, the solution uses every resource level available for a given cloud provider $\mathbf{R} = [R_1, R_2, \ldots, R_i]$ where R_i is the resource with the aforementioned maximum value. In regard to load level, the range between minimum and maximum observed values is divided into equal-sized intervals $\mathbf{L} = [L_1, L_2, \ldots, L_j]$ where L_1 and L_j are minimum and maximum observed load values, respectively. For comparison purposes, we performed tests for both 10 and 20 levels, since their number is a trade-off between accuracy (more levels will provide more data and therefore should produce a more detailed relationship) and the time required to complete the emulation for all selected levels. During load emulation, the *Load emulator* scales *System copy* resources according to $\mathbf{R}$ and selects load levels according to $\mathbf{L}$ (i times j loads in total). Each observed result (response time) is stored $\mathbf{T} = [T_{1,1}, \ldots, T_{1,j}, \ldots, T_{i,1}, \ldots, T_{i,j}]$ and used to define a relationship:

$$(R_a, L_b) \mapsto T_{a,b} \tag{1}$$

for $a \in (1, \ldots, i)$ and $b \in (1, \ldots, j)$. Although both the *Copy of the system* and the *QoS module* are located in the same computing cloud, the load emulation process is vulnerable to distortions caused by the cloud provider's underlying hardware infrastructure. To overcome this drawback, the solution filters out anomalies using the exchangeability martingales function [11] which exhibits very good efficiency, while, working in an unsupervised mode, does not require

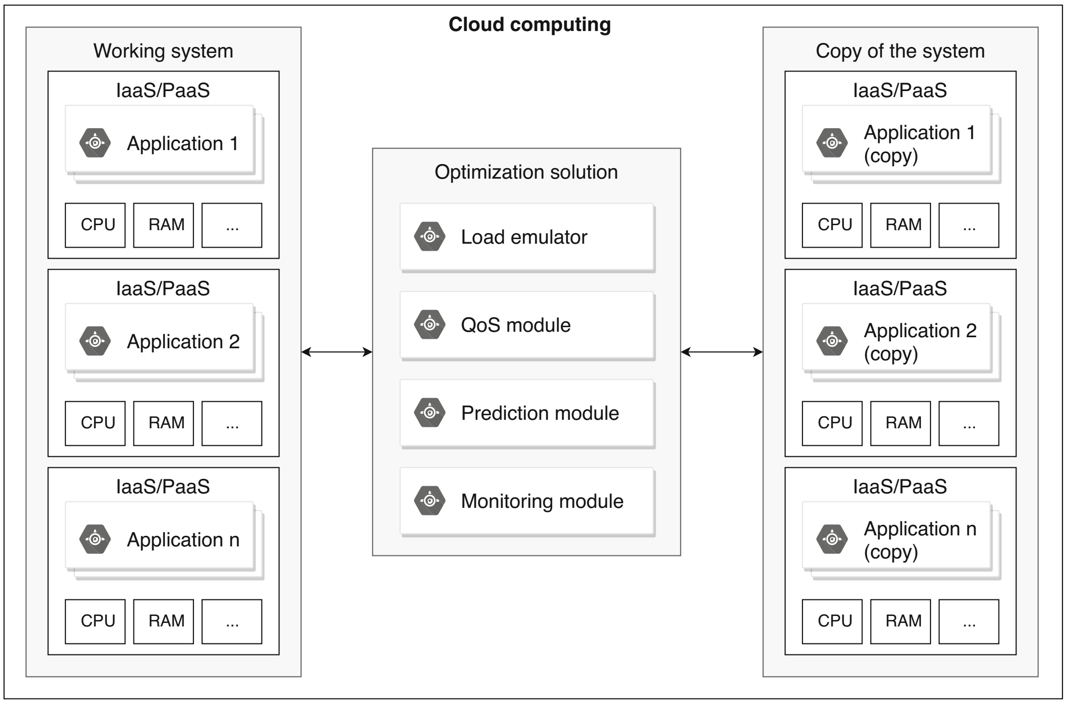


Fig. 1 Architecture of the LTPS

training data. As ground-truth examples are not available in this case, it perfectly suits our needs. The data gathered are subsequently used as training data for the ML regression model. It should be noted that the ML model determines a relationship slightly different than in (1):

$$(L_b, T_{a,b}) \mapsto R_a \tag{2}$$

Therefore, **L** and **T** are treated as features while **R** is treated as a label. This makes it possible to adjust (at a later stage) cloud resources according to predicted load and required response time. For comparison purposes, we selected four different ML regression models: Bayesian Linear Regression (BL) [33], Neural Network Regression (NN) [25], Poisson Regression (P) [9] and Decision Forest Regression (DF) [30]. Arranging parameters (load, resource or QoS) into a vector (e.g. the number of cores, working memory size and network speed as resources) creates a matrix of possibilities and greatly increases the emulation scope; however, the process itself remains unchanged. The discovery process described can be depicted in the form of a diagram (Fig. 2); the arrow directions represent data flow.

Although the *Copy of the system* is a copy of the *Working system*, it might not reflect all working conditions of the *Working system*, e.g. third-party services may not be used and may have to be emulated. Therefore, the accuracy of the *Copy of the system* has to be validated. The validation process uses historical load data (e.g. the number of requests) and historical resource level and its utilization (e.g. the number of cores and their usage) from the *Working system*. The *Copy of the system* is loaded with historical data and its resource usage is compared to historical usage from the *Working system*. To measure validation results (for validation performed during hours $t = 1, 2, \ldots, m$) we defined the V metric, which is the mean absolute percentage accuracy defined as:

$$V = \frac{1}{m} \cdot \sum_{t=1}^{m} \left(1 - \left|\frac{W_t - C_t}{W_t}\right|\right) \tag{3}$$

where W_t is the *Working system's* parameter value and C_t is the *Copy of the system's* value of the same parameter for hour t. V is expressed as a percentage.

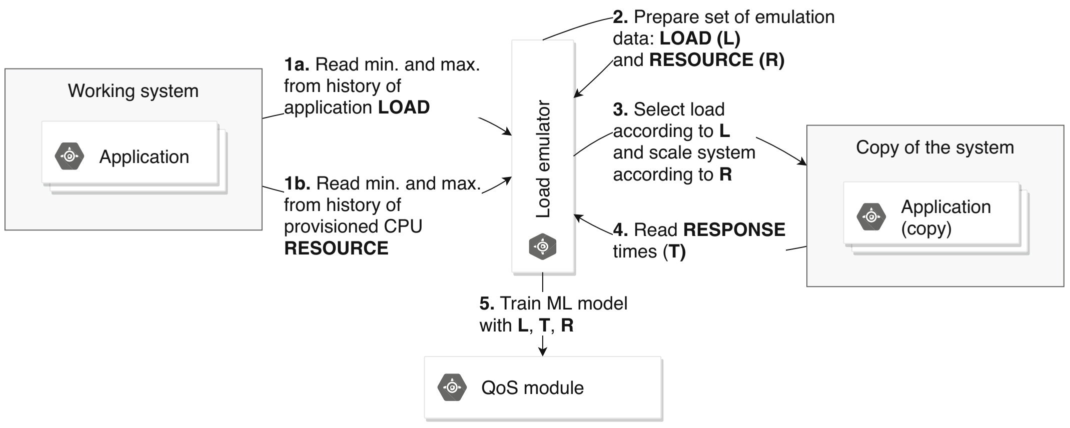

Working system
Application
1a. Read min. and max. from history of application LOAD
1b. Read min. and max. from history of provisioned CPU RESOURCE
Load emulator
2. Prepare set of emulation data: LOAD (L) and RESOURCE (R)
3. Select load according to L and scale system according to R
4. Read RESPONSE times (T)
Copy of the system
Application (copy)
5. Train ML model with L, T, R
QoS module

Fig. 2 Discovery stage – operation sequence and data flow

3.2 Execution Stage

The *Working system*, the *QoS module*, the *Prediction module* and the *Monitoring module* are involved in the *Execution stage*. The main objective of the optimization process (in the *Execution stage*) is to predict system load (requests per second) for subsequent days, calculate resource demand for those days (CPU cores required) and, based on this, calculate a cost-optimal resource configuration for the next days. All predictions and calculations are made for the next week with hourly resolution. A long prediction timeframe reduces prediction frequency and makes it possible for the administrator to inspect the system if required. As mentioned above, for clarity, we use the core count as a resource, requests per second as system load and response time as the QoS. We assumed the required response time to be constant during the prediction period; however, it can be adjusted between predictions (i.e. every seven days). Working in a weekly loop, the *Prediction module* collects from the *Working system* historical load data (the request level along with the time of day and the day of the week) from a past period. The length of this period affects prediction stability and adaptation rate, therefore it has to be adjusted to optimized system characteristics: it must be sufficiently long to observe usage patterns but also sufficiently short to allow prediction adaptation (e.g. four weeks). Subsequently, the data collected are filtered (for the same reason as in the previous stage) and used as training data for the ML prediction algorithm. Based on results from [28], we used Decision Forest Regression (DF); it should be noted that the ML model created by the *Prediction module*, using historical load data, predicts the future system load level for a given hour and day of the week, as opposed to the ML model created by the *QoS module*, which, using load emulation results, determines resource requirements for a given load level and time response required. The load levels predicted for the next week ($\mathbf{O} = [O_1, \ldots, O_m]$ where O_i is the predicted load level for hour i and m is the number of hourly predictions for a week) are combined with the required response time and used as input data for the ML model trained by the *QoS module* in the previous stage, which results in the prediction of required resource levels (core count) ($\mathbf{E} = [E_1, \ldots, E_m]$ where E_i is the prediction of the required resource level for hour i).

In the next step, the *Prediction module* obtains the current pricing plan from the cloud provider and calculates the optimal provisioning configuration. Depending on the cloud provider's policy, the same resource level can be provisioned with a different configuration and therefore at a different cost (i.e. multiple small machines versus a single larger one). As concerns the matrix of possibilities represented by the aforementioned vectors as parameters, the number of possible configurations is usually large, and therefore calculating all variants is not feasible. This problem was the subject of our preliminary research described in [28], a discrete particle swarm optimization (Integer-PSO) algorithm [5], modified by us to select a cost-optimal reservation configuration for a given value of demand, was chosen due to its efficiency and accuracy and, therefore, is used by the *Prediction module*. Given the predicted level of resources E_i for hour i, where $i \in (1, \ldots, m)$ and n different component configuration types (e.g. compute-optimized, memory-optimized, general-purpose) from the cloud provider $[X_1, \ldots, X_n]$, the optimization problem is to find a set of configurations $\mathbf{A}$ which will meet the E_i constraint and will be cost-efficient at the same time. $\mathbf{A} = [z_1, \ldots, z_n]$ defines the number of instances of every configuration type ($[X_1, \ldots, X_n]$) required to be used. Let us define k as the number of the least powerful configurations needed to meet the E_i level. Then the maximum reasonable value for z_i ($i \in (1, \ldots, n)$) while finding $\mathbf{A}$ equals k, as more resources than k will be unnecessarily more expensive because E_i is definitely already met. $\mathbf{A}$ is defined as:

$$\mathbf{A} = [\, z_1, \ldots, z_n \,] \tag{4}$$

where $\forall i \in (1, \ldots, n)\ 0 \leq z_i \leq k, z_i \in \mathbb{N}$. The cost F of set $\mathbf{A}$ is defined as:

$$F(\mathbf{A}, \mathbf{M}) = \mathbf{A} \cdot \mathbf{M} = \mathbf{A} \cdot \begin{bmatrix} m_1 \\ \vdots \\ m_n \end{bmatrix} = \sum_{i=1}^{n} (z_i \cdot m_i) \tag{5}$$

where m_i is the price of the X_i configuration type. The resource level B provided by $\mathbf{A}$ is defined as:

$$B = \mathbf{A} \cdot \begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix} = \sum_{i=1}^{n} (z_i \cdot s_i) \tag{6}$$

where s_i is the resource level provided by the X_i configuration type. Cost definition for minimization algorithm D is as follows:

$$D(F, B, E_i) = \begin{cases} F & \text{if } B \geq E_i \\ \infty & \text{otherwise} \end{cases} \tag{7}$$

$\mathbf{A}$ with the minimal cost can be found using the cost function D using equation (7) and the Integer-PSO algorithm. As the cloud providers' pricing policies are usually complex, it is impossible to define how many minimums exist in the cost function, which is discrete, since a fractional component cannot be provisioned. The arrangement of E_i as a vector (where many resource types exist, as mentioned above) also changes B to a vector; in such case $B \geq E_i$ means that every value of both vectors satisfies the condition. To reduce frequent configuration changes, the *Prediction module* uses a cool-down technique which forces, for some subsequent hours, the $\mathbf{A}$ calculated previously if it still meets the $B \geq E_i$ constraint.

In a separate hourly loop, the *Monitoring module* monitors if resources must be scaled according to the predicted configuration. In case of a configuration change, new resources are provisioned and warmed up before old ones are disposed of, which ensures continuity and the desired QoS parameters. As the warm-up time (30 seconds to 5 minutes depending on IaaS/PaaS and machine type) is rather short, and warm-up occurs usually only once or twice a day, its impact is negligible from the cloud cost point of view. Both loop frequency and prediction resolution can be altered if needed.

The execution process described is depicted in the form of a diagram (Fig. 3), with the arrow directions representing data flow. Additionally, for clarity purposes, the complete solution design, illustrating its innovative workflow and the purpose of the discovery and execution stages, is presented in the form of an algorithm with explanatory comments (Alg. 1).

To measure the efficiency of the LTPS, we defined the P metric, which is the mean savings in terms of money achieved through the optimization performed during hour $t = 1, 2, \ldots, m$ (where m is the number of hours tested). P is defined as:

$$P = \frac{1}{m} \cdot \sum_{t=1}^{m} \left(\frac{N_t - O_t}{N_t} \right) \tag{8}$$

where N_t is the cost of the system for hour t observed before the optimization and O_t is the cost of the system for hour t observed after the optimization. The P metric is calculated based on $\mathbf{A}$ which reflects the fact that fractional components cannot be provisioned, and P is expressed as a percentage.

4 Evaluation

Using the approach described in Section 3, we developed an optimization solution and set up a test environment (Fig. 1). The optimization solution was deployed in the Azure (Microsoft's cloud service) environment and connected to the *Working system* and the *Copy of the system*. The *Working system* is a real-life working system called the Terminal Management System (TMS), which is a cloud-based manager of Internet of Things devices [3]. TMS consists of many endpoint devices (credit card payment devices

Algorithm 1: Discovery and execution stage algorithm.

```
// read minimal and maximal values of load resource levels
(minLoad,maxLoad) = ReadLoadHistoryMinMax(system = 'WorkingSystem');
(minResource,maxResource) = ReadResourceHistoryMinMax(system = 'WorkingSystem');
// create linear spread vectors for load and resource
linearSpreadVectorLoad = LinSpread(minLoad,maxLoad,levels = 20);
linearSpreadVectorResource = LinSpread(minResource,maxResource,levels =20);
// for every combination of load and resource levels perform load emulation and store results
foreach resource of the linearSpreadVectorResource do
end
foreach load of the linearSpreadVectorLoad do
    time = PerformLoadEmulation(system = 'CopyOfTheSystem');
    responseTimes.Add('R' = resource, 'L' = load, 'T' = time);
end
// remove anomalies from collected data
responseTimes = RemoveAnomalies(responseTimes);
// create a QoS ML model using collected data
qosMLModel = MachineLearningFit(features = responseTimes['L','T'],labels = responseTimes['R']);
// end of the discovery stage, start infinite loop of the execution stage
while True do
    // read historical load data from the last 28 days and remove anomalies from them
    loadHistory = GetHistoryData(days = '28');
    loadHistory = RemoveAnomalies(loadHistory);
    // create a load ML model using historical data
    loadMLModel = MachineLearningFit(features = loadHistory['dayOfWeek','hour'],labels =
      loadHistory['loadLevel']);
    // generate time set for the oncoming week (all days of the week with all possible hours)
    predictionTimeSet = GeneratePredictionTimeSet(dayOfWeek = [0..6], hour = [0..23]);
    // perform load level predictions for the next week
    loadPredictions['L'] = loadMLModel.Predict(predictionTimeSet['dayOfWeek','hour']);
    // set expected response time and perform resource level predictions for the next week
    expectedResponseTime = 3;
    loadPredictions['T'] = expectedResponseTime;
    resourcePredictions['R'] = qosMLModel.Predict(loadPredictions['L','T']);
    // get current pricing schema from a cloud provider
    pricing = GetPricingFromCloudProvider();
    // for every predicted resources level calculate and store cost-optimal cloud resources configuration
    foreach dayOfWeek,hour,resourceLevel of the
      predictionTimeSet['dayOfWeek','hour'],resourcePredictions['R'] do
        hourlyConfiguration = IntegerPSOCalculate(resourceLevel,pricing);
        configuration.Add('dayOfWeek' = dayOfWeek, 'hour' = hour, 'configuration' = hourlyConfiguration);
    end
    // for every hour in the forthcoming week
    foreach dayOfWeek,hour,hourlyConfiguration of the
      configuration["dayOfWeek",'hour','configuration'] do
        // check if current cloud resources differ from calculated configuration
        if CurrentResources(system = 'WorkingSystem') != hourlyConfiguration then
            // and scale them if necessary
            ScaleResources(system = 'WorkingSystem',hourlyConfiguration);
        end
        // wait until the next hour
        Sleep(hours = 1);
    end
end
```

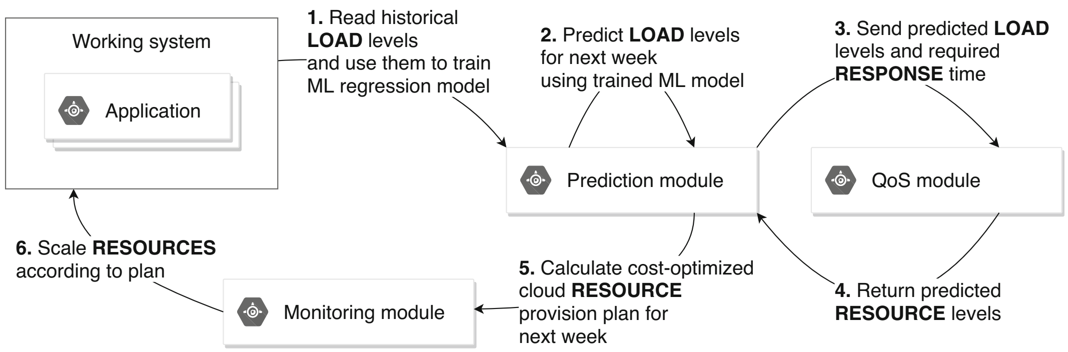

Working system
Application
1. Read historical LOAD levels and use them to train ML regression model
2. Predict LOAD levels for next week using trained ML model
3. Send predicted LOAD levels and required RESPONSE time
Prediction module
QoS module
4. Return predicted RESOURCE levels
5. Calculate cost-optimized cloud RESOURCE provision plan for next week
Monitoring module
6. Scale RESOURCES according to plan

Fig. 3 Execution stage – operation sequence and data flow

in vending machines and kiosks) which connect to the central server. The central server, located in the Azure cloud, processes payments and provides web-based access in order to configure and maintain end-point devices. Payment transactions are processed by micro-services (PaaS) while management/reporting web pages are hosted by virtual machines (IaaS). As these are the only applications in the TMS system, we will use their names (payment micro-services, web pages) and cloud deployment models (PaaS, IaaS) interchangeably further on. Both micro-services and web page servers share the same SQL database, but work independently from one another and therefore can be optimized individually. All steps of the evaluation process were performed separately for payment micro-services and web page servers. The *Copy of the system* was also deployed in the Azure cloud; it uses the same architecture and cloud parameters as the *Working system*; however, we had to use an emulator for external payment gateways, which are third-party services normally used to process payment transactions. As payment gateways introduce significant delays, emulator characteristics were based on historical delay data from the production system.

Any implementation of a cloud resource optimization solution, especially for production deployment, should take into account many additional issues with the foremost one being warm-up delays (which are accounted for in the LTPS). As described in Section 3.2, the resources provisioned ought to be warmed up before usage and this delay should be included in the resource scaling process. As the scaling process not only implies warm-up delays but may also result in various kinds of errors (e.g. the unavailability of new resources), it is good practice to minimize scaling events, e.g. by the introduction of the cool-down effect as illustrated in Section 3.2 (also implemented in the LTPS). In the case of the aforementioned deficiency of new resources, depending on the priorities of the system optimized (high availability or strict cost constraints), the optimization solution may try to provide slightly better or worse resources. The LTPS tries to provide better resources in such cases. An optimization solution should provide means of monitoring throughout its operation, and when deployed in a production environment it should enable provisioning plan review and correction. This monitoring mechanism has also been included in our implementation.

The rest of this section is structured as follows: Section 4.1 presents validation results (according to the description in Section 3.1), Section 4.2 is concerned with the optimization process (according to the description in Sections 3.1 and 3.2), Section 4.3 contains a comparison of the LTPS to related works and illustrates the efficiency of the LTPS in different cloud providers' environments.

4.1 Validation

The *Copy of the system's* micro-services and web page servers were validated (each one independently) using the *Load emulator* and real-life historical load data collected during the 1st week of December 2019. To ensure that the *Load emulator* reflected load requests accurately (despite using the external payment gateway emulator), we compared historical load data from the *Working system* with actual load data captured from the *Copy of the system* during the validation process. Load levels, both historical and observed during the test, are presented in Fig. 4a (IaaS) and Fig. 5a (PaaS). All parameters were collected using an Azure

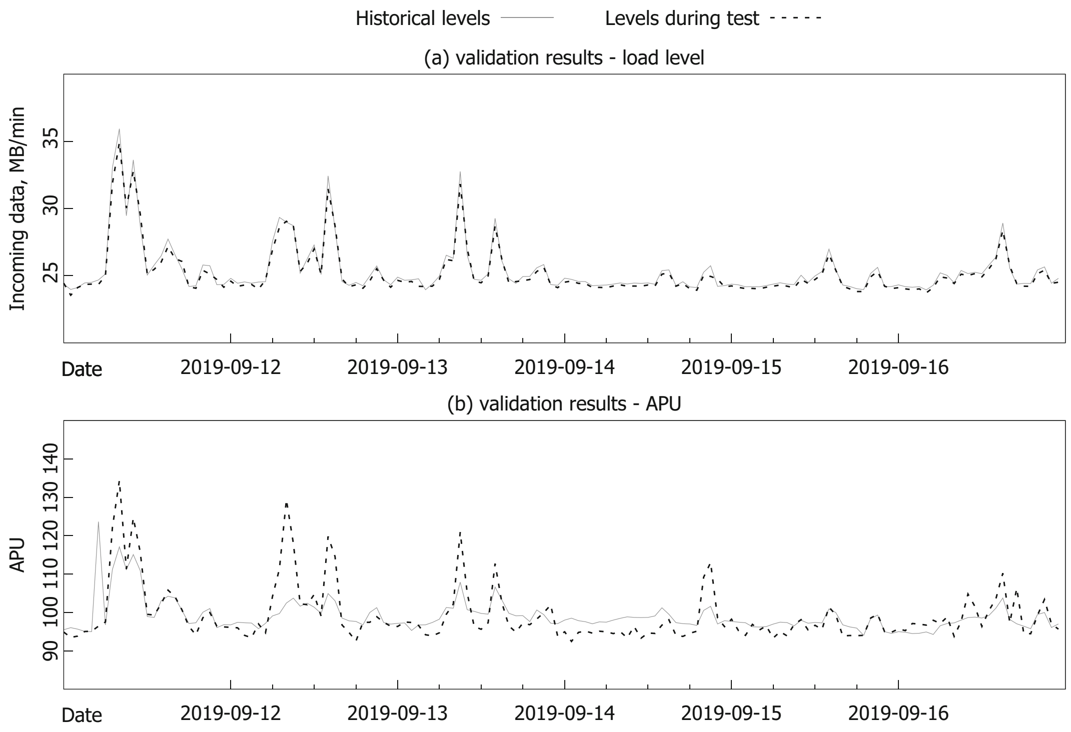


Fig. 4 *Copy of the system* validation results (IaaS)

API, which exposes getting and setting a component's parameters and makes a component's historical usage available for reading. We used this API for all further operations on Azure components. For both micro-services (PaaS) and web page servers (IaaS), we used the CPU utilization level as a reference resource; additionally, we used average response time for PaaS as Azure provides this metric for PaaS components. Usage data (historical and recorded during testing) are presented in Fig. 4b (IaaS), Fig. 5b (PaaS) and Fig. 5c (PaaS).

Using data collected during the validation process, we calculated *V* metric values as defined in equation (3) and presented them in Table 1.

Average response time results are the least accurate; this is caused by the fact that, as mentioned before, we were unable to use 3rd party external components. Moreover, in the production system, we observed anomalous random response time peaks caused by network conditions between the *Working system* and external payment gateways (Fig. 5c), which affected the *V* metric value. Nevertheless, archived accuracy levels show that, despite using an emulator for third-party payment servers, the *Copy of the system* reflects *Working system* characteristics to a large degree.

4.2 Optimization

The optimization solution developed is responsible for both the *Discovery stage* (Fig. 2) and the *Execution stage* (Fig. 3). As mentioned before, the *Working system* consists of IaaS and PaaS components. As the resource, we selected an Azure Compute Unit (*ACU*), which represents a unit of computing performance. As the QoS parameter, we selected the system response time. As the load parameter (*Load*), we used requests per second in the case of PaaS and incoming traffic (inbound) volume[4] in the IaaS case. For emulation purposes, based on historical data from the production system, we determined (separately for IaaS and PaaS) minimum and maximum levels for the *ACU* and *Load* parameters. For *Load*, as defined in Section 3, 10 and 20 equally distributed levels from the defined range were used during load tests. For *ACU*, all available

[4]Inbound traffic refers to information coming into a network, in this case into a specific virtual machine.

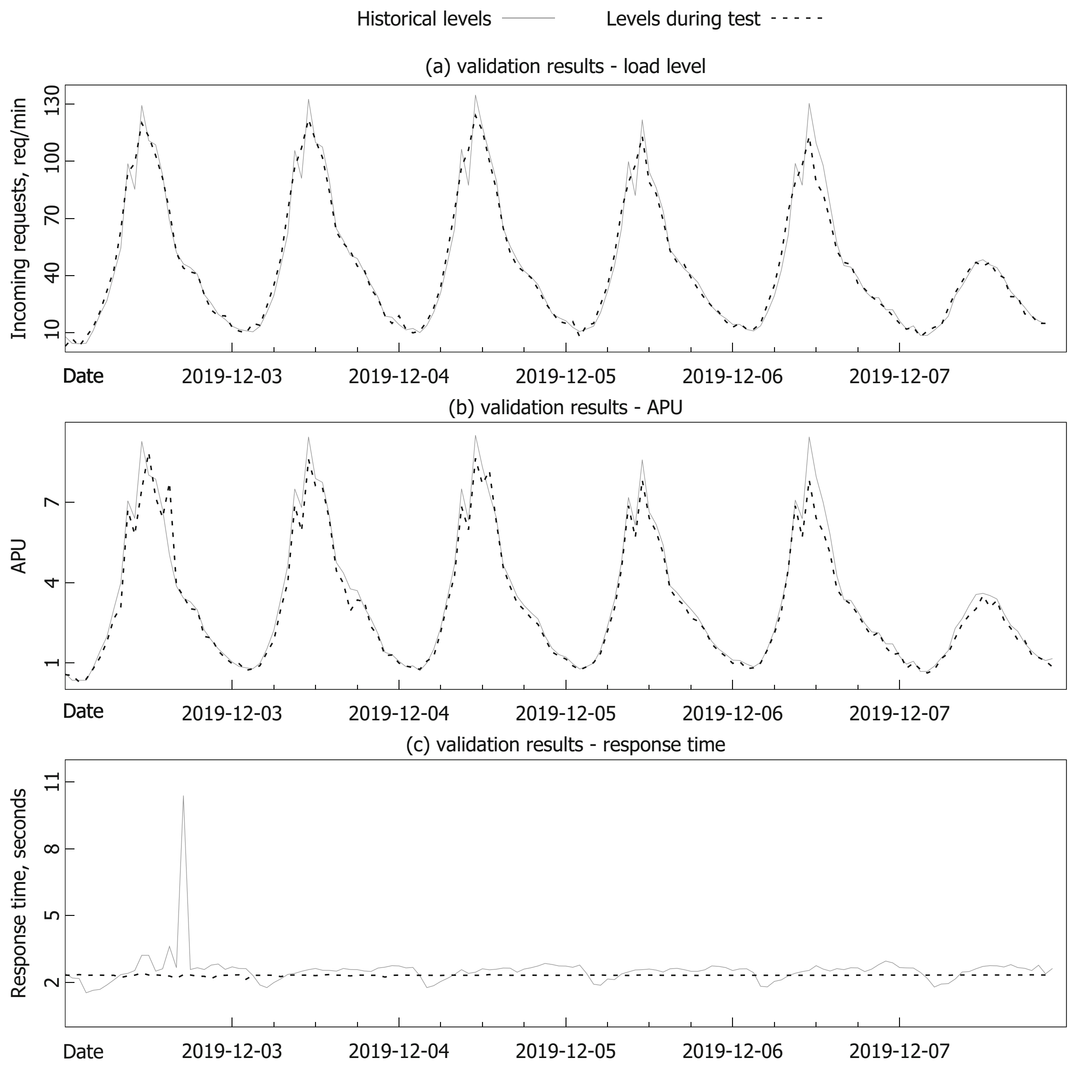


Fig. 5 *Copy of the system* validation results (PaaS)

Azure machine types (from the defined range) were used (Table 2). Next, we performed load emulations for all calculated *Load* and *ACU* value combinations.

Load emulation results for IaaS (Fig. 6a) and PaaS (Fig. 6b) show that there is a discernible relationship between load, system parameters and response time.

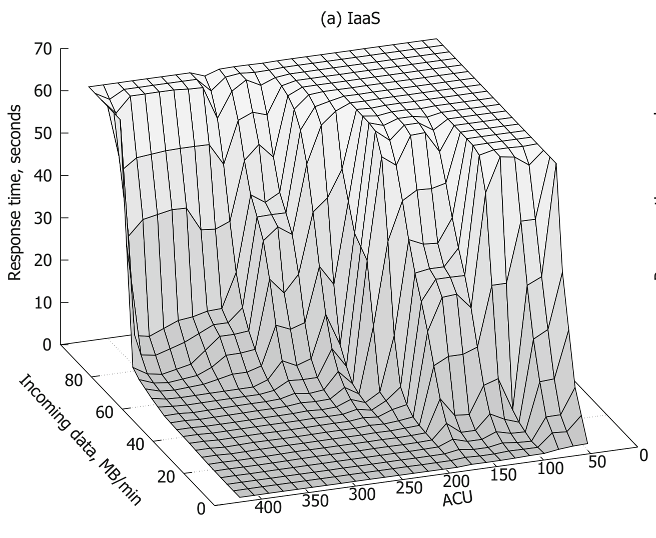
Table 1 *Copy of the system* validation results (*V* metric)

	Load level	CPU utilization	Average response time
IaaS	99.03%	96.89%	–
PaaS	89.75%	91.43%	86.86%

We selected Microsoft Azure Machine Learning Studio (AMLS) as our machine learning engine because it is part of the Azure environment. AMLS offers ready-to-use components with in-built data

Table 2 Data used during QoS emulation process

Cloud service model	Resource	Minimum value	Maximum value
IaaS	ACU	50 ACU	420 ACU
	Network In	2 MB/minute	100 MB/minute
PaaS	ACU	50 ACU	800 ACU
	Request	20 req./minute	1600 req./minute



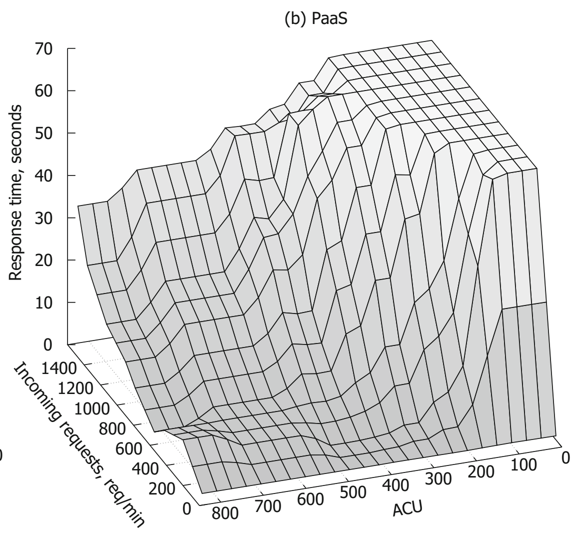


Fig. 6 Response times observed under load

processing and ML capabilities along with full customization thanks to R and Python language support. We also used AMLS to implement the anomaly filter. We filtered out anomalies from QoS load results and used them to create QoS models, as described in Section 3. We implemented four different prediction types: Bayesian Linear Regression (BL) [33], Neural Network Regression (NN) [25], Poisson Regression (P) [9] and Decision Forest Regression (DF) [30], and tested them on QoS data from both PaaS and IaaS. All prediction types used the "Tune Model Hyperparameters"[5] self-tuning mode, which picks optimal prediction algorithm parameters automatically. The values used as an initial configuration for self-tuning are presented for Bayesian Linear Regression in Table 3, for Neural Network Regression in Table 4, for Poisson Regression in Table 5 and for Decision Forest Regression in Table 6.

To estimate the quality of the trained models, we used common metrics: Relative Absolute Error (RAE) and Relative Squared Error (RSE). As data obtained from QoS emulations were sparse in terms of response time, we used all of them as training data, thus the models' quality parameters mentioned above were calculated using the cross-validation technique [37]. Calculated values of the aforementioned parameters are presented in Table 7, and on this basis, we considered DF as a QoS prediction ML algorithm as it exhibits the best results in all cases (but one in which it is almost as good as P).

We tested the LTPS using real-life data from the TMS system collected during periods involving significant load variances (from 25th to 31st March 2019 for IaaS and from 9th to 21st December 2019 for PaaS). The efficiency of the LTPS was demonstrated by time-compressed tests, in which every minute represents an hour of normal time (an acceleration factor of 60). As our optimization process normally uses hourly resolution for the historical data obtained and the predictions made, hence in a time-compressed scenario it used just minutes instead of hours, thus significantly reducing the testing time, as resources changed 60 times more frequently than under normal conditions. Provisioning delays (normally negligible) became important and were taken into account during the test process.

To use the time-compressed approach, we set up separate test environments for PaaS and IaaS (Fig. 7).

The *Load emulator* read load data from the *Working system* (for the aforementioned test period). At the same time, the *Prediction module* used historical load data from the *Working system* to create load level

[5]Tune Model Hyperparameters – https://docs.microsoft.com/en-us/azure/machine-learning/studio-module-reference/tune-model-hyperparameters

Table 3 Initial configuration for self-tuning of Bayesian Linear Regression algorithm

Regularization weight	1
Tune Model Hyperparameters maximum number of runs	50

Table 4 Initial configuration for self-tuning of Neural Network Regression algorithm

Hidden layers	1, fully connected
Number of hidden nodes	100
Learning rate	0.02
Number of iterations	80
Initial learning weights diameter	0.1
Momentum	0
Normalizer type	Do not normalize
Tune Model Hyperparameters maximum number of runs	50

predictions (again for the test period). Load prediction results along with actual load levels from historical data are presented in Figs. 8a and b. Afterwards, using the *QoS module*, the *Prediction module* created a prediction of the resource level required (*ACU*) and calculated a cost-optimized resource provision plan. Based on this plan, the *Monitoring module* performed the scaling of the *Copy of the system's* resources. Both the *Load emulator* and the *Monitoring module* adjusted their actions to the time-compressed scenario (i.e. accelerated their internal clocks 60 times). In the end, the *Load emulator*, according to the load data previously read, performed a load emulation on the *Copy of the system* and gathered response times.

We performed these tests for two desired response time scenarios: strict and lenient, which allowed us to compare the results. In the strict scenario, we adopted 0.7 s for IaaS and 3 s for PaaS, and in the lenient one, we used 7 s for both IaaS and PaaS. Additionally, both strict and lenient scenarios were tested with QoS models trained using 10 and 20 levels. As the PaaS cloud model allows us to acquire historical response times, we also trained for PaaS an additional QoS model with raw historical data without a QoS emulation (0 levels). We performed ten tests in total (four for IaaS and six for PaaS), which allowed us to compare prediction

Table 5 Initial configuration for self-tuning of Poisson Regression algorithm

Optimization tolerance	$1.0 \cdot 10^{-7}$
L1 regularization weight	1
L2 regularization weight	1
Memory size for L-BFGS	200
Tune Model Hyperparameters maximum number of runs	50

Table 6 Initial configuration for self-tuning of Decision Forest Regression algorithm

Re-sampling method	Bagging
Number of decision trees	8
Maximum depth of decision trees	32
Number of random splits per node	128
Minimum number of samples per leaf node	1
Tune Model Hyperparameters maximum number of runs	50

quality for all these scenarios. The solution determined the optimal cloud resource configuration using the modified Integer-PSO algorithm (as described in Section 3) with 3,000 particles in 3,000 epochs. Inertia weight was set to 0.6 and acceleration coefficients to 0.2, as described in [5]. The maximum and minimum velocity were set to $n/7$ (with opposite signs) where n is the number of available configuration options; accuracy was set to 3 digits. Based on test results, we calculated P metric values which, as defined in equation (8), represent percentage savings the LTPS would have generated if implemented in the *Working system*. Additionally, we measured average response times (ART) and presented the aforementioned results in Table 8.

When compared to the original cost, the high savings percentage figure is caused by considerable resource over-provisioning in the TMS system as the *Working system* is scaled up with a margin for both unforeseen load spikes and long-term load changes. Due to this fact, we also performed a cost comparison with the system that was optimized by other means, as described in Section 4.3. The load emulations performed improved prediction efficiency; the best results were obtained for QoS models with 20 levels of load emulation, while the QoS model trained without additional emulated data (PaaS 0 levels) exhibited mediocre performance. The lower-than-required average response time values, even for the best QoS model, resulted from a granularity of cloud resources which resulted in minor overprovisioning.

The optimization solution presented runs in a separate cloud environment and optimization calculations do not affect the *Working system* which is being optimized in terms of performance overhead. The QoS model is created only once, the prediction module, as described in Section 3, due to the long-term prediction ability, runs once a week, the monitoring module runs

Table 7 QoS model parameters (RAE and RSE)

Cloud service model	Metric	BL	NN	P	DF
IaaS (10 levels)	RAE	0.9832	1.7167	0.6731	0.5012
	RSE	1.4775	3.5701	0.5772	0.4036
IaaS (20 levels)	RAE	1.1480	1.0160	0.7566	0.5144
	RSE	1.7082	1.1699	0.6141	0.3795
PaaS (10 levels)	RAE	1.0420	0.8733	0.7336	0.5941
	RSE	1.3195	1.0599	0.6429	0.4623
PaaS (20 levels)	RAE	0.9878	0,9439	0.7024	0.6901
	RSE	1.0505	1.1505	0.5459	0.5631

only when a component change is required (usually once or twice a day), and ML calculations are performed by Microsoft Azure Machine Learning Studio. The free plans offered by Azure cover these operations, and thus the cost of running the LTPS does not affect the savings calculated.

4.3 Comparison

As we did not find any test data or solutions to compare them with our system, we used as a reference point the Azure Autoscale mechanism,[6] which is a scaling mechanism included in the Azure cloud environment. Unfortunately, it only offers time- and threshold-based scaling and only performs a horizontal (quantity) optimization. Vertical (quality) optimization is not available, and that is why we used the cheapest possible components to ensure the most detailed scaling. We performed Autoscale tests for both IaaS and PaaS using the periods selected before (from 25th to 31st March for IaaS and from 9th to 21st December for PaaS). Autoscale does not enable scaling according to response time, thus we used CPU utilization percentage as the scaling parameter. We performed tests with two options: more aggressive scaling settings (scale up when utilization is greater than 50% and scale down when it is less than 15%) and less aggressive settings (scale up when utilization is greater than 70% and scale down when it is less than 25%). More aggressive scaling exhibited better performance and reduced cost to some degree, but the LTPS was 25% more efficient in the strict response time scenario and 49% more efficient in the lenient response time scenario. The periods tested consisted of periodical usage increases, and since Azure Autoscale is a reactive system, it only performed scaling after usage had already increased, which introduced transient but significant response time increases. Thanks to anomaly detection and machine learning, our optimization solution properly predicted usage changes, and scaling was performed in advance, demonstrating industrial-grade readiness and capability of being deployed in commercial environments.

As mentioned in Section 2, it is difficult to compare the optimization efficiency of the LTPS against the results quoted in referenced works, as different rating methods were used. Nevertheless, we gathered, where it was possible, optimization results compared to state-of-the-art solutions and presented them in Table 9. Since different datasets, reference solutions and evaluation techniques were used, we presented an approximate comparison to illustrate the magnitude of optimization efficiency rather than detailed results. The efficiency of the LTPS is in most cases higher than that of those from referenced works, demonstrating the pertinence of our innovative two-staged approach.

Although we tested the optimization solution using the Azure cloud, it is possible to use any cloud provider which exposes an API for resource management. As such APIs are available for the Google Cloud Platform (GCP)[7] and Amazon Web Services (AWS)[8][9], which are major cloud providers, we also conducted savings calculations for AWS and GCP. Due to differences in pricing schemes between cloud providers, the cost of the TMS system without optimization would be different in individual cloud

[6]Azure Autoscale – https://azure.microsoft.com/en-us/features/autoscale

[7]GCP Cloud APIs – https://cloud.google.com/apis

[8]AWS EC2 API – https://docs.aws.amazon.com/AWSEC2/latest/APIReference

[9] Elastic AWS Beanstalk API – https://docs.aws.amazon.com/elasticbeanstalk/latest/api

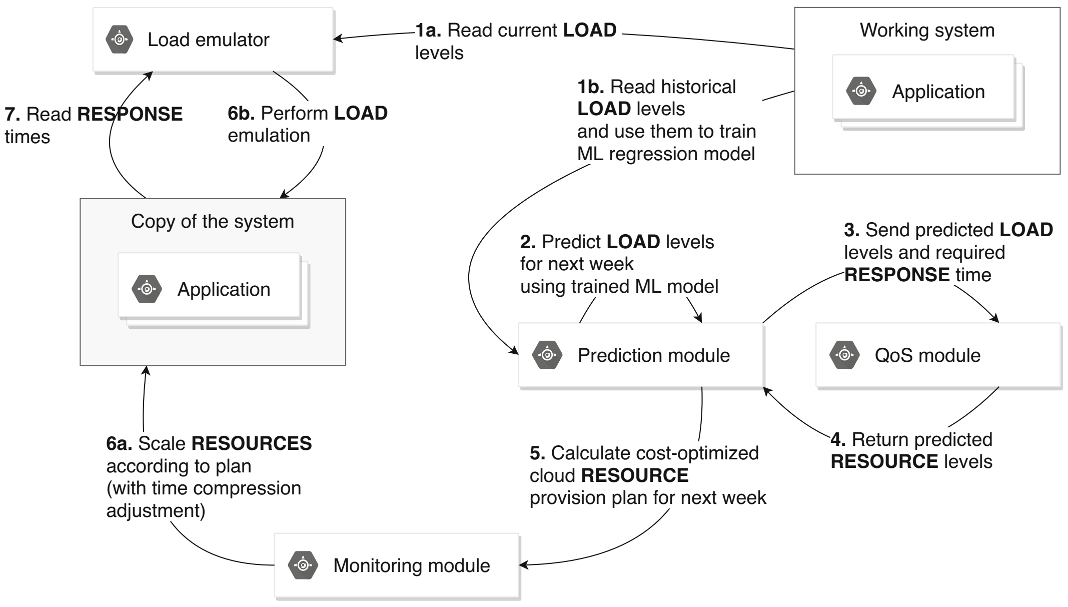

Load emulator
1a. Read current LOAD levels
Working system
Application
1b. Read historical LOAD levels and use them to train ML regression model
7. Read RESPONSE times
6b. Perform LOAD emulation
Copy of the system
Application
2. Predict LOAD levels for next week using trained ML model
3. Send predicted LOAD levels and required RESPONSE time
Prediction module
QoS module
4. Return predicted RESOURCE levels
6a. Scale RESOURCES according to plan (with time compression adjustment)
5. Calculate cost-optimized cloud RESOURCE provision plan for next week
Monitoring module

Fig. 7 Architecture of time-compressed test environment PaaS/IaaS

environments. Also, the same components are differently priced and the price difference is not proportional to the component's quality. Additionally, in some cases, we were unable to find Azure counterparts in GCP and AWS environments and we were forced to use estimates. Therefore, the data presented

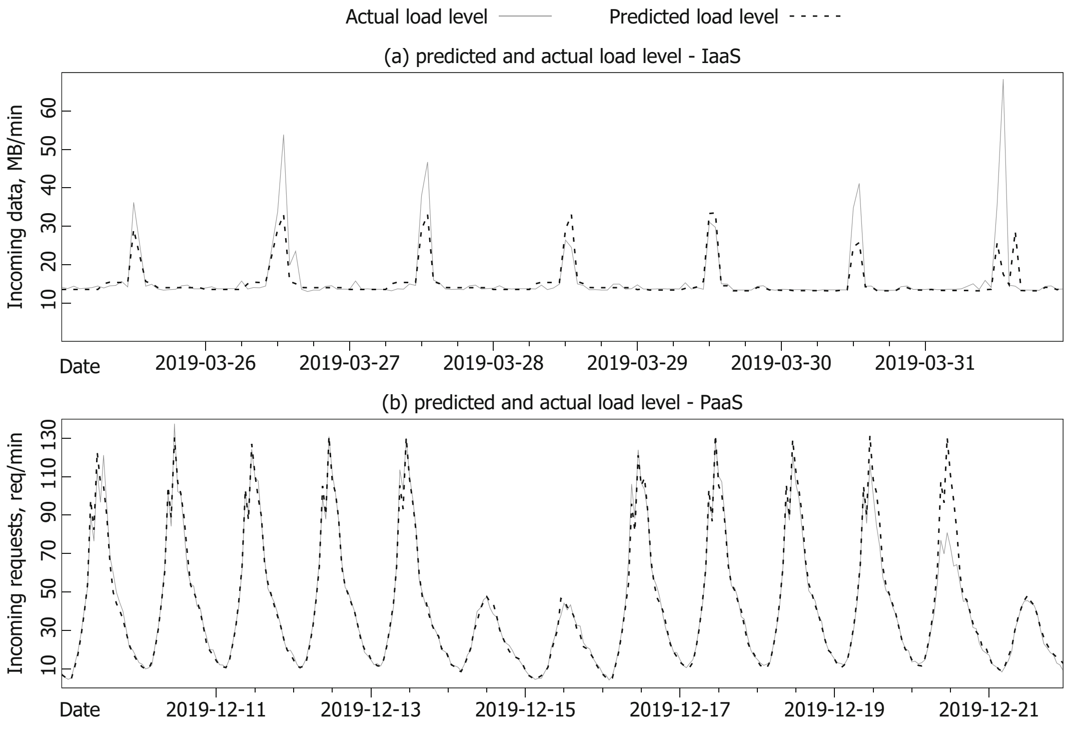


Fig. 8 Load prediction results for IaaS and PaaS model

Table 8 Prediction efficiency results (P metric) and average response times (ART)

Cloud service model	P for strict requirements	P for lenient requirements	ART for strict requirements	ART for lenient requirements
IaaS (10 levels)	70%	80%	0.68 sec.	1.76 sec.
IaaS (20 levels)	72%	85%	0.56 sec.	6.11 sec.
PaaS (0 levels)	27%	27%	2.37 sec.	2.46 sec.
PaaS (10 levels)	47%	32%	2.42 sec.	3.60 sec.
PaaS (20 levels)	51%	63%	2.66 sec.	6.41 sec.

in Table 10 are coarse estimates which intend to present overall differences between cloud providers rather than exact values.

5 Conclusions

In this work, we present an approach to optimizing resource usage while observing the required QoS constraints. The solution proposed is a complete system which consists of two stages: the *Discovery stage* which creates a QoS machine learning model and the *Execution stage* which uses this QoS ML model, predicts resource levels and scales optimized system components. Both stages are assisted by anomaly detection and machine learning. Our tests, which were performed with real-life data from a production system, demonstrated significant cost reductions even for strict response time constraints. Comparing current system costs with those after optimization we observed that the solution would have resulted in savings ranging from 72% to 85% for IaaS (depending on server response time requirements) and from 51% to 63% for PaaS if it had been applied to the *Working system* deployed in Azure. The cost reduction observed would have been even greater for a system deployed using AWS (a 75%-86% cost reduction) but smaller for GCP (a 35%-64% cost reduction).

The LTPS, when evaluated, performed better (by 25–49%) than a commercial solution in Azure; a rough comparison with state-of-the-art solutions (Table 9) shows that our approach performed better in most cases. As a result of an innovative two-staged approach, anomaly detection and dedicated Integer-PSO algorithm, our implementation brings QoS-constrained optimization efficiency improvements along with industry-grade readiness.

The LTPS is most suited for systems with usage patterns that are complicated, difficult to formalize and varied over time. Such patterns will be determined by machine learning algorithms and will be used along with QoS parameters to predict the system resource provisioning plan. The best results will be obtained for systems with stateless communications where every request is treated independently and scaling may be performed freely, e.g. IoT hubs, payment gateways, the server side of client-server systems, web information portals or social networks. Scaling of stateful communications is also offered by cloud providers albeit long communication sessions might disrupt scaling efficiency.

As presented in [31], initial training can be successfully used in resource prediction solutions. As the *Discovery stage* is a time-consuming process which requires the *Copy of the system*, reusing trained QoS models between different systems and adjusting them over time without the need for the *Copy of the system*

Table 9 Approximate comparison of different solutions' resource optimization efficiency

IE-ABC [35]	BULLET [15]	MBSA [19]	LTQA [14]	CML [29]
20%	3-9%	1-2%	20%	34%
SFBL [21]	RCRP [8]	Hybrid PSO [26]	MAPSO [1]	GABVMP [2]
28-45%	2%	25%	15-22%	22%
DASC [10]	MqPM [40]	C2RAM [32]	HRHA [4]	LTPS
20%	1-26%	10%	30%	25-49%

Table 10 Comparison of system daily operation costs for strict and lenient response time requirements in the Azure, AWS and GCP cloud environments

Cloud provider	Azure	AWS	GCP
IaaS			
Cost before optimization	€5.50/day	€7.87/day	€7.49/day
Cost after optimization for strict req.	€1.54/day	€1.73/day	€4.35/day
- savings	72%	78%	42%
Cost after optimization for lenient req.	€0.83/day	€1.10/day	€2.70/day
- savings	85%	86%	64%
PaaS			
Cost before optimization	€2.43/day	€3.47/day	€3.30/day
Cost after optimization for strict req.	€1.19/day	€0.87/day	€2.15/day
- savings	51%	75%	35%
Cost after optimization for lenient req.	€0.90/day	€0.80/day	€2.05/day
- savings	63%	77%	38%

may be the subject of our further research. We would also like to adapt our idea to systems that use Kubernetes [23] and test it on a system deployed in such an environment.

Acknowledgements The research presented in this paper was supported by funds from the Polish Ministry of Science and Higher Education assigned to the AGH University of Science and Technology.

Data Availability The data that support the findings of this study are available from OTI Europa ASEC but restrictions apply to the availability of these data, which were used under license for the current study, and so are not publicly available. Data are, however, available from the authors upon reasonable request and with permission of OTI Europa ASEC.

Compliance with Ethical Standards

Conflict of Interests The authors declare that they have no conflict of interest.

References

1. Adhikari, M., Amgoth, T.: Multi-objective accelerated particle swarm optimization technique for scientific workflows in iaas cloud. In: 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp. 1448–1454. IEEE (2018)
2. Akintoye, S.B., Bagula, A.: Improving quality-of-service in cloud/fog computing through efficient resource allocation. Sensors **19**(6), 1267 (2019)
3. Alessio, B., De Donato, W., Persico, V., Pescapé, A.: On the integration of cloud computing and internet of things. In: Proc. Future Internet of Things and Cloud (FiCloud), pp. 23–30 (2014)
4. Ardagna, D., Ciavotta, M., Lancellotti, R., Guerriero, M.: A hierarchical receding horizon algorithm for qos-driven control of multi-iaas applications. IEEE Transactions on Cloud Computing (2018)
5. Beegom, A.A., Rajasree, M.: Integer-pso: a discrete pso algorithm for task scheduling in cloud computing systems. Evol. Intel. **12**(2), 227–239 (2019)
6. Bheda, H.A., Lakhani, J.: Qos and performance optimization with vm provisioning approach in cloud computing environment. In: 2012 Nirma University International Conference on Engineering (NUiCONE), pp. 1–5. IEEE (2012)
7. Calheiros, R.N., Ranjan, R., Buyya, R.: Virtual machine provisioning based on analytical performance and qos in cloud computing environments. In: 2011 International Conference on Parallel Processing, pp. 295–304 (2011)
8. Chaisiri, S., Lee, B.S., Niyato, D.: Robust cloud resource provisioning for cloud computing environments. In: 2010 IEEE International Conference on Service-Oriented Computing and Applications (SOCA), pp. 1–8. IEEE (2010)
9. Chan, L., Silverman, B.W., Vincent, K.: Multiple systems estimation for sparse capture data: Inferential challenges when there are nonoverlapping lists. J. Am. Stat. Assoc., 1–10 (2020)

10. Chen, Y., Huang, J., Lin, C., Shen, X.: Multi-objective service composition with qos dependencies. IEEE Transactions on Cloud Computing **7**(2), 537–552 (2019)
11. Cherubin, G., Baldwin, A., Griffin, J.: Exchangeability martingales for selecting features in anomaly detection. In: Conformal and Probabilistic Prediction and Applications, pp. 157–170 (2018)
12. Comi, A., Fotia, L., Messina, F., Pappalardo, G., Rosaci, D., Sarné, G.M.L.: A reputation-based approach to improve qos in cloud service composition. In: 2015 IEEE 24th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, pp. 108–113 (2015)
13. Evangelinou, A., Ciavotta, M., Ardagna, D., Kopaneli, A., Kousiouris, G., Varvarigou, T.: Enterprise applications cloud rightsizing through a joint benchmarking and optimization approach. Futur. Gener. Comput. Syst. **78**, 102–114 (2018)
14. Gavvala, S.K., Jatoth, C., Gangadharan, G., Buyya, R.: Qos-aware cloud service composition using eagle strategy. Futur. Gener. Comput. Syst. **90**, 273–290 (2019)
15. Gill, S.S., Buyya, R., Chana, I., Singh, M., Abraham, A.: Bullet: particle swarm optimization based scheduling technique for provisioned cloud resources. J. Netw. Syst. Manag. **26**(2), 361–400 (2018)
16. Goiri, Í., Julià, F., Fitó, J.O., Macías, M., Guitart, J.: Resource-level qos metric for cpu-based guarantees in cloud providers. In: International Workshop on Grid Economics and Business Models, pp. 34–47. Springer (2010)
17. Hayyolalam, V., Kazem, A.A.P.: A systematic literature review on qos-aware service composition and selection in cloud environment. J. Netw. Comput. Appl. **110**, 52–74 (2018)
18. He, F., Sato, T., Chatterjee, B.C., Kurimoto, T., Urushidani, S., Oki, E.: Robust optimization model for backup resource allocation in cloud provider. In: 2018 IEEE International Conference on Communications (ICC), pp. 1–6. IEEE (2018)
19. Jian, C., Li, M., Kuang, X.: Edge cloud computing service composition based on modified bird swarm optimization in the internet of things. Clust. Comput. **22**(4), 8079–8087 (2019)
20. Jiang, W., Lee, D., Hu, S.: Large-scale longitudinal analysis of soap-based and restful web services. In: 2012 IEEE 19th International Conference on Web Services, pp. 218–225 (2012)
21. Kumar, T.S.: Efficient resource allocation and qos enhancements of IoT with fog network. Journal of ISMAC **1**(02), 101–110 (2019)
22. Li, J., Chinneck, J., Woodside, M., Litoiu, M., Iszlai, G.: Performance model driven qos guarantees and optimization in clouds. In: 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing, pp. 15–22. IEEE (2009)
23. Medel, V., Tolosana-Calasanz, R., Bañares, J.Á., Arronategui, U., Rana, O.F.: Characterising resource management performance in kubernetes. Computers & Electrical Engineering **68**, 286–297 (2018)
24. Mehmood, T., Latif, S., Malik, S.: Prediction of cloud computing resource utilization. In: 2018 15th International Conference on Smart Cities: Improving Quality of Life Using ICT & IoT (HONET-ICT), pp. 38–42. IEEE (2018)
25. Mersy, G., Santore, V., Rand, I., Kleinman, C., Wilson, G., Bonsall, J., Edwards, T.: A comparison of machine learning algorithms applied to american legislature polarization. arXiv:2008.04072 (2020)
26. Naseri, A., Navimipour, N.J.: A new agent-based method for qos-aware cloud service composition using particle swarm optimization algorithm. J. Ambient. Intell. Humaniz. Comput. **10**(5), 1851–1864 (2019)
27. Oliveira, T., Barbar, J., Soares, A.: Computer network traffic prediction: A comparison between traditional and deep learning neural networks. Int. J. Big Data Intell. **3**, 28 (2016)
28. Osypanka, P., Nawrocki, P.: Resource usage cost optimization in cloud computing using machine learning. IEEE Transactions on Cloud Computing, 1–1 (2020)
29. Rahman, S., Ahmed, T., Huynh, M., Tornatore, M., Mukherjee, B.: Auto-scaling vnfs using machine learning to improve qos and reduce cost. In: 2018 IEEE International Conference on Communications (ICC), pp. 1–6 (2018)
30. Rokach, L.: Decision forest: Twenty years of research. Information Fusion **27**, 111–125 (2016)
31. Sniezynski, B., Nawrocki, P., Wilk, M., Jarzab, M., Zielinski, K.: VM reservation plan adaptation using machine learning in cloud computing. Journal of Grid Computing **17**(4), 797–812 (2019)
32. Sun, Y., White, J., Li, B., Walker, M., Turner, H.: Automated qos-oriented cloud resource optimization using containers. Automated Software Engineering **24**(1), 101–137 (2017)
33. Tang, Y.: Beyond em: A faster bayesian linear regression algorithm without matrix inversions. Neurocomputing **378**, 435–440 (2020)
34. Tchernykh, A., Schwiegelsohn, U., Alexandrov, V., Talbi, E.g.: Towards understanding uncertainty in cloud computing resource provisioning. Procedia Computer Science **51**, 1772–1781 (2015)
35. Thanka, M.R., Maheswari, P.U., Edwin, E.B.: An improved efficient: Artificial bee colony algorithm for security and qos aware scheduling in cloud computing environment. Clust. Comput. **22**(5), 10905–10913 (2019)
36. Varshney, S., Sandhu, R., Gupta, P.: Qos based resource provisioning in cloud computing environment: a technical survey. In: International Conference on Advances in Computing and Data Sciences, pp. 711–723. Springer (2019)
37. Wong, T.T., Yang, N.Y.: Dependency analysis of accuracy estimates in k-fold cross validation. IEEE Trans. Knowl. Data Eng. **29**(11), 2417–2427 (2017)
38. Yang, J., Xiao, W., Jiang, C., Hossain, M.S., Muhammad, G., Amin, S.U.: Ai-powered green cloud and data center. IEEE Access **7**, 4195–4203 (2018)
39. Yao, J., Lu, Q., Jacobsen, H.A., Guan, H.: Robust multi-resource allocation with demand uncertainties in cloud

scheduler. In: 2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS), pp. 34–43. IEEE (2017)

40. Ye, Z., Mistry, S., Bouguettaya, A., Dong, H.: Long-term qos-aware cloud service composition using multivariate time series analysis. IEEE Trans. Serv. Comput. **9**(3), 382–393 (2016)
41. Yu, Y., Jindal, V., Bastani, F., Li, F., Yen, I.L.: Improving the smartness of cloud management via machine learning based workload prediction. In: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), vol. 2, pp. 38–44. IEEE (2018)